

**PUBLICATIONS
REVISION**

Operating System/3 (OS/3)

Supervisor

User Guide

UP-8075 Rev. 2

This SPERRY UNIVAC[®] Operating System/3 (OS/3) Library Memo announces the release and availability of "SPERRY UNIVAC Operating System/3 (OS/3) Supervisor User Guide," UP-8075 Rev. 2. This is a Standard Library Item (SLI).

This revision reflects the current status of the OS/3 operating system, including changes and enhancements.

The following have been added:

- The new format of the tape block number field, including the tape mark count, and the revised description of physical IOCS handling of the block count.
- The ALLOC, SCRTCH, and OBTAIN macro instructions for diskette space management.
- The ACCESS parameter in the DTFPF macro instruction for shared filelock processing of SAT disc files.
- The ARGLST macro instruction to generate a parameter list for use in program linkage.
- A change in the GETCS macro instruction to permit records of up to 128-bytes to be read by the control stream embedded data reader.
- A description of system debugging aids available for use by experienced personnel for debugging within the supervisor.
- The switch priority and termination code fields in the job accounting table and in the job accounting record printout format.

Minor changes and additions have been made throughout the manual.

Destruction Notice: This revision supersedes and replaces "SPERRY UNIVAC Operating System/3 (OS/3) Supervisor User Guide," UP-8075 Rev. 1 released on Library Memo dated November, 1975. Also destroyed is Updating Package A, UP-8075 Rev. 1—A released on Library Memo dated March, 1976, Updating Package B, UP-8075 Rev. 1—B released on Library Memo dated September, 1976, Updating Package C, UP-8075 Rev. 1—C released on Library Memo dated April, 1977, and Updating Package D UP-8075 Rev. 1—D released on Library Memo dated May, 1977. Please destroy all copies of UP-8075 Rev. 1, UP-8075 Rev. 1—A, UP-8075 Rev. 1—B, UP-8075 Rev. 1—C, UP-8075 Rev. 1—D and their Library Memos. Additional copies may be ordered by your local Sperry Univac Representative.

| LIBRARY MEMO ONLY | LIBRARY MEMO AND ATTACHMENTS | THIS SHEET IS |
|-----------------------------------|--|------------------------------------|
| Mailing Lists 217, 630 and 692 | Mailing Lists 18, 19, 75 and 76 (Cover and 337 pages) | Library Memo |
| | | RELEASE DATE: October, 1977 |



SPERRY UNIVAC
Operating System/3 (OS/3)
Supervisor

User Guide



SPERRY UNIVAC Operating System/3 (OS/3) Supervisor

User Guide

This document contains the latest information available at the time of preparation. Therefore, it may contain descriptions of functions not implemented at manual distribution time. To ensure that you have the latest information regarding levels of implementation and functional availability, please contact your local Sperry Univac representative.

Sperry Univac reserves the right to modify or revise the content of this document. No contractual obligation by Sperry Univac regarding level, scope, or timing of functional implementation is either expressed or implied in this document. It is further understood that in consideration of the receipt or purchase of this document, the recipient or purchaser agrees not to reproduce or copy it by any means whatsoever, nor to permit such action by others, for any purpose without prior written permission from Sperry Univac.

Sperry Univac is a division of the Sperry Rand Corporation.

FASTRAND, SPERRY UNIVAC, UNISCOPE, UNISERVO, and UNIVAC are registered trademarks of the Sperry Rand Corporation. AccuScan, ESCORT, PAGEWRITER, PIXIE, and UNIS are additional trademarks of the Sperry Rand Corporation.

This document was prepared by Systems Publications using the SPERRY UNIVAC UTS 400 Text Editor. It was printed and distributed by the Customer Information Distribution Center (CIDC), 555 Henderson Rd., King of Prussia, Pa., 19406.



PAGE STATUS SUMMARY

ISSUE: UP-8075 Rev. 2

| Part/Section | Page Number | Update Level | Part/Section | Page Number | Update Level | Part/Section | Page Number | Update Level |
|--------------------|-------------|--------------|--------------|-------------|--------------|--------------|-------------|--------------|
| Cover | | | | | | | | |
| Title Page | | | | | | | | |
| PSS | 1 | | | | | | | |
| Preface | 1, 2 | | | | | | | |
| Contents | 1 thru 9 | | | | | | | |
| PART 1 | | | | | | | | |
| Title Page | | | | | | | | |
| 1 | 1 thru 3 | | | | | | | |
| 2 | 1 thru 10 | | | | | | | |
| 3 | 1 thru 11 | | | | | | | |
| PART 2 | | | | | | | | |
| Title Page | | | | | | | | |
| 4 | 1 thru 37 | | | | | | | |
| 5 | 1 thru 18 | | | | | | | |
| 6 | 1 thru 63 | | | | | | | |
| PART 3 | | | | | | | | |
| Title Page | | | | | | | | |
| 7 | 1 thru 18 | | | | | | | |
| PART 4 | | | | | | | | |
| Title Page | | | | | | | | |
| 8 | 1 thru 60 | | | | | | | |
| 9 | 1 thru 52 | | | | | | | |
| 10 | 1 thru 22 | | | | | | | |
| 11 | 1 thru 9 | | | | | | | |
| Index | 1 thru 15 | | | | | | | |
| User Comment Sheet | | | | | | | | |
| | | | | | | | | |

All the technical changes are denoted by an arrow (→) in the margin. A downward pointing arrow (↓) next to a line indicates that technical changes begin at this line and continue until an upward pointing arrow (↑) is found. A horizontal arrow (→) pointing to a line indicates a technical change in only that line. A horizontal arrow located between two consecutive lines indicates technical changes in both lines or deletions.



Preface

This manual is one of a series designed to instruct and guide the programmer in the use of the SPERRY UNIVAC Operating System/3 (OS/3). This manual specifically describes the OS/3 supervisor and its effective use. Its intended audience is the novice programmer with a basic knowledge of data processing, but with limited programming experience, and the programmer whose experience is limited to other than SPERRY UNIVAC systems.

Prerequisite to the use of this manual is a general knowledge of the OS/3 assembler, job control, and data management.

Two other manuals are available that cover the supervisor; one is an introductory manual and the other is a programmer reference manual (PRM). The introductory manual briefly describes the supervisor and its facilities. The PRM provides the characteristics of OS/3 supervisor in skeletal form and is intended as a quick-reference document for the programmer experienced in the use of the supervisor.

This user guide is subdivided into the following parts:

- PART 1. INTRODUCTION

Introduces the supervisor in terms of what it is, what it does, how it is structured, and how it is used. This part also states the general conventions for writing macro instruction statements which request services of the supervisor.

- PART 2. PHYSICAL INPUT/OUTPUT CONTROL

Describes the macro instructions and techniques by which you may write your own routines to control input/output devices and disk storage space, and to process sequential as well as random access files on disk.

- PART 3. MULTITASKING

Describes the macro instructions and techniques by which your program may execute in a multitask and multijob environment.

■ PART 4. SUPERVISOR SERVICES

Describes the macro instructions and techniques by which you may request other services of the supervisor such as program loading, job and task termination, storage displays, etc.

Each of the foregoing parts consists of one or more sections that cover the different aspects of the subject matter covered in each part.

Contents

PAGE STATUS SUMMARY

PREFACE

CONTENTS

PART 1. INTRODUCTION

1. CONCEPT AND ORGANIZATION

| | | |
|--------|--|-----|
| 1.1. | GENERAL | 1-1 |
| 1.2. | FEATURES | 1-2 |
| 1.2.1. | Modularity | 1-2 |
| 1.2.2. | Minimum Main Storage Requirements | 1-2 |
| 1.2.3. | Multijobbing and Multitasking Capability | 1-3 |
| 1.2.4. | Minimum Operator Intervention | 1-3 |

2. SUPERVISOR INTERFACES

| | | |
|-----------|--|-----|
| 2.1. | INTERRUPT HANDLING | 2-1 |
| 2.2. | MODULAR FUNCTIONS | 2-2 |
| 2.2.1. | Task Control | 2-2 |
| 2.2.2. | Physical Input/Output Control | 2-2 |
| 2.2.2.1. | Execute Channel Program Processor Module | 2-2 |
| 2.2.2.2. | PUB Control Module | 2-3 |
| 2.2.2.3. | Queue Control Module | 2-3 |
| 2.2.2.4. | Address Adjustment Module | 2-4 |
| 2.2.2.5. | Channel Scheduler Modules | 2-4 |
| 2.2.2.6. | Interrupt Module | 2-4 |
| 2.2.2.7. | IOST Processor Module | 2-4 |
| 2.2.2.8. | Channel Interrupt Processor Modules | 2-5 |
| 2.2.2.9. | Error Control Module | 2-5 |
| 2.2.2.10. | Error Editing Root Overlay | 2-5 |
| 2.2.2.11. | Device Sense Analyzer Overlay | 2-5 |
| 2.2.2.12. | Error Reply Overlay | 2-5 |

| | | |
|----------|---|------|
| 2.2.3. | Transient Management | 2-5 |
| 2.2.4. | Console Management | 2-6 |
| 2.2.5. | Resource Allocation | 2-6 |
| 2.2.6. | Timer and Day Clock Services | 2-6 |
| 2.2.7. | Program and Machine Error Control | 2-7 |
| 2.2.8. | Spooling Operations | 2-7 |
| 2.2.9. | Diagnostic and Debugging Aids | 2-7 |
| 2.2.9.1. | Monitor and Trace | 2-7 |
| 2.2.9.2. | Snapshot Display of Main Storage | 2-8 |
| 2.2.9.3. | Main Storage Dumps | 2-8 |
| 2.2.9.4. | Standard System Error Message Interface | 2-8 |
| 2.2.10. | Automatic Volume Recognition | 2-8 |
| 2.2.11. | Main Storage Consolidation | 2-9 |
| 2.2.12. | Rollout/Rollin | 2-9 |
| 2.2.13. | Cochanneling | 2-10 |
| 2.2.14. | Disk Seek Separation | 2-10 |
| 2.2.15. | Error Logging | 2-10 |

3. MACRO INSTRUCTION CONVENTIONS

| | | |
|--------|---|-----|
| 3.1. | GENERAL | 3-1 |
| 3.2. | FORMAT ILLUSTRATION AND STATEMENT CONVENTIONS | 3-1 |
| 3.3. | USE OF THE ASSEMBLER CODING FORM | 3-5 |
| 3.3.1. | Label Field | 3-6 |
| 3.3.2. | Operation Field | 3-7 |
| 3.3.3. | Operand Field | 3-7 |
| 3.3.4. | Comments Field | 3-7 |
| 3.3.5. | Continuation Column | 3-7 |
| 3.3.6. | Sequence Field | 3-8 |
| 3.4. | MACRO INSTRUCTIONS | 3-8 |
| 3.4.1. | Declarative Macro Instructions | 3-8 |
| 3.4.2. | Imperative Macro Instructions | 3-8 |
| 3.4.3. | Summary of Supervisor Macro Instructions | 3-8 |

PART 2. PHYSICAL INPUT/OUTPUT CONTROL

4. PHYSICAL INPUT/OUTPUT CONTROL SYSTEM

| | | |
|--------|--|------|
| 4.1. | GENERAL | 4-1 |
| 4.2. | PHYSICAL I/O CONTROL | 4-2 |
| 4.2.1. | General | 4-2 |
| 4.2.2. | General I/O Usage Requirements | 4-4 |
| 4.2.3. | Generate Buffer Control Word (BCW) | 4-5 |
| 4.2.4. | Generate Channel Command Word (CCW) | 4-15 |
| 4.2.5. | Generate Command Control Block (CCB) | 4-18 |
| 4.2.6. | Generate Physical Input/Output Control Block (PIOCB) | 4-24 |
| 4.2.7. | Read File Control Block (RDFCB) | 4-26 |
| 4.2.8. | Execute Channel Program (EXCP) | 4-28 |
| 4.2.9. | Swap I/O (SWAP) | 4-29 |

| | | | |
|----------|---|-----------|------|
| 4.3. | INPUT/OUTPUT SYNCHRONIZATION | | 4-30 |
| 4.3.1. | Wait for I/O Completion | (WAIT) | 4-31 |
| 4.3.2. | Multiple I/O Wait | (WAITM) | 4-32 |
| 4.4. | BLOCK NUMBERED TAPE FILES | | 4-33 |
| 4.4.1. | Block Number Field | | 4-33 |
| 4.4.2. | Tape Restrictions | | 4-35 |
| 4.4.3. | Input/Output Buffer | | 4-35 |
| 4.4.4. | Processing | | 4-35 |
| 4.4.5. | Physical IOCS Requirements and Options | | 4-36 |
| | | | |
| 5. | DISK SPACE MANAGEMENT | | |
| 5.1. | GENERAL | | 5-1 |
| 5.2. | DISK SPACE MANAGEMENT ROUTINES | | 5-2 |
| 5.2.1. | Allocate Routine | | 5-2 |
| 5.2.2. | Extend Routine | | 5-3 |
| 5.2.3. | Scratch Routine | | 5-3 |
| 5.2.3.1. | Scratch Entire File | | 5-4 |
| 5.2.3.2. | Scratch by Prefix | | 5-4 |
| 5.2.3.3. | Scratch All by Date | | 5-4 |
| 5.2.4. | Rename Routine | | 5-4 |
| 5.2.5. | Obtain Routine | | 5-4 |
| 5.3. | DISK MACRO INSTRUCTIONS | | 5-5 |
| 5.3.1. | Assign Space to a New Disk File | (ALLOC) | 5-5 |
| 5.3.2. | Assign Additional Space to an Existing Disk File | (EXTEND) | 5-7 |
| 5.3.3. | Scratch a Disk File | (SCRATCH) | 5-9 |
| 5.3.4. | Rename a Disk File | (RENAME) | 5-10 |
| 5.3.5. | Access VTOC User Block | (OBTAIN) | 5-12 |
| 5.4. | DISKETTE SPACE MANAGEMENT ROUTINES | | 5-14 |
| 5.5. | DISKETTE MACRO INSTRUCTIONS | | 5-14 |
| 5.5.1. | Assign Space to a New Diskette File | (ALLOC) | 5-14 |
| 5.5.2. | Scratch a Diskette File | (SCRATCH) | 5-16 |
| 5.5.3. | Obtain Diskette Label Information | (OBTAIN) | 5-17 |
| 5.6. | SPACE MANAGEMENT ERROR CODES | | 5-18 |
| | | | |
| 6. | SYSTEM ACCESS TECHNIQUE | | |
| 6.1. | GENERAL | | 6-1 |
| 6.2. | DISK SAT FILE ORGANIZATION AND ADDRESSING METHODS | | 6-1 |
| 6.2.1. | PCA Table Entries Used in Addressing | | 6-1 |
| 6.2.2. | Block Addressing by Key | | 6-3 |
| 6.2.3. | Block Addressing by Relative Block Number | | 6-3 |
| 6.2.4. | Disk Space Control | | 6-4 |
| 6.2.5. | Record Interlace | | 6-5 |
| 6.2.5.1. | Interlace Operation | | 6-6 |
| 6.2.5.2. | Lace Factor Calculation | | 6-8 |
| 6.2.6. | Accessing Multiple Blocks | | 6-8 |

| | | | |
|-----------|--|---------------|------|
| 6.3. | DISK SAT FILE INTERFACE | | 6-10 |
| 6.3.1. | Define a New File | (DTFPF) | 6-10 |
| 6.3.1.1. | Filelocks | | 6-12 |
| 6.3.1.2. | Shared Filelock Capability | | 6-13 |
| 6.3.2. | Defining a Partition | (PCA) | 6-14 |
| 6.3.3. | Processing Partitioned SAT Files | | 6-17 |
| 6.3.3.1. | Processing Blocks by Key | | 6-18 |
| 6.3.3.2. | Processing by Relative Block Number | | 6-19 |
| 6.4. | CONTROLLING YOUR DISK FILE PROCESSING | | 6-19 |
| 6.4.1. | Open a Disk File | (OPEN) | 6-19 |
| 6.4.2. | Retrieve Next Logical Block | (GET) | 6-20 |
| 6.4.3. | Output a Logical Block | (PUT) | 6-21 |
| 6.4.4. | Wait for Block Transfer | (WAITF) | 6-22 |
| 6.4.5. | Read by Key Equal/Read by Key Equal or Higher | (READE/READH) | 6-23 |
| 6.4.6. | Access a Physical Block | (SEEK) | 6-24 |
| 6.4.7. | Close a Disk File | (CLOSE) | 6-24 |
| 6.5. | SAT FOR TAPE FILES | | 6-25 |
| 6.6. | SYSTEM STANDARD TAPE LABELS | | 6-26 |
| 6.6.1. | Volume Label Group | | 6-27 |
| 6.6.2. | File Header Label Group | | 6-29 |
| 6.6.2.1. | First File Header Label | (HDR1) | 6-29 |
| 6.6.2.2. | Second File Header Label | (HDR2) | 6-31 |
| 6.6.3. | File Trailer Label Group | | 6-33 |
| 6.7. | TAPE VOLUME AND FILE ORGANIZATION | | 6-37 |
| 6.7.1. | Standard Tape Volume Organization | | 6-38 |
| 6.7.2. | Nonstandard Tape Volume Organization | | 6-42 |
| 6.7.3. | Unlabeled Tape Volume Organization | | 6-44 |
| 6.8. | TAPE SAT FILE INTERFACE | | 6-45 |
| 6.8.1. | Define a Magnetic Tape File | (SAT) | 6-45 |
| 6.8.2. | Define a Tape Control Appendage | (TCA) | 6-47 |
| 6.9. | CONTROLLING YOUR TAPE FILE PROCESSING | | 6-51 |
| 6.9.1. | Open a Tape File | (OPEN) | 6-51 |
| 6.9.2. | Get Next Logical Block | (GET) | 6-52 |
| 6.9.3. | Output Next Logical Block | (PUT) | 6-53 |
| 6.9.4. | Wait for Block Transfer | (WAITF) | 6-54 |
| 6.9.5. | Control Tape Unit Functions | (CNTRL) | 6-54 |
| 6.9.6. | Close a Tape File | (CLOSE) | 6-55 |
| 6.10. | BLOCK NUMBER PROCESSING | | 6-56 |
| 6.10.1. | Facilities Required for Block Number Processing | | 6-57 |
| 6.10.2. | Specifications for Block Number Processing | | 6-57 |
| 6.10.2.1. | Initialized Processing | | 6-58 |
| 6.10.2.2. | Noninitialized Processing | | 6-58 |

PART 3. MULTITASKING**7. MULTITASKING**

| | | | |
|--------|--------------------------------------|--|-----|
| 7.1. | GENERAL | | 7-1 |
| 7.1.1. | Multijobbing and Multitasking | | 7-1 |

| | | | |
|----------|---|----------|------|
| 7.1.1.1. | Primary Task | | 7-2 |
| 7.1.1.2. | Subtask | | 7-2 |
| 7.2. | TASK MANAGEMENT | | 7-2 |
| 7.2.1. | General | | 7-2 |
| 7.2.2. | Task Creation | | 7-3 |
| 7.2.3. | Task Priority | | 7-4 |
| 7.2.4. | Task Termination | | 7-4 |
| 7.2.5. | Queue Driven Task | | 7-4 |
| 7.2.6. | Hierarchical Structure | | 7-4 |
| 7.3. | TASK MANAGEMENT MACRO INSTRUCTIONS | | 7-5 |
| 7.3.1. | Generate an Event Control Block | (ECB) | 7-6 |
| 7.3.2. | Create an Additional Task | (ATTACH) | 7-9 |
| 7.3.3. | Terminate a Task | (DETACH) | 7-10 |
| 7.3.4. | Yield Until Task Completion | (TYIELD) | 7-11 |
| 7.3.5. | Reactivate a Task | (AWAKE) | 7-12 |
| 7.3.6. | Change a Priority | (CHAP) | 7-13 |
| 7.4. | TASK SYNCHRONIZATION | | 7-14 |
| 7.4.1. | General | | 7-14 |
| 7.4.2. | Wait for Task Completion | (WAIT) | 7-15 |
| 7.4.3. | Multiple Task Wait | (WAITM) | 7-16 |
| 7.4.4. | Activate the Waiting Task | (POST) | 7-17 |

PART 4. SUPERVISOR SERVICES

8. PROGRAM MANAGEMENT

| | | | |
|----------|-----------------------------------|----------|------|
| 8.1. | GENERAL | | 8-1 |
| 8.1.1. | Program Initiation and Loading | | 8-1 |
| 8.2. | PROGRAM LOADER | | 8-2 |
| 8.2.1. | Block Loader | | 8-2 |
| 8.2.2. | Relocation | | 8-3 |
| 8.2.3. | Library Search Order | | 8-4 |
| 8.2.4. | Read Pointer for Repetitive Loads | | 8-4 |
| 8.2.5. | Loader Error Processing | | 8-5 |
| 8.2.6. | Load a Program Phase | (LOAD) | 8-5 |
| 8.2.7. | Load a Program Phase and Relocate | (LOADR) | 8-7 |
| 8.2.8. | Locate a Program Phase Header | (LOADI) | 8-9 |
| 8.2.8.1. | Program Phase Header | | 8-10 |
| 8.2.9. | Load a Program Phase and Branch | (FETCH) | 8-11 |
| 8.3. | PROGRAM TERMINATION | | 8-12 |
| 8.3.1. | Normal Termination | | 8-13 |
| 8.3.2. | Abnormal Termination | | 8-13 |
| 8.3.3. | Printout | | 8-13 |
| 8.3.4. | End-of-Job Step | (EOJ) | 8-13 |
| 8.3.5. | Cancel a Job | (CANCEL) | 8-14 |
| 8.4. | TIMER SERVICES | | 8-15 |
| 8.4.1. | Date and Time Facilities | | 8-16 |
| 8.4.1.1. | Current Date | | 8-16 |
| 8.4.1.2. | Time of Day | | 8-17 |
| 8.4.1.3. | Get Current Date and Time | (GETIME) | 8-17 |

| | | | |
|----------|--|--------------|------|
| 8.4.2. | Timer Interrupt Facilities | | 8-20 |
| 8.4.2.1. | Set Timer Interrupt | (SETIME) | 8-21 |
| 8.4.2.2. | Continue Processing Until Interrupt | | 8-22 |
| 8.4.2.3. | Wait for Interrupt | | 8-24 |
| 8.4.2.4. | Cancel a Previous Timer Interrupt Request | | 8-24 |
| 8.5. | PROGRAM LINKAGE | | 8-25 |
| 8.5.1. | Linkage Register Conventions | | 8-25 |
| 8.5.2. | Linkage Procedure | | 8-26 |
| 8.5.3. | Register Save Area | | 8-27 |
| 8.5.4. | Call a Program | (CALL/VCALL) | 8-28 |
| 8.5.5. | Generate an Argument List | (ARGLST) | 8-30 |
| 8.5.6. | Save Register Contents | (SAVE) | 8-30 |
| 8.5.7. | Restore Registers and Return | (RETURN) | 8-32 |
| 8.6. | ISLAND CODE LINKAGE | | 8-34 |
| 8.6.1. | Attaching Island Code to a Task | (STXIT) | 8-35 |
| 8.6.1.1. | Attaching Program Check, Abnormal Termination, and Interval Timer Island Code | | 8-35 |
| 8.6.1.2. | Attaching Operator Communication Island Code | | 8-36 |
| 8.6.2. | Detaching Island Code From a Task | (STXIT) | 8-38 |
| 8.6.3. | Island Code Entrance | | 8-39 |
| 8.6.4. | Island Code Exit | (EXIT) | 8-39 |
| 8.6.4.1. | Exiting From Program Check Interval Timer, and Operator Communication Island Code | | 8-39 |
| 8.6.4.2. | Exiting From Abnormal Termination Island Code | | 8-40 |
| 8.6.5. | Program Check | | 8-40 |
| 8.6.6. | Abnormal Termination | | 8-43 |
| 8.6.7. | Interval Timer | | 8-45 |
| 8.6.8. | Operator Communication | | 8-46 |
| 8.6.9. | Use of Island Code With Multitasking | | 8-49 |
| 8.6.9.1. | Program Check and Interval Timer With Multitasking | | 8-49 |
| 8.6.9.2. | Abnormal Termination With Multitasking | | 8-51 |
| 8.6.9.3. | Operator Communication With Multitasking | | 8-51 |
| 8.7. | SYSTEM INFORMATION CONTROL | | 8-51 |
| 8.7.1. | Get Data From Communication Region | (GETCOM) | 8-52 |
| 8.7.2. | Put Data Into Communication Region | (PUTCOM) | 8-53 |
| 8.7.3. | Get Data From System Control Tables | (GETINF) | 8-53 |
| 8.8. | CONTROL STREAM READER | | 8-55 |
| 8.8.1. | Embedded Data | | 8-56 |
| 8.8.2. | Reading Embedded Data | | 8-56 |
| 8.8.3. | Get File From Control Stream | (GETCS) | 8-57 |
| 8.8.4. | Rereading Embedded Data | | 8-59 |
| 8.8.5. | Reset Control Stream Reader | (SETCS) | 8-59 |
| 8.8.6. | Minimizing Disk Accesses | | 8-60 |
| 9. | DIAGNOSTIC AND DEBUGGING AIDS | | |
| 9.1. | STORAGE DISPLAYS | | 9-1 |
| 9.1.1. | Snapshot Dumps | (SNAP/SNAPF) | 9-1 |
| 9.1.2. | Normal Termination Dumps | (DUMP) | 9-5 |
| 9.1.3. | Abnormal Termination | | 9-10 |

| | | | |
|------------|--|---------------------------|------|
| 9.2. | CHECKPOINT AND RESTART CAPABILITY | | 9—10 |
| 9.2.1. | How to Generate Checkpoint Records | (CHKPT) | 9—12 |
| 9.2.2. | Using Magnetic Tape as the Checkpoint File | | 9—14 |
| 9.2.3. | Using a SAT Disk as a Checkpoint File | | 9—15 |
| 9.2.3.1. | Estimate Spare Requirements for a Disk Checkpoint File | | 9—16 |
| 9.2.3.2. | Define, Open, and Close a Disk Checkpoint File | (DDCPF, DCPOP, DCPCLS) | 9—17 |
| 9.2.4. | Processing PIOCS Files | (DCFLT) | 9—18 |
| 9.3. | MONITOR AND TRACE CAPABILITY | | 9—22 |
| 9.3.1. | How to Call the Monitor Routine | | 9—23 |
| 9.3.1.1. | Monitoring From the Beginning of the Job | | 9—23 |
| 9.3.1.2. | Monitoring After Execution Begins | | 9—25 |
| 9.3.2. | Monitor Input Format | | 9—27 |
| 9.3.3. | Defining What You Want to Monitor | | 9—29 |
| 9.3.4. | Specifying Options | | 9—31 |
| 9.3.4.1. | Storage Reference Option (S) | | 9—32 |
| 9.3.4.1.1. | Program Relative Address (PR) | | 9—32 |
| 9.3.4.1.2. | Base/Displacement Address (B/D) | | 9—34 |
| 9.3.4.1.3. | Absolute Address (ABS) | | 9—34 |
| 9.3.4.2. | Instruction Location Option (A) | | 9—35 |
| 9.3.4.3. | Instruction Sequence Option (I) | | 9—36 |
| 9.3.4.4. | Register Change Option (R) | | 9—37 |
| 9.3.4.5. | No Option Specified? You Get a Default | | 9—37 |
| 9.3.5. | Specifying Actions | | 9—38 |
| 9.3.5.1. | Display Actions | | 9—38 |
| 9.3.5.1.1. | Register Display (DΔR) | | 9—39 |
| 9.3.5.1.2. | Storage Display (DΔS) | | 9—40 |
| 9.3.5.1.3. | Default Display | | 9—42 |
| 9.3.5.2. | Halt Action (H) | | 9—43 |
| 9.3.5.3. | Quit Action (Q) | | 9—44 |
| 9.3.6. | Cancel of Monitor | | 9—45 |
| 9.4. | SYSTEM DEBUGGING AIDS | | 9—45 |
| 9.4.1. | Supervisor Debug Option | | 9—46 |
| 9.4.2. | Mini Monitor | | 9—48 |
| 9.4.3. | Console Debug Options | | 9—50 |
| 9.4.4. | Transient Halt Location | | 9—51 |
| 9.4.5. | Symbiont Halt Location | | 9—51 |

10. MESSAGE DISPLAY, LOGGING, AND OPERATOR COMMUNICATION

| | | | |
|-----------|---|----------|-------|
| 10.1. | GENERAL | | 10—1 |
| 10.1.1. | The Canned Message File | | 10—3 |
| 10.1.1.1. | Canned Messages | | 10—3 |
| 10.1.1.2. | Inserting Variable Characters in a Canned Message | | 10—3 |
| 10.1.2. | The System Log | | 10—6 |
| 10.2. | MESSAGE AND LOGGING MACRO INSTRUCTIONS | | 10—6 |
| 10.2.1. | Write to the Log | (WTL) | 10—6 |
| 10.2.2. | Display a Message and Write to the Log | (WTLD) | 10—9 |
| 10.2.3. | Get a Canned Message | (GETMSG) | 10—14 |

| | | | |
|---------|------------------------------------|-------|-------|
| 10.3. | USER-OPERATOR COMMUNICATION | | 10-17 |
| 10.3.1. | General | | 10-17 |
| 10.3.2. | Display a Message to the Operator | (OPR) | 10-18 |

11. OTHER SERVICES

| | | | |
|-----------|--|---------|------|
| 11.1. | SPOOLING | | 11-1 |
| 11.1.1. | General | | 11-1 |
| 11.1.1.1. | Initialization | | 11-1 |
| 11.1.1.2. | Input Reader | | 11-2 |
| 11.1.1.3. | Spooler | | 11-2 |
| 11.1.1.4. | Output Writer | | 11-3 |
| 11.1.1.5. | Special Functions | | 11-4 |
| 11.1.2. | To Use Spooling | | 11-4 |
| 11.1.3. | Create a Breakpoint in a Spool Output File | (BRKPT) | 11-5 |
| 11.2. | JOB ACCOUNTING | | 11-5 |
| 11.2.1. | General | | 11-5 |
| 11.2.2. | Accounting Data | | 11-6 |
| 11.2.2.1. | Job Step Level Data | | 11-6 |
| 11.2.2.2. | Job Level Data | | 11-7 |
| 11.2.3. | Data Printout | | 11-8 |

INDEX

USER COMMENT SHEET

FIGURES

| | | | |
|--------|--|--|------|
| 3-1. | 9000 Series Assembler Coding Form | | 3-6 |
| 4-1 | Relationship of Basic Physical IOCS Macro Instructions | | 4-3 |
| 4-2. | Buffer Control Word (BCW) Format for Integrated Disk Adapter | | 4-7 |
| 4-3. | Buffer Control Word (BCW) Format for Integrated Peripheral Channel | | 4-10 |
| 4-4. | Buffer Control Word (BCW) Format for Multiplexer Channel | | 4-13 |
| 4-5. | Channel Command Word (CCW) Format for Selector Channel | | 4-16 |
| 4-6. | Channel Address Word (CAW) Format | | 4-17 |
| 4-7. | Command Control Block (CCB) Format | | 4-22 |
| 4-8. | Physical I/O Control Block (PIOCB) and File Control Block (FCB) Format | | 4-25 |
| → 4-9. | Tape Block Number Field Format | | 4-34 |
| 6-1. | Partition Control Appendage (PCA) Table Format | | 6-2 |
| 6-2. | Record Formats for Disk Devices | | 6-3 |
| 6-3. | Definition of Interface Variables | | 6-6 |
| 6-4. | Interface Accessing | | 6-7 |
| 6-5. | Define the File (DTF) Table Format | | 6-9 |
| 6-6. | Tape Volume 1 (VOL1) Label Format for an EBCDIC Volume | | 6-28 |
| 6-7. | First File Header Label (HDR1) Format for an EBCDIC Tape Volume | | 6-30 |
| 6-8. | Second File Header Label (HDR2) Format for an EBCDIC Tape Volume | | 6-32 |
| 6-9. | Tape File EOF1 and EOY1 Label Formats for EBCDIC Tapes | | 6-34 |
| 6-10. | Tape File EOF2 and EOY2 Label Formats for EBCDIC Tapes | | 6-36 |
| 6-11. | Reel Organization for EBCDIC Standard Labeled Tape Volumes Containing a Single File | | 6-39 |

| | |
|---|------|
| 6—12. Reel Organization for EBCDIC Standard Labeled Tape Volume: Multifile Volume With End-of-File Condition | 6—40 |
| 6—13. Reel Organization for EBCDIC Standard Labeled Tape Volumes: Multifile Volumes With End-of-Volume Condition | 6—41 |
| 6—14. Reel Organization for EBCDIC Nonstandard Volumes Containing a Single File | 6—42 |
| 6—15. Reel Organization for EBCDIC Nonstandard Multifile Volumes | 6—43 |
| 6—16. Reel Organization for Unlabeled EBCDIC Volumes | 6—44 |
| 6—17. Tape Volume 1 (VOL1) Label Format for an EBCDIC Volume With Block Numbers | 6—59 |
| 6—18. First File Header Label (HDR1) Format for an EBCDIC Tape Volume With Block Numbers | 6—60 |
| 6—19. Second File Header Label (HDR2) Format for an EBCDIC Tape Volume With Block Numbers | 6—61 |
| 6—20. Tape File EOF1 and EOVS1 Label Formats for Block Numbered EBCDIC Files | 6—62 |
| 6—21. Tape File EOF2 and EOVS2 Label Formats for Block Numbered EBCDIC Files | 6—63 |
| 7—1. Event Control Block (ECB) Format | 7—8 |
| 8—1. Example of GETIME Macro Instruction | 8—19 |
| 8—2. Example of SETIME Macro Instruction | 8—23 |
| 8—3. Register Save Area Format | 8—27 |
| 8—4. Example of Program Check Island Code Linkage Using Symbolic Addresses | 8—41 |
| 8—5. Example of Program Check Island Code Linkage Using Register Addresses | 8—42 |
| 8—6. Example of Abnormal Termination Island Code Linkage Using Symbolic Addresses | 8—44 |
| 8—7. Example of Interval Timer Island Code Linkage Using Symbolic Addresses | 8—45 |
| 8—8. Example of Operator Communication Island Code Linkage Using Symbolic Addresses | 8—47 |
| 8—9. Example of Operator Communication Island Code Linkage Using Register Addresses | 8—48 |
| 8—10. Example of Discrete Program Check Island Code for Each Task in a Job Step | 8—49 |
| 8—11. Example of Common Program Check Island Code for All Tasks in a Job Step | 8—50 |
| 9—1. Monitor Input Format | 9—29 |
| 10—1. Canned Message Buffer Formats | 10—4 |
| 10—2. Insertion of Variable Characters in a Canned Message | 10—5 |
| 11—1. Relationship of Spooling Devices and Programs | 11—2 |
| 11—2. Job Accounting Table Format | 11—6 |
| 11—3. Job Accounting Record Printout Format | 11—9 |

TABLES

| | |
|---|------|
| 3—1. Supervisor Macro Instructions | 3—9 |
| 6—1. Tape Volume 1 (VOL1) Label Format, Field Description for an EBCDIC Volume | 6—29 |
| 6—2. First File Header Label (HDR1), Field Description | 6—31 |
| 6—3. Second File Header (HDR2), Field Description | 6—33 |
| 6—4. Tape File EOF1 and EOVS1 Labels, Field Description | 6—35 |
| 6—5. Tape File EOF2 and EOVS2 Labels, Field Description | 6—37 |
| 8—1. Register Save Area | 8—28 |
| 9—1. Checkpoint/Restart Error Codes | 9—13 |
| 9—2. Summary of Actions and Program Information Printed | 9—45 |
| 9—3. Summary of System Debugging Aids | 9—52 |
| 10—1. Summary of Message Macro Instructions | 10—2 |



PART 1. INTRODUCTION

OFFICE OF THE ATTORNEY GENERAL

1. Concept and Organization

1.1. GENERAL

The SPERRY UNIVAC Operating System/3 (OS/3) Supervisor (supervisor) is the component that operates with problem programs (user programs) to provide the central control necessary for optimum and continuous utilization of the system hardware and software. It provides the control, interface, coordination, and allocation of hardware and controls the initiation, loading, executing, and termination of user jobs. The efficient and flexible capabilities provided by the supervisor are particularly useful for small to medium sized disc-oriented computing systems.

Within the context of this manual the following definitions apply:

Job

A total processing application comprising one or more processing steps. Each job is divided into job steps (programs) that are executed serially. With the exception of disk space, resources are allocated on a job basis.

Job Step

The unit of work associated with one processing program. A job step is an executable program consisting of one or more tasks that requires a specific amount of the hardware resources of the system.

Task

A unit of work capable of competing with other tasks for control of the central processor. A task is a logical point of control rather than a physical set of instructions. Each job step has at least one task and may create additional tasks (subtasks) all of which compete independently for processor time.

Multitasking

The concurrent processing of many tasks asynchronously. Multitasking applies to the switching of processor control among two or more tasks on a priority or rotational basis. Job steps with more than one task are capable of using multitasking.

Multijobbing

The concurrent scheduling, loading, and execution of more than one job at a time. This term is not synonymous with multitasking.

1.2. FEATURES

1.2.1. Modularity

The supervisor is designed around control modules, each representing functions or services to be provided. At system generation time, a supervisor program is produced with modules modified and combined to provide the specific combination of capabilities to meet the requirements and restrictions of the particular user installation and applications.

1.2.2. Minimum Main Storage Requirements

The modular design of the supervisor keeps the resident main storage requirement to a minimum. Modules that are frequently used and constitute an integral part of the supervisor are called resident routines because they require permanent residence in main storage. Modules that are not continuously required and are not time critical to normal job execution are called transient routines and are kept on the system resident disk storage. These transient routines are located and loaded from disk into main storage only when needed, and executed as an extension of the requesting program.

The following modules are always part of the resident supervisor (except for timer and day clock services which are optional):

Supervisor Interface

The points at which control is passed to the supervisor by means of the supervisor call (SVC) instruction interrupt.

Task Switcher

The hub of the supervisor, this routine controls allocation of the processor based upon internal priorities.

Transient Management

Schedules, locates, and loads the noncritical transients which perform the nonresident supervisor functions.

Supervisor Overlay Scheduler

Schedules critical supervisor overlays.

Physical Input/Output Control

Controls the dispatching, queueing, and interrupt processing for all I/O devices directly connected to the system.

Timer and Day Clock Services

Provides system clock and timer activities control.

Error Control

Handles unresolved I/O, machine check, and program check interrupts; schedules user island code subroutines or overlay functions to handle errors appropriately.

Other modules may be selected for inclusion within the resident supervisor at system generation. Such modules as clock control will be either resident or not available; whereas, most modules will be either resident or transient, depending upon system generation options.

1.2.3. Multijobbing and Multitasking Capability

The supervisor provides multijobbing and multitasking capability through the submission of job control streams which represent the jobs to be performed. In multijobbing environments, from one to seven user jobs may be executed concurrently, with the jobs consisting of a series of job steps (programs). The job steps are executed in a serial manner within each job. Job steps may have from 1 to 256 tasks capable of executing concurrently with other tasks within the job step or system.

1.2.4. Minimum Operator Intervention

Operator intervention is kept to a minimum. Operating in conjunction with the job control system, the supervisor provides efficient control of the multijobbing environment. Most error situations are handled by the supervisor and by user-supplied error routines, so that an operator usually is not required to initiate error recovery procedures.



2. Supervisor Interfaces

2.1. INTERRUPT HANDLING

The OS/3 Supervisor is informed of an event, either within the supervisor complex or external to it, by an interrupt. Interrupts may be enabled (allowed) or disabled (held pending) to avoid simultaneous interrupts and to service interrupts based on their relative priorities. Upon recognizing an interrupt, the executing task is suspended and program control is transferred to the appropriate interrupt handler. The interrupt handler analyzes the cause of the interrupt and activates the appropriate interrupt servicing routine.

There are six classes of interrupts by which control is returned to the supervisor:

- Supervisor call
- Interval timer
- Input/output
- Program errors
- Hardware errors
- Operator request

Of these six, the supervisor call and operator request interrupts provide the user with an interface to the supervisor and, therefore, the operating system. The rest of the interrupts are handled by the specific routines which are described on the following pages.

Tasks have access to the supervisor via the supervisor call (SVC) instruction which is generated within system macro instructions. These supervisor macro instructions provide the access and generate the parameter list associated with the desired function.

The service interrupt routine (SVC decode) determines what function is being requested and passes control to the appropriate resident module. If the function is nonresident, then control is passed to transient management and the overlay is loaded from the system resident disk.

The operator request interrupt allows the operator to initiate action by the supervisor from the console or to answer a previously asked question. The attention interrupt causes the loading of the operator communication overlay, which allows the operator to enter the command or response.

2.2. MODULAR FUNCTIONS

2.2.1. Task Control

Up to seven user jobs can be activated by the job scheduler for concurrent execution. Job steps consist of one or more tasks which are asynchronously executed based on internal priority.

Each job step has one primary task generated by the system. This task is deleted at the termination of the job step. When it is desirable to establish additional tasks for the program, supervisor requests are provided to attach additional tasks. Facilities to synchronize and detach tasks are also provided.

The allocation of processor time to a task is based on a system switch list which contains information about switching priorities. The number of priorities is a parameter in the supervisor generation (SYSGEN).

2.2.2. Physical Input/Output Control

The physical input/output control system (PIOCS) is structured in function-oriented modules. This allows a user to have the minimum usage of resident main storage needed for a particular configuration. The following is a brief description of the functions performed by each of these modules.

2.2.2.1. Execute Channel Program Processor Module

The execute channel program (EXCP) processor is a primary module of PIOCS together with the hardware interrupt processor. These modules access the remaining functional modules to complete their task.

The EXCP module receives control from the SVC decode routine, validates the request, queues the request, and conditionally executes the request based on channel and device availability.

The modules accessed by the EXCP processor include the relocation module, the physical unit block (PUB) control module, the queue control module, and the various channel scheduler modules.

In the case of certain error conditions, it relinquishes primary control to the hardware interrupt module.

The diagnostic adapter module interface and the error logging module interface is optional.

The standard entrance to the EXCP module is the SVC interrupt routine processing the EXCP imperative macro instruction.

The standard exit of the module is the switcher.

The module exits to an error control routine in the event of a failure in the validation checks performed by the module.

The module accesses various special purpose modules in performing its function. These modules include:

- Physical unit block (PUB) control module

This module is accessed when the request is first submitted to validate the existence of the device being called. It is also accessed after retrieval of a CCB from the I/O queue. This is to validate that the contents of the CCB have not been inadvertently altered in the interim period between the time it is first submitted and the time it is retrieved from the I/O queue for execution.

- Queue control module

This module is accessed by the EXCP module to place the request in the I/O queue; to retrieve a request from the I/O queue; and, in the case of program errors detected by the channel schedulers, to delete a request from the I/O queue.

- Channel scheduler modules

There is a unique module for each channel type which performs the functions necessary to prepare a request for the start I/O operation. This includes the set-up of the low storage buffer control words (BCW) and the command address word (CAW).

2.2.2.2. PUB Control Module

The PUB control module has a number of primary functions:

- Verification of a PUB associated with a CCB
- Location of the device associated with a hardware interrupt
- Location of the device associated with the execution of an REXCP imperative macro (answered operator communication)
- Location of a device with an interrupt held in abeyance in a communication environment

2.2.2.3. Queue Control Module

The queue control module is responsible for the maintenance of the queue list module. This includes adding to the I/O queues, searching and retrieving from the I/O queues, and deletion from the I/O queues. A queue head address is maintained for the system and each job for each I/O path (i.e., integrated disk adapter, each selector channel, each integrated peripheral subchannel, and each multiplexer subchannel). Requests are queued first-in, first-out by priority within the queue for the job. Retrieval from a given job queue is in the sequence in which the CCBs are queued.

The selection of the CCB to be executed among the queue heads is accomplished by scanning the queue heads from the last executed queue head in a circular direction looking for the highest priority CCB to be executed. This scan takes the first encountered job if two CCBs of equal priority are detected during the scan.

At system generation time, you may select the option to scan the queue heads of random access devices for the closest seek address rather than lowest priority. This scan would also start with system function and resolve conflicts by selecting the first CCB encountered.

2.2.2.4. Address Adjustment Module

The CCW data address adjustment module converts all addresses in a command chain to absolute form prior to issuance of the start I/O (SIO) command to the hardware. The addresses are converted to relative form prior to returning the command chain to the caller. The module is accessed by the selector scheduler module to create absolute addresses, and by the selector interrupt module to create relative addresses.

2.2.2.5. Channel Scheduler Modules

The various channel scheduler modules prepare a command for execution on a particular channel. This function includes the validation of the command as it applies to a unique channel, and the preparation of the channel's fixed low order storage locations. Additionally, they present a transparent interface to systems users. The integrated channel format is converted to multiplexer or selector channel format.

2.2.2.6. Interrupt Module

The interrupt modules perform processing common to the handling of all interrupts. These functions include:

- accessing the I/O status tables (IOST) for an interrupt to be processed;
- accessing the PUB control module for the device associated with the interrupt;
- transferring control to the particular channel interrupt processing for further processing of solicited interrupts.;
- alerting the console manager for console unsolicited interrupts;
- alerting automatic volume recognition (AVR) for other unsolicited interrupts;
- processing of answered console error communications for I/O; and
- final processing of a CCB, which includes posting the CCB and transferring control to the EXCP processing module for further commands.

2.2.2.7. IOST Processor Module

The module performs the maintenance of the I/O status tables. This involves processing an interrupt, setting the verification indicator, updating the soft status tables pointer, and presenting an interrupt to be processed to the interrupt module.

2.2.2.8. Channel Interrupt Processor Modules

The various channel interrupt processor modules perform functions unique to particular channels. They also access the common error control module on the occurrence of an error condition.

2.2.2.9. Error Control Module

This module performs error processing functions common to all error conditions. This includes processing an error action table of a particular channel and alerting the error editing transient when an operator communication becomes necessary.

2.2.2.10. Error Editing Root Overlay

This overlay performs preliminary processing common to all I/O error messages to the operator. The device and channel address, the device status, the channel status, and the sense information are prepared in a canned message format. The reply options are validated and prepared. The appropriate device or channel error mnemonic sense analysis is then called as an overlay.

2.2.2.11. Device Sense Analyzer Overlay

These various critical overlays convert the sense information of a particular error into an English language message to the operator. The operator communication critical overlay is then called to output the message to the operator.

2.2.2.12. Error Reply Overlay

This overlay processes an answered I/O error communication to the operator. It validates the reply and prepares resident control to perform final processing of the message reply.

2.2.3. Transient Management

Transient management consists of two routines, the transient scheduler and the transient loader. The transient scheduler receives control and executes as a supervisor critical function. It allocates a transient area and schedules the transient loader to receive control as the task associated with this transient area. The transient loader computes the disk address based upon the transient identifier and initiates a read of the transient. Upon normal completion of the read, control is passed to the transient.

The transients will either request the overlay of themselves with subsequent phases or release the area when finished processing. The transient loader performs the read of the overlay or yields control to the task switcher.

Transient management is designed to locate a transient and load in a very efficient manner, requiring only one access to the disk. In addition, it supports serially reusable and private copy transients in order to gain additional efficiencies by reducing disk I/O accesses.

2.2.4. Console Management

Console management provides for the displaying of messages on the CRT screen with responses and commands coming from the operator. The screen images are rolled upward with new display lines or operator input appearing on the bottom of the screen. These routines selectively delete messages not requiring responses from the top of the screen.

Console management is nonresident and is loaded as an overlay when requested. These requests come either as an SVC instruction from a program or as an attention interrupt from the operator.

2.2.5. Resource Allocation

Resources are allocated by the supervisor or job control on a job basis. Main storage and devices are allocated at job initiation for the job's duration. Normally, disk storage should be allocated at job initiation. However, the capability is provided to allocate or extend permanent and temporary files on a dynamic basis during execution of a job step.

Disk space management routines provide an efficient and completely automatic space accounting and maintenance feature, which relieves you of the responsibility of knowing the precise contents of disk volumes. The routines also permit resolution of competing demands for allocation and establishment of standard interfaces.

Disk space management consists of service routine sets that allocate space to files on disk volumes. This is accomplished by maintaining the volume table of contents (VTOC), through standard procedures, for all files: system, temporary, and permanent.

The routines maintain the VTOC by creating control records for new files and deleting control records for files removed from the volume. When a file is created, unused space is found for it by searching the appropriate records in the VTOC, allocating the space as extents of the file, and removing it from free space. When a file is deleted, the control record for the file is removed from the VTOC; the extents previously assigned to the file are then available for allocation.

Job control requests a main storage job region at job initiation. This region is capable of satisfying the main storage requirements of any job step within the job. Job control determines the main storage necessary for the largest job step in addition to that needed by the operating system for this job. This amount of main storage is requested from the supervisor if all device requirements are satisfied.

Devices are allocated to job steps and particular volumes as the job steps are initiated. The supervisor is not involved in device allocation to jobs.

2.2.6. Timer and Day Clock Services

The system hardware contains a high resolution timer. An interface is provided to allow a task to request an interrupt after any time period greater than 1 millisecond. The calling task may specify the wait interval in milliseconds or seconds.

The time of day is provided by a simulated day clock. In addition to providing the time to programs upon request, this time is used by the supervisor for time stamping of log messages and job accounting entries.

2.2.7. Program and Machine Error Control

Any error which causes a program interrupt is examined to determine the type of interrupt with the appropriate action being taken.

An interface is provided for processing error information by means of user-supplied island code. Island code is a closed subroutine, having the entry point defined to the supervisor by various action macro instructions, and is given control upon the occurrence of certain contingencies. Standard actions are initiated in the absence of user code. If the unrecovered error is in the system, the system will terminate that task which initiated the action resulting in the error.

If any requester of a supervisor function provides a set of parameters which are inconsistent or invalid, the requester is abnormally terminated.

2.2.8. Spooling Operations

The supervisor uses a spooling technique which consists of a set of routines that buffer data files for low speed input and output devices to a direct access storage device. There are three types of routines used for spooling operations: job control stream and card disk readers, supervisor printer/punch spooling cooperative, and output printer/punch writers. In addition, you can utilize the data conversion utilities for converting slow speed media to high speed devices or reverse.

Output writers are provided for online devices, as well as those used in a remote batch environment. This allows user jobs to be unaware of whether they are operating with real devices or spooled files.

Input readers are provided for local subsystems which are normally used as batch mode input devices. The user job is required to discern whether data files were submitted as control stream embedded data, or spooled input, or input from unit record device.

2.2.9. Diagnostic and Debugging Aids

Diagnostic and debugging aids provided in the supervisor include monitor mode, snapshot display of main storage, main storage dumps, standard system error message interface, uniform error responses to user programs and program checkpoint restart. Descriptions of these aids are provided in the following paragraphs.

2.2.9.1. Monitor and Trace

The monitor routine enables you to trace the execution of a program by a hardware monitor interrupt so that errors can be located and corrected. You can monitor an entire task or part of a task. In your input to the monitor routine, you can specify actions to be performed at specific points in the program. The monitor routine interrupts each instruction before it is executed and tests for the conditions specified in your monitor input. For each condition, you can request a monitor printout of current program information (PSW contents, next instruction to execute, etc), and continue program execution under monitor control, suspend program execution, or continue program execution without monitor intervention.

2.2.9.2. Snapshot Display of Main Storage

The capability is provided for requesting a partial storage printout at given points in a program by means of a SNAP or SNAPF macro instruction within the program itself. It is also possible to enable or disable these dumps at run time by means of job control. This enables a program to be tested without recompilation to include and disable SNAP or SNAPF requests.

2.2.9.3. Main Storage Dumps

A main storage dump may be provided for programs under the following conditions:

- Abnormal termination dump for user job provides a main storage dump of the region in hexadecimal plus a formatted display of error codes, job-oriented tables, and supervisor information to assist the user in debugging.
- Program or operator request dump provides an orderly capability for the operator or any program to request a main storage dump in the same format as the normal termination dump.
- System failure dump. This is a program intended for use when, for some unexplained reason, the operating system performs abnormally.

2.2.9.4. Standard System Error Message Interface

An error message service routine provides complete and specific error messages without requiring each system module to contain alphanumeric error information. This routine locates the message in a disk file and transfers control to the system console handler for message display or system logging.

2.2.10. Automatic Volume Recognition

Automatic volume recognition allows the console operator to premount magnetic tapes and disk packs before the devices are required for a job step. This reduces time lost due to job step setup and console responses. The automatic volume recognition function is performed during supervisor initialization and as a result of an attention interrupt being received from an online I/O device. This attention interrupt is caused by physically activating the device online, or, in the case of a device that does not have an attention interrupt capability, by the operator issuing an AVR console command.

Using the physical unit block (PUB) for the devices, automatic volume recognition checks to see if the required tape and disk volumes are already mounted. In addition, it performs special processing to handle unique characteristics of various devices. For example, when required at supervisor initialization, it distinguishes between an 8418 disk pack with high density and an 8418 disk pack with low density or an 8416; it performs special interrupt processing for the 8415 disk; it identifies an 0776 printer configured as an 0770 printer. It then marks the device type in the PUB for that device. It also distinguishes between block numbered and unnumbered tapes. If a tape is not at loadpoint, it rewinds the tape so that it can read the label and the volume serial number.

The automatic volume recognition function displays console messages to the operator to indicate such conditions as a disk or tape not prepped, an I/O error, or a duplicate volume serial number.

A system generation option incorporates a retry on the attention interrupts feature in the AVR function. This permits automatic retry of a recoverable error when an attention interrupt is received on a printer, card reader, or card punch that has an unanswered physical IOCS error message. The operator can initiate the recovery retry at the device by placing it online, instead of having to return to the console to respond to the error message.

2.2.11. Main Storage Consolidation

Main storage consolidation is a system generation option that repositions jobs and reallocates space in main storage so that enough contiguous space can be made available when needed to hold the next job to be initiated. This reduces fragmentation of main storage and permits a job to be run that requires more contiguous space than is currently available without consolidation.

When a job or a symbiont terminates, the next job to be run is evaluated to determine whether there is enough space available or whether main storage consolidation is necessary and which jobs must be moved. If this job is scheduled and consolidation is required, the jobs are moved down one by one, starting with those farthest from the supervisor. Each job to be moved is brought to an idle state, then moved down. Addresses are adjusted and the job is reactivated. When all these jobs have been moved, the next scheduled job is read in and initiated.

Main storage consolidation does not move symbionts because they do not have an associated relocation register. Nor does main storage consolidation move jobs with open interfaces to the integrated communications access method (ICAM), because these jobs may be reading or writing directly into or out of user main storage. This restriction is minimized if ICAM is loaded first, then ICAM user jobs next, in order to retain the maximum continuous main storage region for further allocation.

2.2.12. Rollout/Rollin

The rollout/rollin function is a system generation option that temporarily transfers jobs from main storage to disk to make room for a job with a preemptive scheduling priority. Jobs currently in main storage are suspended and written to the job's run library. The preemptive job is then read into main storage and initiated. As enough space becomes available, the rolled-out jobs are read back into main storage and allowed to continue processing.

When a job or a symbiont terminates and there is a preemptive job in the job queue, the preemptive job is evaluated to determine whether there is enough existing main storage available, or whether main storage consolidation or rollout is necessary to make space available. If the job is scheduled and rollout is required, the rollout function brings each job marked for rollout to an idle state, delinks the TCBs from the switch list, and writes the job's image from the job region to disk. These rolled-out jobs have asterisks appended to their names on the top line of the display on the system console. If the needed I/O devices are available, the preemptive job is read into the freed main storage and initiated.

As space becomes available and if there are no other preemptive jobs, the job scheduler tries to bring in the rolled-out jobs, one by one. The job slots and I/O devices remain in effect from the time the jobs were rolled out. The job scheduler ignores any jobs on the high- or normal-priority job queues until all of the rolled-out jobs have been rolled back in and reactivated.

2.2.13. Cochannelling

Cochannelling is the capability of accessing a single peripheral device through either of two physical paths. Under OS/3, it provides for the support of both the dual access and dual channel capabilities of the 90/30 hardware.

Dual access cochannelling permits simultaneous I/O operations (read/read, read/write, write/write) on any two devices using two control units and two selector channels. Each input/output device is connected to both control units, one control unit on each selector channel. Depending on the control units used, dual access cochannelling is applicable to SPERRY UNIVAC 8414 Disk Subsystems and UNISERVO 12, 16, and 20 Magnetic Tape Subsystems on selector channels.

Dual channel cochannelling provides for nonsimultaneous access to a single control unit from either of two selector channels. The devices are connected to one control unit, which is connected to both selector channels. When one channel is busy, the second channel is used to access the device, thereby avoiding a wait for the busy channel. Depending on the control units used, dual channel cochannelling is applicable to SPERRY UNIVAC 8411 and 8414 Disc Subsystems, and UNISERVO 10, 12, 14, 16, and 20 Magnetic Tape Subsystems on selector channels.

2.2.14. Disk Seek Separation

This feature provides for the execution of seek commands to the requested devices before executing the data transfer command, thereby freeing the I/O channel during the device positioning time. This is accomplished by executing seek commands to all devices that have queued requests whenever the channel is free (channel end status received). The data transfer (reads and writes) on the channel are executed as soon after the positioning of the head (device end status) is completed.

Seek separation will increase the number of I/O requests that can be completed in a given time frame when multiple devices are in use on a channel. It will not change the sequence of a job I/O request, but it may change the sequence of various job I/O requests with regard to execution.

2.2.15. Error Logging

The error logging function records hardware errors for later statistical and historical use by Sperry Univac customer engineers. It is a default feature of all OS/3 supervisors above the minimum PIOCS configuration and can be turned on and off, and suspended and resumed, by system console commands. At the time the error occurs, pertinent information is stored temporarily in the error log file on the system resident volume. The customer engineer can read these records from the error log file into main storage for processing and permanent record, assisting him in maintaining customer equipment.

3. Macro Instruction Conventions

3.1. GENERAL

The OS/3 provides a complement of macro instructions to facilitate service requests between a user program and the supervisor. This set of macro instructions is available only when using the assembler language and cannot be directly evoked when using higher-level languages.

Conventions used in this manual to illustrate the supervisor macro instruction formats and some general rules for writing macro instruction statements are contained in 3.2.

General rules and conventions for writing programs using the SPERRY UNIVAC 9000 Series assembler coding form are contained in 3.3.

3.2. FORMAT ILLUSTRATION AND STATEMENT CONVENTIONS

The general format of a macro instruction is:

| LABEL | Δ OPERATION Δ | OPERAND |
|------------------|-------------------|------------|
| symbolic name | macro mnemonic | parameters |

- A symbolic name can appear in the label field. It can have a maximum of eight characters and must begin with an alphabetic character.
- The appropriate macro instruction mnemonic must appear in the operation field and identifies the operation or service requested.
- When parameters are specified in the operand field, they must be positional parameters or keyword parameters as required by the particular function.
- Parameters must not be separated by blanks.
- Assembler rules regarding blank columns and continuation of the operand field must be followed.

The conventions used to delineate the supervisor macro instructions are as follows:

- Capital letters, commas, parentheses, and equal signs must be coded exactly as shown.

Examples:

R
ALL
(1)
SIZE=

- Lowercase letters and words are generic terms representing information that must be supplied by the user. Such lowercase terms may contain hyphens and acronyms (for readability). Acronyms that form part of the variable symbolic name remain capitalized.

Examples:

symbol
start-addr
number-of-bytes
param-1
CCB-name

- Information contained within braces represents mandatory entries of which one must be chosen.

Examples:

{ PC
IT
AB }

{ input-area
(1) }

- Information contained within brackets represents optional entries that (depending upon program requirements) are included or omitted. Braces within brackets signify that one of the specified entries must be chosen if that parameter is to be included.

Examples:

[,entry-number]
[,R]
[, { CCB-name
ALL
(1) }]
[,ERROR=symbol]
[,WAIT=YES]

- When an uppercase portion of a parameter is underlined, only that portion need be coded. For example:

PR:xv

can be coded as either P:12 or PR:12.

- An ellipsis (series of three periods) indicates the omission of a variable number of entries.

Example:

CCB-name-1, ..., CCB-name-n

- An optional parameter that has a list of optional entries may have a default specification that is supplied by the operating system when the parameter is not specified by the user. Although the default may be specified by you with no adverse effect, it is considered inefficient to do so. For easy reference, when a default specification occurs in the format delineation it is printed on a shaded background. If, by parameter omission, the operating system performs some complex processing other than parameter insertion, it is explained in an "if omitted" sentence in the parameter description.

Example:

[, { **S** }
 { **M** }]

- Positional parameters must be written in the order specified in the operand field and must be separated by commas. When a positional parameter is omitted, the comma must be retained to indicate the omission, except for the case of omitted trailing parameters.

Examples:

Assume that LOAD is a supervisor macro instruction with one mandatory positional parameter (phase-name) and four optional positional parameters (load-addr, error-addr, and R):

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|---|
| [symbol] | LOAD | { phase-name } [, { load-addr }] (1) (0) [, { error-addr }] [, R] [, DA] (r) |

Macro instruction statements might be written:

| 1 | LABEL | Δ OPERATION Δ | OPERAND | Δ | COMMENTS |
|---|-------|-----------------------------|---------|---|----------|
| | LOAD | RECAPLOS, INADDR, ERADDR, R | | | |
| | LOAD | RECAPLOS, ERADDR | | | |
| | LOAD | RECAPLOS, INADDR | | | |
| | LOAD | RECAPLOS | | | |

- A keyword parameter consists of a word or a code immediately followed by an equal sign, which is, in turn, followed by a specification. Keyword parameters can be written in any order in the operand field. Commas are required only to separate parameters.

Examples:

Assume that PCA is a supervisor macro instruction with two mandatory keyword parameters (IOAREA1 and BLKSIZE) and nine optional keyword parameters (EODADDR, FORMAT, KEYLEN, LACE, LBLK, SEQ, SIZE, UOS, and VERIFY):

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|---|
| [symbol] | PCA | IOAREA1=area-name ,BLKSIZE=n [,EODADDR=end-of-data-addr] [,FORMAT=NO] [,KEYLEN=n] [,LACE=n] [,LBLK=n] [,SEQ=YES] [,SIZE=n] [,UOS=n] [,VERIFY=YES] |

Macro instruction statements might be written:

| LABEL | △ OPERATION △ | OPERAND | △ | COMMENTS |
|-------|---------------|---|---|----------|
| 1 | PCA | IOAREA1=WORKAREA, BLKSIZE=256, FORMAT=NO, EODADDR=ENDNAME, SEQ=YES, SIZE=1, UOS=1, VERIFY=YES | | X |
| | PCA | EODADDR=ENDNAME, IOAREA1=INAREA, UOS=1, BLKSIZE=256 | | |

- The option to use register preloading is indicated by a register number enclosed in parentheses and may be shown as (1), (0), (15), or (r). This indicates that, instead of entering a symbolic address or a value as the parameter in the macro instruction, you intend to load the designated register with the required data prior to the execution of the macro instruction. For example, in the format illustration:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|---|
| [symbol] | GETCS | { input-area } [{ (0) }] (1) [1] [{ error-addr }] [{ n }] (r) [80] |

The optional entries (1) and (0) refer to registers 1 and 0. The optional entry (r) refers to a register (other than 1 or 0) to be designated by you in the macro instruction statement. For example, the instruction:

| 1 | LABEL | ΔOPERATIONΔ | OPERAND | Δ |
|---|-------|-------------|--------------------|---|
| | | 10 16 | | |
| | | GETCS | WORK, (0), ERRADDR | |

Specifies the input area (positional parameter 1) as WORK and the error address (positional parameter 3) as ERRADDR. It also specifies that, at the time this macro instruction is executed, register 0 will contain the number of records to be read (positional parameter 2).

Note the use of the shaded entry 1, which means that an entry of one as the number of records is assumed if you omit positional parameter 2; and the shaded entry 80, which means a record image of 80 bytes is to be read if you omit positional parameter 4.

3.3. USE OF THE ASSEMBLER CODING FORM

To convert your written program to a form that can be conveniently inputted to the computer, your written work is keypunched into 80-column cards. To make the job of the programmer, keypunch operator, and any other person who may reference this program easier, there are conventions for writing and reading programs and reference materials. A useful tool is the 9000 Series assembler coding form. (See Figure 3-1.)

Theoretically, you could write your program on a plain sheet of paper, as long as you observe the assembly language formatting rules. Using an assembler coding form, however, will ease the job, both for you and for the keypunch operator, who must prepare the punched card deck from your written program.

The paragraphs that follow describe the conventions and rules that apply to the use of this form. Following these rules will result in a stylized assembly listing that is easy to read and use, in addition to ensuring that your program executes properly. The assembler user guide, UP-8061 (current version) gives a detailed description of how to use the coding form. However, some of the rules and conventions are included here for your convenience.

3.3.2. Operation Field

The operation code is written in the operation field (column 10 through 14). These codes specify the operation to be performed. The rules for using this field are:

1. The operation code must not contain embedded blanks.
2. The operation code must be written exactly as shown in the list of mnemonics for application instructions, directives, and macro proc instructions.
3. The operation field must be terminated by a blank.
4. An operation code consisting of six characters, (for example, the macro instruction ATTACH) will fall in columns 10 through 15. In this case, column 16 must be blank to terminate the operation field.

3.3.3. Operand Field

The operand field begins in column 16 and usually ends in or before column 71. The operands that form part of the assembler statements are written in this field. The rules for using this field are:

1. The operand field is terminated by a blank that is not enclosed by an apostrophe.
2. Operands may be continued onto the next line by placing a nonblank character in column 72. The continuation line starts at column 16. Up to two continuation lines are permitted.

3.3.4. Comments Field

Program documentation is as important to the programmer writing the program as it is to those who must refer to it later. Operand specification is usually completed by column 40, thus leaving column 41 through 71 free for comments. There must be at least one blank between the end of the operand specification and the start of the comments. Long comments can be entered by coding an asterisk in column 1.

3.3.5. Continuation Column

When the operand specification is to be continued onto the next line, a nonblank character must be written in column 72. Do not confuse this with continuing a comment. An operand specification can be continued for a total of three lines. The second and third continuation lines start in column 16.

3.3.6. Sequence Field

Columns 73 through 80 may be used for entering sequence numbers. This is done by assigning consecutive numbers to each line of coding and is useful for reassembling the card deck if it should be dropped. It is good practice to number the lines in multiples of 10, or even 100. This allows you to insert additional coding lines without having to renumber the cards when they have been keypunched prior to the modification. Some programmers use letters in addition to the numbers. This is useful in identifying the deck from which cards have come if they have been removed for any reason.

3.4. MACRO INSTRUCTIONS

3.4.1. Declarative Macro Instructions

Declarative macro instructions generate nonexecutable code sequences in the user program and are used to allocate areas in main storage containing control information for various system services.

3.4.2. Imperative Macro Instructions

Imperative macro instructions generate executable code sequences in the user program. These code sequences make up the interface between the user program and the supervisor. Imperative macro instructions are used to request services of the supervisor or to direct the operation of the user program.

3.4.3. Summary of Supervisor Macro Instructions

→ Table 3—1. is a list of the OS/3 supervisor macro instructions and a brief statement of the service performed by each. In this list, ARGLST, BCW, CCW, CCB, PIOC, DTFPF, PCA, SAT, TCA, ECB, DDCPF, and DCFLT are declarative macro instructions; the remainder are imperative macro instructions. Complete descriptions of the macro instructions are contained in Sections 4 through 11 of this manual in the same functional groups as indicated in the table.

Table 3-1. Supervisor Macro Instructions (Part 1 of 3)

| | |
|---|--|
| PHYSICAL INPUT/OUTPUT CONTROL SYSTEM | |
| Physical Input/Output Control | |
| BCW | Generate buffer control word. |
| CCW | Generate channel command word. |
| CCB | Generate command control block. |
| PIOCB | Generate physical input/output control block. |
| RDFCB | Read file control block. |
| EXCP | Execute channel program. |
| SWAP | Access the next physical input/output device. |
| Input/Output Synchronization | |
| WAIT | Wait for one or all input/output requests to complete. |
| WAITM | Wait for one of several input/output requests to complete. |
| SPACE MANAGEMENT | |
| Disk | |
| ALLOC | Assign space to a new disk file or to an existing disk file. |
| EXTEND | Assign additional space to an existing disk file. |
| SCRATCH | Deallocate one or more disk files. |
| RENAME | Rename a disk file. |
| OBTAIN | Access VTOC user block. |
| Diskette | |
| ALLOC | Assign space to a new diskette file. |
| SCRATCH | Deallocate a diskette file. |
| OBTAIN | Obtain diskette label information. |
| SYSTEM ACCESS TECHNIQUE (SAT) | |
| Disk SAT | |
| DTFPF | Define a partitioned file. |
| PCA | Define a partition control appendage. |
| OPEN | Open a disk file. |
| GET | Retrieve next logical block. |
| PUT | Output a logical block. |
| WAITF | Wait for block transfer. |
| READE | Search track by key, equal. |
| READH | Search track by key, equal or higher. |
| SEEK | Access a physical block. |
| CLOSE | Close a disk file. |
| Tape SAT | |
| SAT | Defines magnetic tape file. |
| TCA | Define a tape control appendage. |
| OPEN | Open a tape file. |
| GET | Get next logical block. |
| PUT | Output next logical block. |
| WAITF | Wait for block transfer. |
| CNTRL | Control tape unit functions. |
| CLOSE | Close a tape file. |

Table 3-1. Supervisor Macro Instructions (Part 2 of 3)

| | |
|---------------------------|---|
| MULTITASKING | |
| Task Management | |
| ECB | Generate an event control block. |
| ATTACH | Create and activate an additional task. |
| DETACH | Terminate a task normally. |
| TYIELD | Deactivate a task. |
| AWAKE | Reactivate an existing nonactive task. |
| CHAP | Change the priority of a task. |
| Task Synchronization | |
| WAIT | Wait for a task request to complete. |
| WAITM | Wait for one of several task requests to complete. |
| POST | Activate the waiting task. |
| PROGRAM MANAGEMENT | |
| Program Loader | |
| LOAD | Load a program phase and return control. |
| LOADR | Load a program phase, relocate address-constants, and return control. |
| LOADI | Locate a program phase and store its phase header in a work area. |
| FETCH | Load a program phase and branch. |
| Job and Task Termination | |
| EOJ | Terminate a job step normally. |
| CANCEL | Terminate a job abnormally. |
| Timer Services | |
| GETIME | Obtain current time and date. |
| SETIME | Set an elapsed time counter for the requesting task. |
| Subroutine Linkage | |
| CALL/VCALL | Call a program. |
| ARGLST | Generate an argument list. |
| SAVE | Save register contents. |
| RETURN | Restore registers and return. |
| Island Code Linkage | |
| STXIT | Link to island code subroutine. |
| EXIT | Exit from island code subroutine. |

Table 3-1. Supervisor Macro Instructions (Part 3 of 3)

| | |
|--|---|
| PROGRAM MANAGEMENT (cont) | |
| System Information Control | |
| GETCOM | Retrieve data from job communication area. |
| PUTCOM | Place data into job communication area. |
| GETINF | Retrieve data from system control tables. |
| Control Stream Reader | |
| GETCS | Retrieve embedded data file submitted in job control stream. |
| SETCS | Reset pointer to embedded data file. |
| DIAGNOSTIC AND DEBUGGING | |
| Storage Displays | |
| SNAP/SNAPF | Printout portions of main storage and return control. |
| DUMP | Printout the job main storage and terminate the job step. |
| Checkpoint Facility | |
| CHKPT | Record a checkpoint. |
| DDCPF | Define a disk checkpoint file. |
| DCPOP | Open a disk checkpoint file. |
| DCPCLS | Close a disk checkpoint file. |
| DCFLT | Generate a file list table. |
| Monitor and Trace | |
| // OPTION TRACE | Monitor from start of job. (This is a job control statement, not a macro instruction.) |
| MESSAGE DISPLAY, LOGGING, AND OPERATOR COMMUNICATION | |
| WTL | Write a message into system log file. |
| WTLD | Write a message into system log file after displaying on system console. |
| GETMSG | Retrieve message from canned message file. |
| OPR | Display a message to operator on system console. |
| OTHER SERVICES | |
| Spooling | |
| BRKPT | Create a breakpoint in a spool output file. |



**PART 2. PHYSICAL INPUT/OUTPUT
CONTROL**



4. Physical Input/Output Control System

4.1. GENERAL

The resident supervisor of OS/3 contains a set of routines called the physical input/output control system (IOCS) that controls the activity between the processor and all peripheral devices connected to the multiplexer, selector, and integrated channels. These input/output (I/O) channels operate independently of the processor and allow I/O operations on a channel to overlap with processing and with operations on other I/O channels.

Physical IOCS:

- schedules I/O requests to maintain optimum I/O throughput without burdening the problem program;
- initiates I/O operations;
- tests for error or other exceptional conditions pertinent to the actual physical transfer of data; and
- activates error recovery procedures in the event of peripheral device errors.

Problem program interface to the IOCS is provided at two levels: data management (logical I/O control system) and physical IOCS macro instructions.

Data management routines substantially reduce programming effort, especially for jobs requiring a great amount of I/O processing. The routines, by handling the foregoing I/O functions for the programmer automatically, enable you to concentrate on the logical record, because the applicable physical IOCS macro instructions are contained in the data management macro routines and you need only limited knowledge of the peripheral device. The data management macro instructions are described in the data management user guide, UP-8068 (current version).

The use of the physical IOCS macro instructions may be advantageous for certain programs, which, because of unique I/O devices, need to control the actual handling of the data to be read or written. To use physical IOCS macro instructions, you must have an in-depth knowledge of the particular peripheral device and its control requirements. At the physical IOCS level, the problem program is responsible for performing functions such as:

- constructing the actual I/O commands processed by the device as well as constructing the control blocks used by physical IOCS for issuing the I/O order;
- ensuring the desired sequence of I/O commands by the proper use of I/O synchronization macro instructions;
- blocking/deblocking logical records;
- alternating I/O buffer areas;
- detecting wrong-length records;
- handling end-of-file (EOF) or end-of-volume (EOV) conditions;
- processing labels;
- translating ASCII data to EBCDIC on input, or EBCDIC data to ASCII on output; and
- handling unique error conditions.

4.2. PHYSICAL I/O CONTROL

4.2.1. General

Detailed tabular information pertaining to each request must be supplied if the problem program is to communicate effectively with the IOCS facilities of the resident supervisor through the physical IOCS macro instructions.

The following physical IOCS macro instructions are available for establishing the tabular information and for requesting services of the supervisor and the IOCS:

- Table generation macro instructions (declarative)

BCW

Constructs a buffer control word (BCW), which is used by the integrated I/O channels and multiplexer channel.

CCW

Constructs a channel command word (CCW), which is used by the selector I/O channel and the physical device.

CCB

Constructs a command control block (CCB), which is used as a bidirectional communications medium between the problem program and the IOCS routines in the supervisor.

PIOCB

Constructs a physical input/output control block (PIOCB), which is used as a buffer for file control blocks (FCB) containing file and device information that is compiled by job control at the time the job control stream is processed.

■ Service request macro instructions (imperative)

RDFCB

Reads a file control block (FCB), which completes the PIOCIB with information compiled at job execution time by job control. (The RDFCB macro instruction must be executed prior to any service for an associated PIOCIB.)

EXCP

Requests execution of a channel program. The EXCP macro instruction initiates the physical IOCS routine. Before this instruction can be executed, you must construct an I/O control packet that consists of a CCB, a CCW or a BCW, and a PIOCIB.

SWAP

Accesses the next physical I/O device as allocated by job control.

The relationship of the basic physical IOCS macro instructions is illustrated in Figure 4-1.

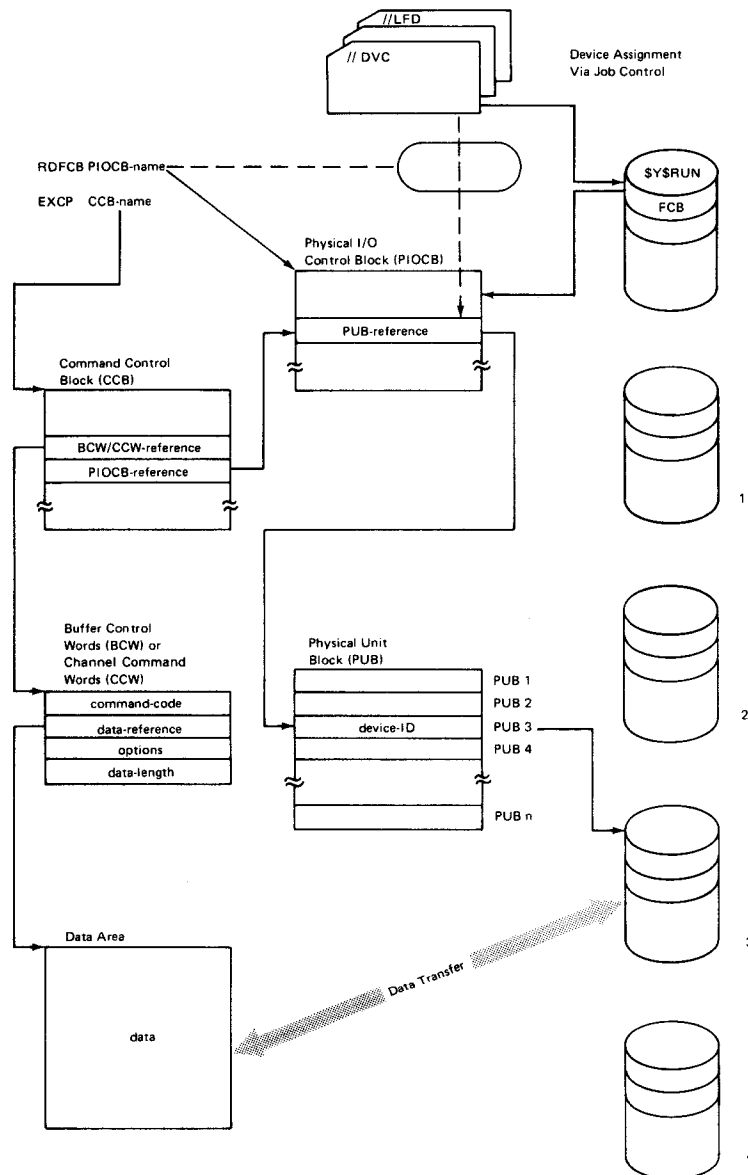


Figure 4-1. Relationship of Basic Physical IOCS Macro Instructions

4.2.2. General I/O Usage Requirements

The users of I/O facilities are required to perform certain prerequisites for I/O communication. These include:

- Description of the file to the operating system through DVC, LBL, or LFD statements.
- Description of the file to the data management system through file description tables and file control routines.

Description of the file to the operating system is through job control statements which describe the device to be used, the volume which contains the file, and the logical name assigned to the file.

Description of the file to the data management system includes the option of linking to a standard data management file control module, using a resident module, or assembling and/or linking a special tailored module with the user program.

The file description table must be included with the user program.

The macro instructions used in the I/O system are best described at the levels at which they are employed.

- User level macro instructions

The execution of imperative macros (EXCP, RDFCB, SWAP) results in control being passed to the appropriate control routine within the operating system. You specify the name of the file, which is the name that was assigned to the file control block by an entry in the label field of the PIOCB macro instruction.

Example:

| | | |
|--------|-------|--------|
| | RDFCB | MASTER |
| | EXCP | FILEIN |
| | WAIT | FILEIN |
| | . | |
| | . | |
| FILEIN | CCB | |
| MASTER | PIOCB | |

The PIOCB declarative macro instruction reserves an area which is the repository of the file control block. The name assigned to the PIOCB must be a duplicate of the character string in the LFD job control statement.

■ Data management level macro instructions

Execution of the imperative I/O macro instructions results in the data management file control routine reducing your macro to a new level of imperative macro instructions. These include the RDFCB (read file control block), the EXCP (execute channel program), and the WAIT (wait for channel program completion) macro instructions.

The primary parameter to the EXCP and WAIT macro instructions is the CCB. The CCB macro provides the ability to specify a particular command to a particular device.

4.2.3. Generate Buffer Control Word (BCW)

Function:

The BCW macro instruction generates a buffer control word which provides the hardware parameter interface to the integrated disk adapter, integrated peripheral channel, multiplexer channel, and the integrated line adapters for use by the physical IOCS routines. Also, the BCW macro instruction provides you with a limited device-independent interface across selector channel devices. In this case, the physical IOCS routines construct a CCW chain by using the information provided in the BCW. The formats of the BCW are shown in Figures 4-2, 4-3, and 4-4.

Note that the BCW of formatting commands sent to the 8411 and 8414 disk subsystems must specify a single record.

This is a declarative macro instruction and must not appear in a sequence of executable code.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|--------|---------------|---|
| symbol | BCW | device-cmd-code [,data-addr] [,data-flag] [,data-byte-count] [,repl-addr] [,repl-flag] [,repl-byte-count] [,control-flag] |

Label:

symbol

Specifies the symbolic address of the buffer control word. This name is used to refer to the BCW.

Positional Parameter 1:

device-cmd-code

Specifies the actual device command code that directs the operation of the I/O device. (For a complete description of the command codes for a particular device, refer to the appropriate subsystem programmer reference manual.)

If omitted, 16 bytes containing 0's are reserved for the BCW, and the assembly listing will contain an error note.

Positional Parameter 2:**data-addr**

Specifies the symbolic address of the data being transferred. This is the active buffer for the system console and the integrated line adapters. For the read/punch, it is the address of the punch output buffer. This parameter is required if data is being transferred to or from storage.

If omitted, the data address field in the BCW is set to 0's, and the assembly listing will contain an error note.

Positional Parameter 3:**data-flag**

Specifies the flag byte associated with the address of the active buffer. This is written in the form X'xx' as follows:

For the integrated disk adapter:

X'40' Indicates a search operation is to be performed on an entire cylinder rather than a track.

X'80' Indicates no data to be transferred.

For the integrated peripheral channel:

X'20' Indicates no data to be transferred. (This entry can also be used for the multiplexer channel.)

X'80' Indicates a replacement operation is to be performed. If this entry is used, positional parameters 5, 6, and 7 are also required.

If omitted, X'00' is assumed, indicating normal operation as specified by the device command code, data address, and data byte count fields in the BCW.

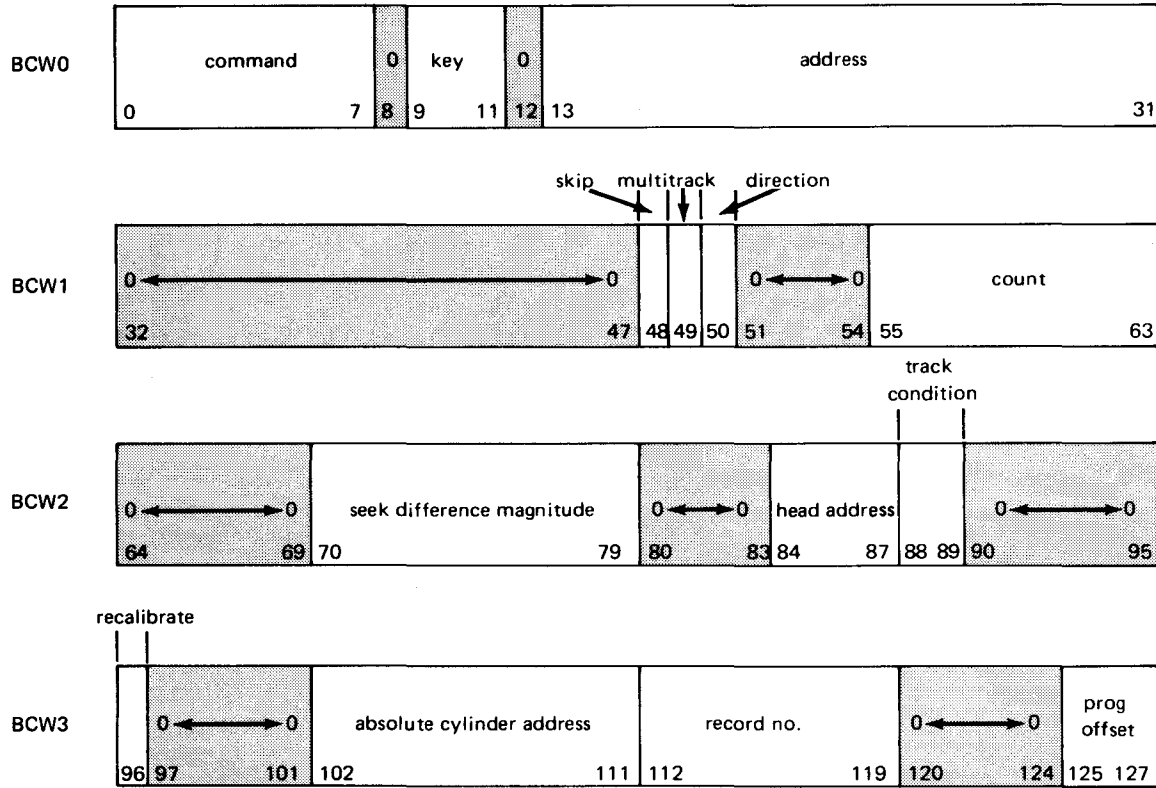
Positional Parameter 4:**data-byte-count**

Specifies the number of bytes to be transferred.

If omitted, zero is assumed. For a search on the integrated disk adapter, this indicates the maximum number of bytes are to be transferred; and for a read or a write, this indicates no data is to be transferred. For the integrated peripheral channel, this indicates the maximum number of bytes are to be transferred.

NOTE:

Positional parameters 5, 6, 7, and 8 apply only to the integrated peripheral channel.



| Bits | Allocation | Function |
|-------|--------------|---|
| 0-7 | Command code | Command code to be executed by IDA; bits 0-3 must be zero |
| 8 | | Unassigned; must be set to zero |
| 9-11 | Key | 3-bit field containing storage protection key |
| 12 | | Unassigned; must be set to zero |
| 13-31 | Address | Storage address on which command operates |


Figure 4-2. Buffer Control Word (BCW) Format for Integrated Disk Adapter (Part 1 of 3)

| Bits | Allocation | Function |
|-------|---------------------------|--|
| 32-47 | | Unassigned; must be set to zero |
| 48 | Skip sentinel | Set with read data command to indicate data transfers inhibited to main storage; set with search/read commands to indicate search begins at index |
| 49 | Multitrack sentinel | Set to 1 with search/read command to indicate search limited to cylinder boundaries rather than single track |
| 50 | Direction sentinel | If 1, specifies accessor moves in direction of decreasing cylinder numbers |
| 51-54 | | Unassigned; must be set to zero |
| 55-63 | Count | On search/read commands - number of bytes to be searched On data read or write commands - number of records to be processed |
| 64-69 | | Unassigned; must be zero |
| 70-79 | Seek difference magnitude | During seek operation, specifies magnitude of difference between accessor present position and desired position |
| 80-83 | | Unassigned; must be set to zero |
| 84-87 | Head address | 4-bit field specifying current operation head address |
| 88,89 | Track condition | Condition of track where operation acts |
| 90-95 | | Unassigned; must be set to zero |

Figure 4-2. Buffer Control Word (BCW) Format for Integrated Disk Adapter (Part 2 of 3)

| Bits | Allocation | Function | | | | | | | | | | | | |
|-------------|--|--|-------------|------------------------------------|-------------|--|-------------|--------------|-------------|--------------|-------------|----------------------|-------------|-------------------|
| 96 | Recalibrate | Set to 1 — accessor reoriented and moved to cylinder 0; overrides bits 71—79 and 50 | | | | | | | | | | | | |
| 97—101 | | Unassigned; must be set to zero | | | | | | | | | | | | |
| 102—111 | Absolute cylinder address | Final position of accessor after completed seek or recalibrate | | | | | | | | | | | | |
| 112—119 | Record number | Number of record where operation is performed or initiated | | | | | | | | | | | | |
| 120—124 | | Unassigned; must be set to zero | | | | | | | | | | | | |
| 125—127 | Programmer offset | <table border="0"> <tr> <td>Bit 125 = 1</td> <td>Programmed offset used for command</td> </tr> <tr> <td>Bit 125 = 0</td> <td>Programmed offset not used; bits 126 and 127 ignored</td> </tr> <tr> <td>Bit 126 = 1</td> <td>Major offset</td> </tr> <tr> <td>Bit 126 = 0</td> <td>Minor offset</td> </tr> <tr> <td>Bit 127 = 1</td> <td>Offset away from hub</td> </tr> <tr> <td>Bit 127 = 0</td> <td>Offset toward hub</td> </tr> </table> | Bit 125 = 1 | Programmed offset used for command | Bit 125 = 0 | Programmed offset not used; bits 126 and 127 ignored | Bit 126 = 1 | Major offset | Bit 126 = 0 | Minor offset | Bit 127 = 1 | Offset away from hub | Bit 127 = 0 | Offset toward hub |
| Bit 125 = 1 | Programmed offset used for command | | | | | | | | | | | | | |
| Bit 125 = 0 | Programmed offset not used; bits 126 and 127 ignored | | | | | | | | | | | | | |
| Bit 126 = 1 | Major offset | | | | | | | | | | | | | |
| Bit 126 = 0 | Minor offset | | | | | | | | | | | | | |
| Bit 127 = 1 | Offset away from hub | | | | | | | | | | | | | |
| Bit 127 = 0 | Offset toward hub | | | | | | | | | | | | | |

LEGEND:

 System-supplied data


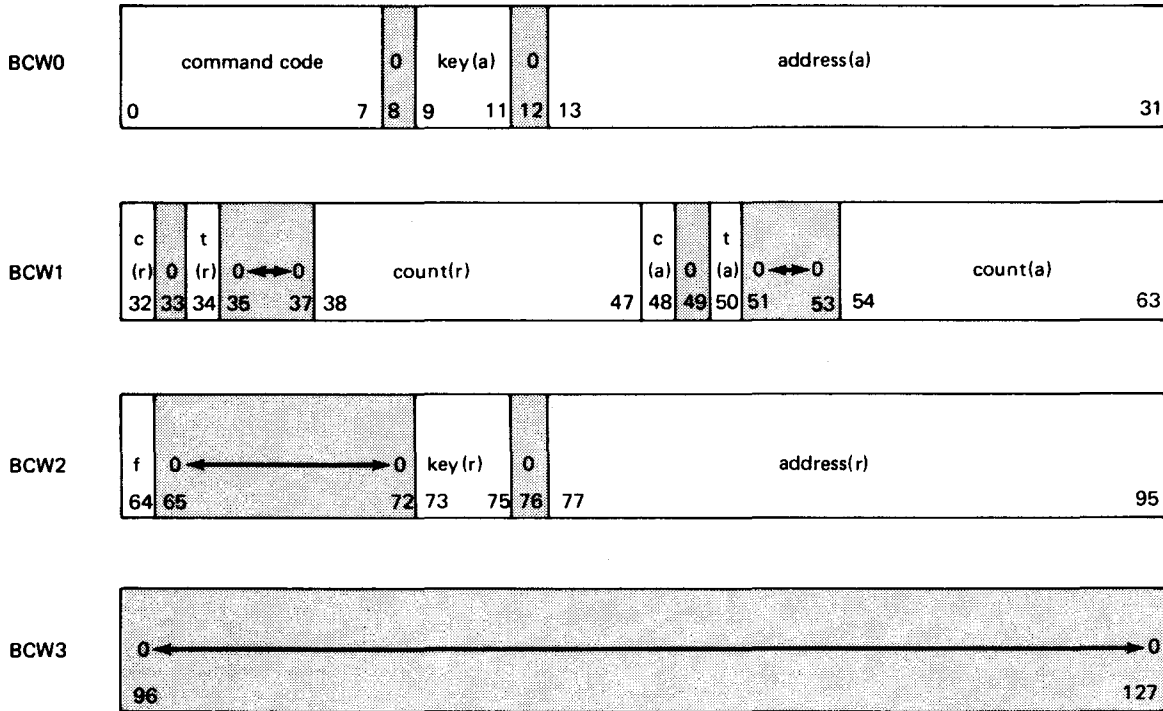
 Data supplied by the user via the macro instruction that directs the supervisor to generate the control block

Figure 4—2. Buffer Control Word (BCW) Format for Integrated Disk Adapter (Part 3 of 3)



| Bits | Allocation | Function |
|-------|--------------|--|
| 0-7 | Command code | Field accessed by IPC during SIO instruction |
| 8 | | Unassigned; must be set to zero |
| 9-11 | Key (a) | 3-bit field containing I/O storage protection key |
| 12 | | Unassigned; must be set to zero |
| 13-31 | Address (a) | Allows IPC to reference any byte in main storage during data transfer sequences Bit 31 = 0 Most significant byte of addressed half word Bit 31 = 1 Least significant byte of addressed half word |

Figure 4-3. Buffer Control Word (BCW) Format for Integrated Peripheral Channel (Part 1 of 3)

| Bits | Allocation | Function |
|-------|------------|---|
| 32 | c (a) | Specifies data chaining operations when set to 1 |
| 33 | | Unassigned; must be set to zero |
| 34 | t (a) | <p>Single control bit used with c(a) bit:</p> <p>c(a) = 0 and t = 0 Use a fields for current data transfer sequence (no data chaining)</p> <p>c(a) = 0 and t = 1 Terminates control</p> <p>c(a) = 1 and t = 0 Use a fields for current data transfer sequence (data chaining initial a and r setting)</p> <p>c(a) = 1 and t = 1 a fields depleted; replacement operation required</p> <p>t(a) and c(a) = 1:</p> <p>f = 0 Terminates with buffer wraparound error</p> <p>f = 1, c(r) = 1 or 0, t(r) = 1 Terminates normally</p> <p>f = 1, c(r) = 0, t(r) = 0 Normal data transfer; no chaining</p> <p>f = 1, c(r) = 1, t(r) = 0 Normal data transfer with chaining</p> |
| 35-37 | | Unassigned; must be set to zero |
| 38-47 | Count (r) | Byte count required for all data transfer operations |
| 48 | c (a) | Specifies data chaining operations when set to 1 |
| 49 | | Unassigned; must be set to zero |
| 50 | t(a) | Same as for bit 34 |
| 51-53 | | Unassigned; must be set to zero |
| 54-63 | Count (a) | Byte count required for all data transfer operations |

Figure 4-3. Buffer Control Word (BCW) Format for Integrated Peripheral Channel (Part 2 of 3)

| Bits | Allocation | Function |
|--------|--------------|--|
| 64 | f (flag bit) | Indicates to IPC that current contents of r fields are valid for replacement operation |
| 65-72 | | Unassigned; must be set to zero |
| 73-75 | Key (r) | 3-bit field containing I/O storage protection key |
| 76 | | Unassigned; must be set to zero |
| 77-95 | Address (r) | Allows IPC to reference any byte in main storage during data transfer sequences Bits 31 and 95 = 0 Most significant byte of addressed half word Bits 31 and 95 = 1 Least significant byte of addressed half word |
| 96-127 | | Unassigned; must be set to zero |

LEGEND:



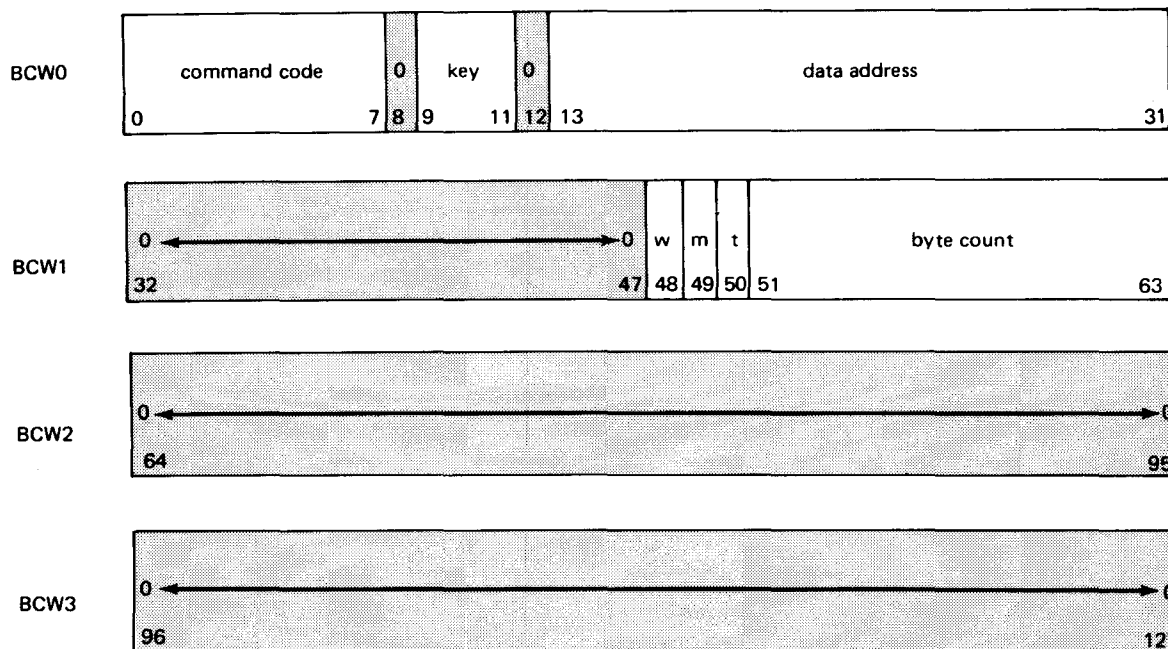
-  = System-supplied data
-  = Data supplied by the user via the macro instruction that directs the supervisor to generate the control block
- a = active
- c = chaining
- f = flag
- r = replacement
- t = transfer

Figure 4-3. Buffer Control Word (BCW) Format for Integrated Peripheral Channel (Part 3 of 3)



| Bits | Allocation | Function |
|--------|--------------|---|
| 0-7 | Command code | Specifies operation to be performed by device and channel |
| 8 | | Unassigned; must be set to zero |
| 9-11 | Key | Contains I/O storage protection key |
| 12 | | Unassigned; must be set to zero |
| 13-31 | Data address | Allows multiplexer channel to reference any byte in main storage during data transfer sequences |
| 32-47 | | Unassigned; must be set to zero |
| 48 | w | w = 0 Input operation (read) w = 1 Output operation (write) |
| 49 | m | m = 0 Ascending address (forward sequence) m = 1 Descending address (reverse sequence) |
| 50 | t | t = 0 Transfer data t = 1 Termination of data transfer |
| 51-63 | Byte count | Contains byte count required for all data transfers |
| 64-127 | | Unassigned; must be set to zero |

LEGEND:

- System-supplied data
- Data supplied by the user via the macro instruction that directs the supervisor to generate the control block

Figure 4-4. Buffer Control Word (BCW) Format for Multiplexer Channel

Positional Parameter 5:

repl-addr

Specifies the symbolic address of the second buffer area. This is the replacement buffer for the system console and the line adapters. For the read/punch, it is the address of the input buffer.

When the byte count decrements to zero during a data transfer operation, this address replaces the data address specified in positional parameter 2.

Positional parameter 3 (data-flag) must be X'80'.

Positional Parameter 6:

repl-flag

Specifies the flag byte associated with the address of the replacement buffer. When the byte count decrements to zero during a data transfer operation, this flag byte replaces the data-flag specified in positional parameter 3. To continue the replacement operation, this entry must be X'80'.

Positional parameter 3 (data-flag) must be X'80'.

Positional Parameter 7:

repl-byte-count

Specifies the number of replacement bytes to be transferred. When the byte count decrements to zero during a data transfer operation, this byte count replaces the data byte count specified in positional parameter 4.

Positional parameter 3 (data-flag) must be X'80'.

Positional Parameter 8:

control-flag

Specifies the control flag for communication devices associated with the line adapters of the integrated peripheral channel. Details of this parameter and its use will be supplied later.

Examples of BCW usage:

| 1 | LABEL | △OPERATION△ 10 16 | OPERAND | △ |
|----|--------|----------------------|---------------------------------|---|
| 1. | CARDIN | BCW | 2, IOAREAL, , 80 | |
| 2. | CDINOT | BCW | 3, IOAREAL, , 80, IOAREA2, , 80 | |
| 3. | PRINT | BCW | 9, IOAREAL, , 132 | |
| 4. | RDDISC | BCW | 2, IOAREAL, , 1 | |

Explanations:

1. Read one 80-column card in EBCDIC mode.
2. Read/punch 80-column card in EBCDIC mode. Punch buffer is IOAREA1; read buffer is IOAREA2.
3. Print 132 positions and advance one line.
4. Read one sector on 8416/18 disk.

NOTE:

The cylinder half word (BCW name+12), the head address byte (BCW name+10), and the record (sector) number byte (BCW name+14) can be set statically by use of the ORG assembler control directive, or dynamically via instruction execution.

4.2.4. Generate Channel Command Word (CCW)

Function:

The CCW macro instruction generates a channel command word which provides the hardware parameter interface to the selector channels for use by the physical IOCS routines. The format of the CCW is shown in Figure 4-5. The format of the CAW, which contains the first CCW address, is shown in Figure 4-6.

The supervisor can only handle command chains on selector devices through two levels of transfer in channel (TIC) within command chain. This limitation is due to the lack of hardware address relocation on CCWs and the need to have a software function perform the absolutizing and relativizing of CCW addresses.

This is a declarative macro instruction and must not appear in a sequence of executable code.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|--------|---------------|--|
| symbol | CCW | [device-cmd-code] [,data-addr] [,flag] [,data-byte-count] |

Label:

symbol

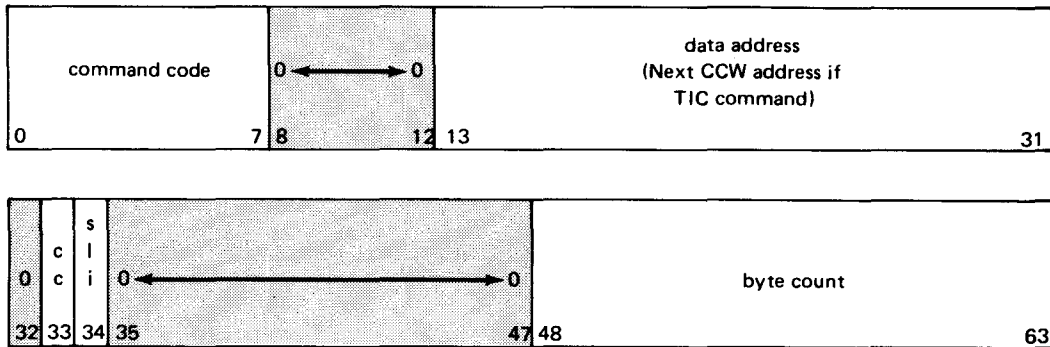
Specifies the symbolic address of the channel command word. This name is used to refer to the CCW.

Positional Parameter 1:

device-cmd-code

Specifies the actual device command code that directs the operation of the I/O device. (For a complete description of the command codes for a particular device, refer to the appropriate subsystem programmer reference manual.)

If omitted, eight bytes containing 0's are reserved for the CCW, and the assembly listing will contain an error note.



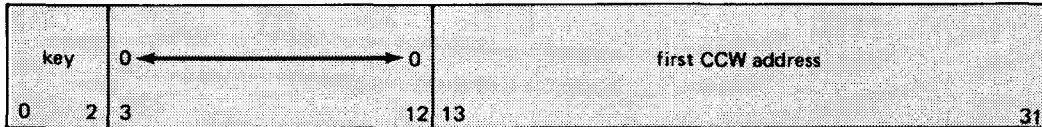
| Bits | Allocation | Function |
|-------|---------------------------------------|---|
| 0-7 | Command code | Specifies operation to be performed by device and channel |
| 8-12 | | Unassigned; must be set to zero |
| 13-31 | Data address | Address of location in main storage into or from which first byte of data is transferred |
| 32 | | Unassigned; must be set to zero |
| 33 | cc (chain command flag) | When valid ending device status received, new CCW fetched and operation specified by new command code initiated |
| 34 | sli (suppress length indication flag) | If set to 1, incorrect length condition not indicated to program; if cc = 1 also, command chaining not suppressed |
| 35-47 | | Unassigned; must be set to zero |
| 48-63 | Byte count | Byte count required for all data transfer operations |

LEGEND:

System-supplied data

Data supplied by the user via the macro instruction that directs the supervisor to generate the control block

Figure 4-5. Channel Command Word (CCW) Format for Selector Channel



| Bits | Allocation | Function |
|-------|-------------------|--|
| 0-2 | Key | I/O storage protection key used by channel for all storage accesses of data and CCWs |
| 3-12 | | Bits set to zero |
| 13-31 | First CCW address | Controls I/O operation initiated by SIO instruction |

LEGEND:


 System-supplied data

Figure 4-6. Channel Address Word (CAW) Format

Positional Parameter 2:

data-addr

Specifies the symbolic address of the data being transferred. This parameter is required if data is being transferred to or from storage.

If omitted, the data address field in the CCW is set to 0's, and the assembly listing will contain an error note.

Positional Parameter 3:

flag

Specifies the flag byte associated with the address of the buffer. This is written in the form X'xx' as follows:

- X'20' Indicates incorrect data length to be suppressed.
- X'40' Indicates command chaining.
- X'60' Indicates both of above.

If omitted, X'00' is assumed, indicating normal operation as specified by the device command code, data address, and data byte count fields in the CCW.

Positional Parameter 4:

data-byte-count

Specifies the number of bytes to be transferred.

If omitted, zero is assumed, resulting in a maximum data transfer.

Examples:

Search key equal/read data (for a following update) on 8414 disk.

| LABEL | ΔOPERATIONΔ | OPERAND | Δ | COMMENTS |
|--------|-------------|----------------------------|---|---|
| EQREAD | CCW | 7, DISKADDR, X'40', 6 | | SEEK |
| NAME1 | CCW | X'12', RECDADDR, X'60', 5 | | Read ID (for a following update) |
| | CCW | X'29', KEYARGUE, X'40', 10 | | Search key equal |
| | CCW | 8, NAME1 | | Repeat search and read ID if not correct record |
| | CCW | 6, IOAREA1, 200 | | Read data |

Format write a data record on 8414 disc.

| | | | | |
|--------|-----|-----------------------------|--|-------------------------------------|
| FORWRT | CCW | 7, DISKADDR, X'40', 6 | | SEEK |
| | CCW | X'31', DISKADDR+2, X'60', 5 | | Search ID equal on preceding record |
| | CCW | 8, *-9 | | Repeat search if not correct record |
| | CCW | X'1D', IOAREA1, 218 | | Format write |
| | CCW | | | |

The 218-byte buffer defined as IOAREA1 contains an 8-byte count field, a 10-byte key field, and a 200-byte data field.

4.2.5. Generate Command Control Block (CCB)

Function:

The CCB macro instruction generates a command control block which serves as a link between the PIOC and the BCW or the CCW. There must be at least one CCB macro instruction for each type of I/O peripheral device to be controlled by physical I/O macro instructions. An active CCB pertains to one I/O request at a time; therefore, each outstanding I/O request must have a unique CCB. The format of the CCB is shown in Figure 4-7. This is a declarative macro instruction and must not appear in a sequence of executable code.

Format:

| LABEL | ΔOPERATIONΔ | OPERAND |
|--------|-------------|--|
| symbol | CCB | PIOC-name, {BCW-name } [, { PUB-entry-number }] [, { error-option }] [, { X'00' }] |

Label:

symbol

Specifies the symbolic address of the command control block. This name is used to refer to the CCB.

Positional Parameter 1:

PIOCB-name

Specifies the symbolic address of an associated physical input/output control block generated by the PIOCB macro instruction. (The address furnished will be modified by this macro instruction to be the address of the PUB address within the PIOCB.)

Positional Parameter 2:

BCW-name

Specifies the symbolic address of a BCW.

CCW-name

Specifies the symbolic address of a CCW, or a list of CCWs if command chaining is used.

When you use data management macro instructions, the BCWs and CCWs are generated automatically. When using physical IOCS macro instructions, you must specify each BCW and CCW according to the I/O functions desired.

Positional Parameter 3:

PUB-entry-number

May be 0, 2, 4, 6, 8, 10, 12, or 14 indicating one of eight 2-byte fields in the PIOCB containing the absolute address of the PUB for the device involved in the I/O operation. (Zero indicates the first entry, 2 the second, 4 the third, etc.)

If omitted, zero is assumed (indicating the first PUB address).

Positional Parameter 4:

error-option

Specifies error acceptance options elected at assembly time. This is written in the form X'xx' as follows:

- X'00' Indicates that no error conditions are acceptable to the problem program.
- X'01' Block number area is reserved in buffer.
- X'02' Reserved for system use.
- X'04' Reserved for system use.
- X'08' Indicates system access CCB. Device independence can be achieved by furnishing a BCW for integrated peripheral.
- X'10' Indicates a diagnostic request. Reserved for system use.
- X'20' Indicates that, following the normal error recovery attempts by the supervisor, those errors classified as unique are acceptable to the problem program. See note 1.
- X'40' Indicates that all unrecoverable error conditions are acceptable to the problem program following the normal error recovery attempts by the supervisor. See note 2.
- X'80' Indicates user has own error code. No recovery will be attempted by the supervisor, and device status and sense are communicated to user in the CCB.

NOTES:

1. *Accept Unique Errors (byte 3, bit 2). Unique errors may be considered as recoverable errors. The meaning of unique errors is different for different devices.*

For a disk, unique error means record not found. Your program may expect that certain records you are looking for in a file may not be there. An example of this is an update-add program. If the record is found, it is updated; if it is not found, it is added to the file. In this case, you should set the accept unique errors bit (byte 3, bit 2) in the CCB. If you receive a no record found condition (byte 2, bit 3), physical IOCS will retry the error twice. If the record is still not found and the CCB is marked to accept unique errors, no error message is displayed on the console and control is returned to your program with the no record found bit set in the CCB.

For tape, unique error means a tape that is busy rewinding. If you issue an EXCP to a magnetic tape which is rewinding, the CCB will be returned with the unique error bit (byte 2, bit 2) set. This occurs whether or not accept unique errors is set in the CCB. The EXCP should be reexecuted until the status does not occur. At that time, the EXCP is considered completed.

↓

For printers, unique error means character mismatch. This means that there is no match between a code in the load code buffer (LCB) and a character in the print line buffer. When you generate the LCB for your printer, you may choose whether or not to report character mismatches. If you choose not to report character mismatches, they will be ignored and no console error message will be displayed. If the LCB is generated so that character mismatches are to be reported and a character mismatch occurs, an error message will be displayed on the console. If the accept unique errors bit is set in the CCB, the options on the error message will be R (retry) or I (ignore).

If the operator responds I, control will be returned to your program with the unique error bit set in the CCB. If the accept unique errors bit is not set, the options on the error message will be R (retry) or C (cancel).

There are no unique errors for readers and punches. Note that, except for tape, if a unique error occurs and the CCB is not marked to accept unique errors, physical IOCS will treat the unique error as an unrecoverable error.

2. *Accept Unrecoverable Errors (byte 3, bit 1). If you set this bit in the CCB and an unrecoverable error occurs, the console message will appear with the R (retry) and U (accept unrecoverable) options. If the operator responds R, the command will be retried. If the operator responds U, control will be returned inline following the command, and the unrecoverable error bit (byte 2, bit 1) will be set in the CCB. If you do not set the accept unrecoverable bit in the CCB and an unrecoverable error occurs, the console message will appear with the R (retry) and the C (cancel) options. After successive retries, if the error still is unrecoverable, the operator may choose to respond C and the job will be cancelled.*

If omitted, the entry X'00' is assumed, indicating that no error conditions are acceptable to the problem program.

↑

The CCB is used to communicate with the functional IOCS routines executing the I/O operations. The generated CCB forms the logical connection between the PIOC and the CCW or the BCW. The PIOC references the actual peripheral device and the CCW or the BCW defines and controls the function of the particular device and its data transfer. The CCB also specifies user options pertinent to the I/O request in the event of an error, and reflects the status of the request. When the related I/O interrupt occurs, the IOCS also stores status information pertinent to the interruption in the associated CCB.

Because the CCB serves as a 2-way communications medium between the IOCS and the problem program, it is used for one active I/O request at a time; therefore, every active I/O request must have a unique CCB.

| Byte | 0 | 1 | 2 | 3 |
|------|---|-----------------|--|----------------|
| 0 | control byte 1 | I/O error count | transmission byte | control byte 2 |
| 4 | TCB address ^① or next CCW address | | | |
| 8 | CCB link | | address ^② or residual CCW byte count | |
| 12 | CCW address | | | |
| 16 | PIOCB pointer (PUB address) | | | |
| 20 | sense byte 0 | sense byte 1 | sense byte 2 | sense byte 3 |
| 24 | sense byte 4 | sense byte 5 | device status | channel status |

NOTES:

- ① During the I/O command execution, contains the address of the TCB associated with this CCB. At I/O command termination, physical IOCS inserts the address of the next CCW in the chain.
- ② During I/O command execution, bytes 8—11 contain the address of the next CCB in the chain at this job level. At I/O command termination, physical IOCS inserts the number of bytes remaining in the CCW byte count (when the I/O command terminated) into bytes 10 and 11.

Figure 4—7. Command Control Block (CCB) Format (Part 1 of 2)

| Byte | Length | Content |
|-------|--------|--|
| 0 | 1 | Control byte 1 Bits 0-2 Reserved 3 1 = Ignore block numbers 4 Reserved 5 1 = CCB in wait condition 6-7 Reserved |
| 1 | 1 | Binary count of errors encountered processing the CCB |
| 2 | 1 | Transmission byte Bit 0 0 = CCB in process 1 = CCB processed 1 1 = Unrecoverable error 2 1 = Unique error 3 1 = No record found 4 1 = Unit exception 5 1 = Block numbers not equal 6 1 = Track end 7 1 = Cylinder end |
| 3 | 1 | Control byte 2 Bit 0 1 = User error recovery 1 1 = Accept unrecoverable errors 2 1 = Accept unique errors 3 1 = Diagnostic CCB. Reserved for system use 4 1 = System access CCB 5 Reserved 6 Reserved 7 1 = Block number area reserved |
| 4-7 | 4 | During I/O command execution, full-word address of TCB associated with this CCB or At I/O command termination, full-word address of next CCW if not at end of command chain |
| 8-11 | 4 | During I/O command execution, full-word address of next CCB or |
| 10-11 | 2 | At I/O command termination, bytes remaining in CCW byte count when I/O command was terminated |
| 12-15 | 4 | Full-word address of first CCW |
| 16-19 | 4 | Address of PIOCB entry which contains the half-word address of PUB associated with this CCB |
| 20-23 | 4 | Sense bytes 0 through 3 |
| 24,25 | 2 | Sense bytes 4 and 5 |
| 26 | 1 | Device status Bit 0 1 = Attention 1 1 = Status modifier 2 1 = Control unit end 3 1 = Busy 4 1 = Channel end 5 1 = Device end 6 1 = Unit check 7 1 = Unit exception |
| 27 | 1 | Channel status Bit 0 0 1 1 = Incorrect length 2 1 = Program check 3 1 = Invalid address 4 1 = Channel data check 5 1 = Interface control check 6 1 = Channel control check 7 1 = Buffer terminate |

Figure 4-7. Command Control Block (CCB) Format (Part 2 of 2)

4.2.6. Generate Physical Input/Output Control Block (PIOCB)

Function:

The PIOCB macro instruction generates a physical I/O control block which provides a buffer into which the RDFCB will transfer the FCB which contains file and device information defined in the job control stream. The format of the PIOCB is shown in Figure 4—8. This is a declarative macro instruction and must not appear in a sequence of executable code.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|--------|---------------|--|
| symbol | PIOCB | $\left[\begin{array}{l} \text{MAX} \\ \text{FCB-length} \\ \text{16} \end{array} \right]$ |

Label:

symbol

Specifies the symbolic address of the physical I/O control block. This name is used to refer to the PIOCB. The characters appearing in this name become the 8-byte character string (file name) generated in the first eight bytes of the PIOCB and used as a search key by the RDFCB macro instruction to locate the file control block.

Positional Parameter 1:

MAX

Specifies that an area is to be reserved within the PIOCB large enough to contain the complete FCB including the 8-byte file name. The size of the FCB area is stored as a binary constant in the 2-byte FCB length field in the PIOCB following the file name.

FCB-length

Specifies the number of bytes to be reserved within the PIOCB for the FCB. The size may be from 16 to 256 bytes. This option is used to limit the size of the PIOCB for the purpose of reading partial file control blocks.

If omitted, a minimum size PIOCB of 16 bytes is generated, allowing for storage of the file name, the FCB-length, and only the first six bytes of the file control block data. These six bytes contain the 4-byte device type code and the absolute address of the physical unit block for the device assigned to the file.

Error-free use of space management functions (for example, those provided by the ALLOC and SCRATCH macro instructions) requires a fairly complete FCB. When you issue an RDFCB and PIOCB macro instruction combination to read the FCB into main storage, do not use the default value (16 bytes) in the PIOCB macro instruction. Instead, either specify the necessary FCB length or, for maximum safety, use the MAX parameter.

The PIOCB macro instruction is used to generate physical I/O control blocks. These blocks serve as repositories for file and device information previously compiled by job control at the time the job control stream was evaluated. This information is stored in the form of a file control block. File control blocks are stored in the job run library on the system resident direct access device.

At assembly time, the PIOC macro instruction provides main storage space for the following information:

- Eight-byte file name (search key)

An 8-byte character string is generated within each physical I/O control block. This character string is required by the RDFCB macro instruction to obtain the file control block. The characters in this 8-byte search key are identical to the characters appearing as the label of the PIOC macro instruction.

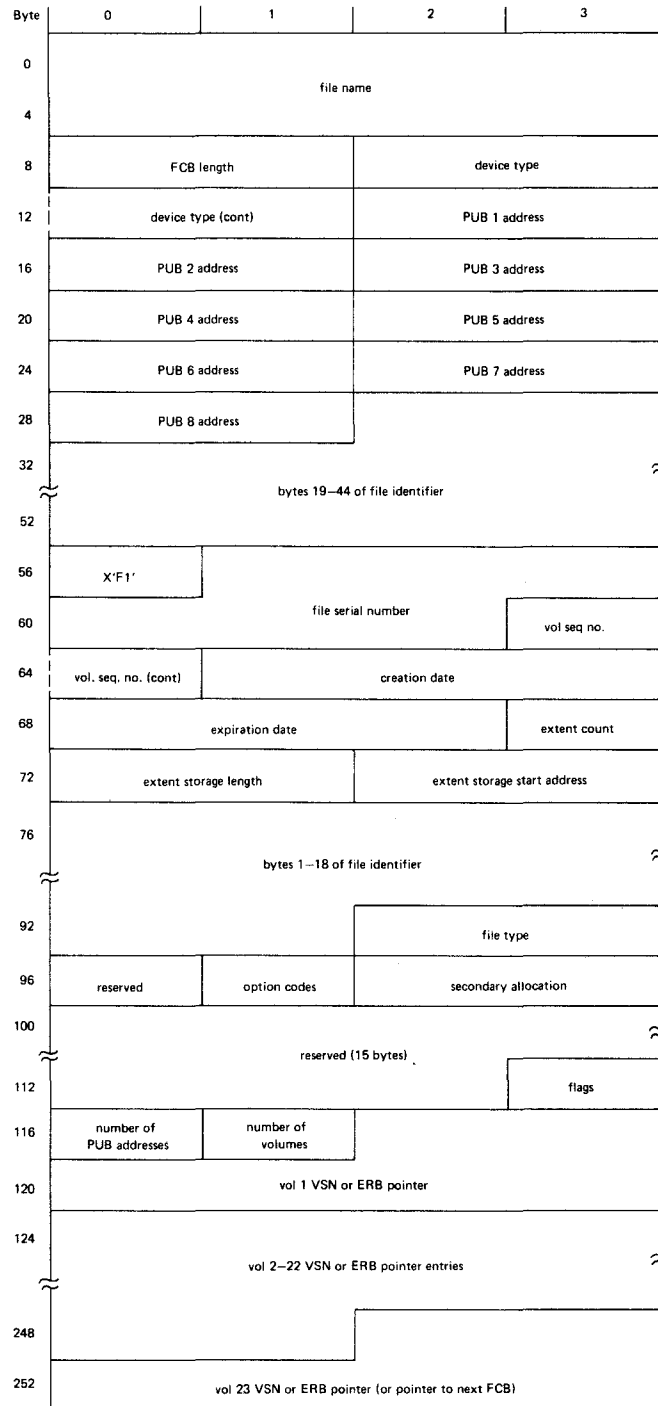


Figure 4-8. Physical I/O Control Block (PIOCB) and File Control Block (FCB) Format (Part 1 of 2)

NOTES:

1. Physical I/O control block (PIOCB) refers to the buffer area generated by the PIOCB macro instruction. File control block (FCB) refers to the file control data read into the buffer area by the RDFCB macro instruction. Minimum PIOCB includes bytes 0 to 15; maximum PIOCB includes bytes 0 to 255.
2. Bytes 118 to 255 consist of a 6-byte field for each volume which contains either a 6-byte volume serial number (VSN) or a 4-byte extent request block (ERB) pointer if the volume was not allocated at the time the FCB was built. If there are more than 23 volumes, the last field contains a pointer to the next FCB.

Figure 4-8. Physical I/O Control Block (PIOCB) and File Control Block (FCB) Format (Part 2 of 2)

- Half-word length field

A 2-byte field immediately follows the 8-byte search key. This field contains a binary count of the number of bytes reserved for the file control block. This binary count ranges from a minimum of 16 to a maximum of 256. Altering the contents of this half-word field is not recommended since the field defines the size of the PIOCB and is used as the requested length of the file control block. The RDFCB macro instruction transfers only the number of bytes equal to this size or less.

- Part or all of a file control block

A 2-byte field is reserved for each device that is allocated to a file. A maximum of eight fields is permitted. The first 2-byte field is referred to in positional parameter 3 of the CCB macro instruction (4.2.5) as PUB-entry-number zero, the second field as entry two, the third field as entry four, and the fourth field as entry six, etc. Following the successful completion of an RDFCB macro instruction, these PUB address fields contain the absolute addresses of the physical unit blocks that identify the assigned devices. Device assignments indicated in the file control block are made by job control from the parameters in the LFD and DVC statements. Thus, the RDFCB macro instruction, in conjunction with the PIOCB macro instruction, dynamically links the problem program with the device allocations made by job control.

4.2.7. Read File Control Block (RDFCB)

Function:

The RDFCB macro instruction locates and transfers the file control block into the physical I/O control block in main storage.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|--|
| [symbol] | RDFCB | $\left\{ \begin{array}{c} \text{PIOCB-name} \\ (1) \end{array} \right\} \left[, \left\{ \begin{array}{c} \text{error-addr} \\ (r) \end{array} \right\} \right]$ |

Positional Parameter 1:

PIOCB-name

Specifies the symbolic address of the physical I/O control block. These characters appear in the first eight bytes of the PIOCB and are used as a search key to identify the desired file control block.

(1)

Indicates that register 1 has been preloaded with the address of the PIOCB.

Positional Parameter 2:

error-addr

Specifies the symbolic address of an error routine to be executed if an error occurs.

(r)

Specifies that register n (other than 0 or 1) has been preloaded with the address of the error routine.

If omitted, the calling task will be abnormally terminated if an error occurs.

The RDFCB macro instruction is used to locate the file control block and read it into the physical I/O control block in main storage. To accomplish this function, positional parameter 1 of the RDFCB macro instruction must be the address of a physical I/O control block that contains an 8-byte character string identifying the desired file control block. This character string is used when locating the file control block. Any references to a physical I/O block, by means of an EXCP macro instruction, before the device assignment fields are filled by the RDFCB macro instruction results in a software validation error. Therefore, each physical I/O block should be initialized by RDFCB macro instruction before the block is referenced by an EXCP macro instruction. Figure 4-1 shows the interrelationship among the command control block, buffer control word, physical I/O control block, file control block, and physical unit block.

Examples:

1. Read file control block for logical file INFILE and return control to symbolic address ERROR if error occurs.

| 1 | LABEL | △OPERATION△ | | OPERAND | △ |
|---|--------|-------------|----|--------------------------|---|
| | | 10 | 16 | | |
| | | RDFCB | | INFILE, ERROR | |
| | | . | | | |
| | ERROR | N | | O ₉ =A(X'FF') | |
| | | C | | O ₉ =A(X'3F') | |
| | | . | | | |
| | INFILE | PIOCB | | | |

- Read file control block for logical file INFILE and return control to symbolic address error.

| 1 | LABEL | △OPERATION△ | | OPERAND | △ |
|---|--------|-------------|----|--------------|---|
| | | 10 | 16 | | |
| | | LA | | 1, INFILE | |
| | | LA | | 10, ERROR | |
| | | RDFCB | | (1), (10) | |
| | | : | | | |
| | | : | | | |
| | ERROR | N | | 0, =A(X'FF') | |
| | | C | | 0, =A(X'3F') | |
| | | : | | | |
| | | : | | | |
| | INFILE | PIOCB | | | |

4.2.8. Execute Channel Program (EXCP)

Function:

The EXCP macro instruction requests that an I/O operation be executed by the physical I/O control system.

Format:

| LABEL | △OPERATION△ | OPERAND |
|----------|-------------|-------------------------|
| [symbol] | EXCP | { CCB-name } [C] (1) |

Positional Parameter 1:

CCB-name

Specifies the symbolic address of the CCB.

(1)

Indicates that register 1 has been preloaded with the address of the CCB.

Positional Parameter 2:

C

Specifies that the I/O request is conditional on the peripheral device not being shared with another job running in the system. This enables you to issue conditional seek commands when running in a multijobbing environment.

If omitted, the I/O request is assumed to be unconditional.

The EXCP macro instruction communicates directly with the I/O scheduler for the purpose of submitting I/O requests to the system. Before the EXCP macro instruction is executed, you must construct an I/O packet consisting of the following:

- Use a CCB macro instruction to define the CCB.
- Use a PIOC macro instruction to define the physical I/O control block.
- Use one or more CCW macro instructions or a BCW to construct the channel program.
- Use an RDFCB macro instruction to identify the I/O device and to obtain file information specified by job control.

Linkage between these components is as follows:

- The EXCP macro instruction passes the address of the CCB to the physical IOCS routines.
- The address of a 2-byte field in a physical I/O control block is stored in the CCB. This field contains the address of the physical unit block for the peripheral device concerned.
- The address of the first CCW or BCW is stored in the CCB.
- Each CCW or BCW contains the address of an input/output data area.

Whenever an EXCP macro instruction is executed, the I/O request counter in the task control block of the requester is incremented and a status indicator in the CCB is set to signify that the order is outstanding. Control is returned to the calling program immediately by the supervisor with the degree of completion of this I/O order uncertain. You must use the WAIT or WAITM macro instruction for synchronization with this I/O.

An EXCP issued to a magnetic tape which is rewinding will result in the posting of the CCB with unique error status. The EXCP should be reexecuted until the status does not occur. At that time the EXCP is considered completed.

4.2.9. Swap I/O (SWAP)

Function:

The SWAP macro instruction is used for multivolume files to access the next physical I/O device in the sequence in which the volumes have been defined in the job control stream.

The SWAP macro instruction replaces the location of the currently active PUB address entry in the CCB with the location of the next PUB address entry from the PIOC. The PUB addresses are swapped in a sequential and circular manner. After the last PUB address entry in the PIOC has been accessed, the next execution of a SWAP macro instruction with the same CCB will access the first PUB address entry.

If more volumes have been defined than PUBs assigned, a console message will request the operator to mount the currently required volume on the device specified in the earliest assigned PUB.

The complete FCB must reside within the PIOCB when this macro instruction is executed. Also, the CCB must be pointing to the currently active PUB address in the PIOCB.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|---------------------|
| [symbol] | SWAP | { CCB-name } (1) |

Positional Parameter 1:

CCB-name

Specifies the symbolic address of the command control block used to control the I/O operations to this file.

(1)

Indicates that register 1 has been preloaded with the address of the CCB.

4.3. INPUT/OUTPUT SYNCHRONIZATION

Macros are available that provide the means by which a task can await the completion of one or more outstanding I/O operations. Specifically the task can await one, several, or all outstanding I/Os; however, the I/O being waited for must have been requested by the task doing the waiting.

Tasks are waited by setting a unique wait bit within that task control block (TCB). These wait bits signal the switcher that this task is nondispatchable and indicate the reason for the wait. Upon clearing the wait bits, the task becomes dispatchable and can be activated.

Two macro instructions are available for I/O synchronization:

- **WAIT**

Wait for one or all I/O requests to complete.

- **WAITM**

Wait for one of several I/O requests to complete.

These macro instructions can also be used (with different parameters) to synchronize a task with the execution of other tasks. For I/O synchronization, the macro instruction references a CCB; and for task synchronization, the macro instruction references an event control block. Task synchronization is described in 7.4.

It must be remembered when you use these macro instructions that only the task having executed an I/O request can await its completion; and when you await a task, it is not valid to await the executing task.

4.3.1. Wait for I/O Completion (WAIT)

Function:

The WAIT macro instruction temporarily suspends program execution until a specified I/O operation is completed (or until all I/O operations in the task are completed). If the related operation is completed, control is returned to the point immediately following the WAIT macro instruction. If the operation is not complete, the task is placed in a wait state and control is passed to another task.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|--|
| [symbol] | WAIT | { CCB-name (1) ALL } [, { branch-addr (15) }] |

Positional Parameter 1:

CCB-name

Specifies the symbolic address of the CCB to be tested for completion.

(1)

Indicates that register 1 has been preloaded with the address of the CCB.

ALL

Specifies that the I/O counter in the task control block is tested instead of the status byte in the CCB. If no orders are outstanding, the problem program continues execution with the instruction following the WAIT macro instruction. If I/O orders are outstanding, the program is suspended until the I/O counter is zero (indicating all orders are completed).

Positional Parameter 2:

branch-addr

Specifies the symbolic address to which program control is transferred if the requested I/O operation is completed and an exception has occurred. The cause of the exception is posted in the appropriate CCB.

NOTE:

When using a label as positional parameter 2, the contents of register 15 are not altered by the WAIT macro instruction even though transfer of control may occur. Also, it is assumed that the base register coverage is provided in the problem program to permit branching to this alternate address.

(15)

Indicates that register 15 has been preloaded with the branch address.

If omitted, the WAIT macro instruction tests for complete or incomplete status without testing for exceptions. If ALL is specified as positional parameter 1, this parameter must be blank.

The WAIT macro instruction is written in the problem program at the point where processing cannot logically proceed until either the completion of related I/O requests initiated by the EXCP macro instruction or synchronization with another task. When utilized for I/O, the WAIT macro instruction is executed in reference to a single CCB or to the I/O counter in the task control block. If the related I/O operation is completed, control is returned immediately and processing continues without interruption. If the I/O operation is not complete, the task is placed in a wait state and program control is passed to another task. As each I/O operation is completed, the interrupt servicing routine posts the CCB as complete, decrements the I/O counter in the task control block and clears the wait bit so that control can be returned by the program switching routine. When a task being awaited completes, the waiting task is reactivated and control is returned to the point of interruption (immediately following the WAIT macro instruction that resulted in the delay).

Examples:

| 1 | LABEL | Δ OPERATION Δ | OPERAND | Δ |
|---|-------|---------------|---------------|---|
| | | 10 | 16 | |
| | | WAIT | PRCCB1, BRERR | |
| | | WAIT | PRCCB2 | |
| | | WAIT | (1), (15) | |
| | | WAIT | ALL | |

4.3.2. Multiple I/O Wait (WAITM)

Function:

The WAITM macro instruction temporarily suspends program execution until any one of several I/O operations specified by the instruction is completed. Upon completion of one of the I/O operations, control is returned to the program at the point immediately following the WAITM macro instruction, with register 1 containing the address of the CCB associated with the I/O operation. The appropriate wait indicators will be cleared with regard to the unfinished I/O operation.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|--|
| [symbol] | WAITM | { CCB-name-1, CCB-name-2 [, ..., CCB-name-n] } { list-name } (1) |

Positional Parameter 1:

CCB-name-1,CCB-name-2,[...,CCB-name-n]

Specifies the symbolic addresses of the CCBs to be tested that are associated with the I/O operations to be awaited. At least two CCBs must be specified.

list-name

This is a single entry which specifies the symbolic address of a list containing full-word addresses of CCBs associated with the I/O operations to be awaited. The byte following the last full word must be nonzero to indicate end of table.

(1)

Indicates that register 1 has been preloaded with the address of the list of CCB addresses.

NOTE:

The WAITM macro instruction also may specify a combination of CCB and ECB addresses as parameters. See also the multiple task wait macro instruction described in 7.4.3.

When this macro instruction is executed, each referenced CCB is marked as being awaited. Upon completion of a marked CCB, the waiting task is activated and the remaining CCBs that are marked as being awaited are cleared.

The WAITM macro instruction always requires more than one event to be tested. If only one event is to be tested, use the WAIT macro instruction.

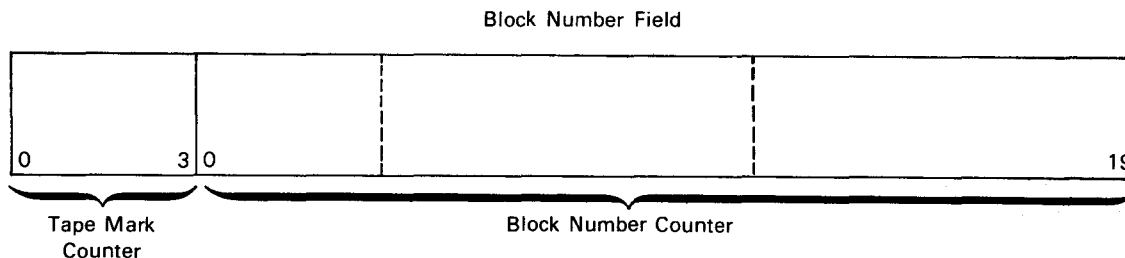
4.4. BLOCK NUMBERED TAPE FILES

OS/3 can process magnetic tapes with or without block numbers. The use of block numbers reduces the possibility of incorrect tape positioning and therefore incorrect tape processing. This is especially helpful for error recovery on read and write commands and in restarting at a checkpoint.

4.4.1. Block Number Field

When the block numbering capability is being used, all blocks on tape except tape marks will include a 3-byte block number field as the first three bytes of the block. This 24-bit block number field consists of a 4-bit tape mark counter and a 20-bit block number counter. The block number counter reflects the number of blocks (including tape marks) written; the tape mark counter reflects the number of contiguous preceding tape marks since the last data block. The format of the block number field is shown in Figure 4-9.





| Field | Bit | Description |
|----------------------|------|---|
| Tape mark counter | 0 | 0 = Preceding block is not a tape mark 1 = Preceding block is a tape mark |
| | 1-3 | All zeros if bit 0 = 0 0 to 7 ₁₆ if bit 0 = 1. Number of contiguous preceding tape marks (modulo 8) |
| Block number counter | 0-19 | Tape block number in binary |

Figure 4-9. Tape Block Number Field Format

The first block on tape that is not a tape mark will contain a block number field with a block count of 1 plus the number of tape marks that preceded it. This block number count is incremented by 1 for each block and tape mark on the tape. If a volume contains more than one file, the count is continued from the preceding file on the volume and the blocks are consecutively numbered to the end of the tape.

All label, data, and checkpoint blocks are counted and numbered; tape marks do not contain a block number field and are counted only. When a tape mark is written, the block number is incremented but is not written. When a tape mark is read, the count is merely incremented because there is no number to check.

During input, the 3-byte actual block number (including the tape mark counter) in the physical block is checked against the expected block number. If there is a mismatch, appropriate error recovery is performed. Use of the tape mark counter in the block number field will have no effect on block numbered tapes that were written without the tape mark counter by an earlier supervisor. On input tapes, the first data block after a tape mark will indicate whether or not the tape contains a tape mark counter in the block number field. If there is a tape mark counter in the block number field of this block, it is assumed there are tape mark counters throughout the entire tape and these will be checked. If there is no tape mark counter in this block, it is assumed there are none throughout the entire tape and this field (bits 0-3) will not be checked. On output tapes, all block numbered tapes will be written with a tape mark counter as part of the block number. The counter will contain a nonzero value only in the first data block following one or more tape marks.

4.4.2. Tape Restrictions

The 3-byte block number fields are added to standard labels on block numbered tapes. The three bytes precede the label identifier (VOL1, HDR1, etc.) making the label 83 bytes long. This is true for tapes written in ASCII as well as EBCDIC. Note that in the case of ASCII tapes, the 83-byte label is nonstandard. It can be used for internal processing, but cannot be used for information interchange. Block number processing will be exactly the same for both EBCDIC and ASCII tape files. Tape label formats for block numbered EBCDIC tapes are shown in Figures 6-17 through 6-21.

Block numbers will be volume dependent and file independent. Files on a volume and volumes in a multivolume file must be all numbered or all unnumbered, not mixed.

Block number processing is available for magnetic tapes on selector or multiplexer channels. These may be 9-track tapes, or 7-track odd parity tapes operating in data conversion mode. Block size of 7-track tapes operating in data conversion mode must be a multiple of 3.

4.4.3. Input/Output Buffer

When processing block numbered tapes you must reserve a 4-byte storage area immediately preceding your input/output area for supervisor processing of the block number. This 4-byte block number area, and the input/output area, must be aligned on a full-word boundary. Do not include these four bytes in either the location or the block size of the input/output area.

Block numbers will be checked when reading in either direction. When reading backward, you must be sure your input/output area is large enough to hold the entire block of data. If the data is truncated on a backward read, the block number will be lost and incorrect positioning of the tape may result.

4.4.4. Processing

A number of software components are affected by block number processing; these include system generation, tape preparation, job control, automatic volume recognition, physical IOCS, data management, and system access technique (SAT) on tape files. Several control tables in main storage are also affected, including the systems information block, the device PUB trailer, and the CCB. These tables contain fields that are updated and bits that are set, tested, and cleared to reflect user options and processing events.

Physical IOCS will perform block number processing for data management, tape SAT, and EXCP-level physical IOCS users. A general description of required and optional parameters and processing performed is contained on the following pages. Details pertinent to physical IOCS users are contained in 4.4.5. Details of the requirement for tape SAT are contained in 6.5 to 6.10 of this manual. For data management details, refer to the data management user guide, UP-8068 (current version).

The supervisor must be configured to process block numbered tapes, in which case, the generated supervisor can process both numbered and unnumbered tapes. A bit in the SIB is set to indicate that the supervisor supports block numbering. If the supervisor does not have the block numbering capability, only unnumbered tapes can be processed; otherwise, misalignment and possible truncation of data will result because of the block number field.

To use the block numbering capability of the supervisor, you must also reserve a 4-byte storage area, aligned on a full-word boundary, immediately preceding the input/output area. If you are a data management user, you indicate that you have reserved this 4-byte area by using the BKNO=YES parameter in the DTFMT macro instruction. If you are a tape SAT user, you do this by using the BKNO=YES parameter in the TCA macro instruction. If you are a physical IOCS user at the EXCP level, you must also indicate that you have reserved the 4-byte area by setting a bit in the CCB (4.4.5).

You have the option not to use block number processing even though the supervisor has the capability and you have indicated there is a block number field preceding the input/output area. If you enter N as the first parameter in the VOL job control statement, block numbers will not be written on output tapes, and will be ignored if present on input tapes.

Automatic volume recognition will read and store volume serial numbers and will set appropriate bits in the PUB trailer to indicate whether or not it is processing standard labeled tapes and block numbered tapes.

The PUB trailer for a block numbered tape file will contain an expected block number. This number will reflect the next block number anticipated in a forward read and will be adjusted accordingly for backward reads. When the tape is read in either direction, the block number read from tape is stored in the PUB trailer and compared with the expected block number. If there is no discrepancy (and no other errors), control is returned to the user program. If there is a discrepancy, physical IOCS attempts to find the correct block by moving the tape backward or forward the number of blocks implied by the discrepancy. If the correct block is found, control is returned to the user. If the correct block cannot be found, the tape is left positioned where it was on the last attempt and an error message is sent to the console.

When processing control macros, block number processing will not be performed, because no data transfer is involved. However, for commands involving single blocks (FSB, BSB), the block number count will be updated.

On block numbered tapes, CCW chains with more than one tape movement command and multiblock BCW commands can be processed only through the first tape movement command.

4.4.5. Physical IOCS Requirements and Options

Physical IOCS users at the EXCP level have an additional requirement. Before issuing any EXCP macro instruction for a block numbered tape, you must set byte 3, bit 7, in the CCB. This indicates that the 4-byte block number field preceding the input/output buffer has been reserved. If this bit is not set, the job will be cancelled.

You can request that block numbering be ignored on input tapes by setting byte 0, bit 3, in the CCB before issuing an EXCP. In this case, block numbered tapes will be read, but the block numbers will not be verified. You must set this bit each time you want to ignore block number processing on a read. ↓

Another option available at the physical IOCS level is to accept unrecoverable errors. You can do this by setting byte 3, bit 1, in the CCB. You don't have to reset this bit for each EXCP; it need only be set once and stays set.

On a read, if physical IOCS detects a variance between the expected block number and the actual block number and is unable to resolve this variance after ten retries, a console message is issued. If byte 3, bit 1 (accept unrecoverable errors) is set, the console message gives the operator opportunity to request a retry or accept the error. If retry is requested but is still unsuccessful, the operator will again be asked to request a retry or accept the error. If he accepts the error (or if he first requests retry and it is still unsuccessful), physical IOCS sets byte 2, bit 5 (block numbers not equal), and byte 2, bit 1 (unrecoverable error). Physical IOCS then sets byte 2, bit 0, to indicate that CCB processing is complete and returns control to your program. On input, you should test byte 2, bit 5, after the WAIT is executed to ensure that the correct block has been processed.

If byte 3, bit 1 (accept unrecoverable errors) was not set, the operator has the option only to request a retry or cancel. If retry is requested but is still unsuccessful, the operator will again be asked to request a retry or cancel.

On a write, if byte 3, bit 1, is set and the tape cannot be positioned correctly, a console message gives the operator the opportunity to accept the error or cancel. If this bit is not set, he must cancel. ↑



5. Disk Space Management

5.1. GENERAL

Space management comprises a group of routines that provide you an efficient and completely automatic disk and diskette space accounting capability. These routines relieve you of the responsibility of knowing the precise contents of disk and diskette volumes. These routines also resolve competing demands for space allocation and establish standard interfaces with your other programs as well as job control, utility, and service programs.

Using job control statements in your job stream at run time, you can enter the information required by these routines as parameters in your job control statements. For example, within your device assignment set you can use the EXT job control statement to allocate space to a new disk or diskette file, or to extend a disk file. You can use the SCR statement to deallocate (scratch) a disk or diskette file.

These functions can also be requested within your program. The following macro instructions are available:

- ALLOC
Allocates files.
- EXTEND
Extends files already allocated (disk only).
- SCRATCH
Scratches files that are no longer needed.
- RENAME
Renames files (disk only).
- OBTAIN
Retrieves label and extent information.

↓
For disk file processing, you may use all five of these macro instructions; for diskette file processing, you may use only ALLOC, SCRTCH, and OBTAIN. Diskette space management macro instructions are a compatible subset of the disk space management macro instructions. Although the formats are the same, there are some differences due to the different labelling patterns and physical characteristics. For convenience, disk and diskette macro instructions are described separately in this manual.

↑ 5.2. DISK SPACE MANAGEMENT ROUTINES

→ The disk space management routines are transient service routines. Allocation is accomplished by maintaining the volume table of contents (VTOC) through standard procedure for all files: system, temporary, and those considered permanent by the user. The VTOC is a permanently allocated, unmovable file which exists on every disk volume. The VTOC is addressed by the standard volume label and is included in a disk volume by the disk volume initialization program. The VTOC file comprises a control block, or set of control blocks, for each file on the volume and for all unused space on the volume. Refer to the data management user guide, UP-8068 (current version) for the formats and description of the VTOC and disk file labels.

The disk space management routines maintain the VTOC by creating control blocks for new files and deleting control blocks for files removed from the volume. When a file is to be created, unused space is found for it by searching the appropriate blocks in the VTOC, allocating the space as the extents of the file, and removing the amount from free space. When a file is deleted, the control block for the file is removed from the VTOC; the extents previously assigned to the file are again available for allocation.

Both disk space management and the system access technique (SAT) allocate file space according to the characteristics of the device. Also, in the case of the 8418 Mod I disk subsystem, allocated file space is based on 400 cylinders; while on the 8418 Mod II disk subsystem, allocated file space is based on 800 cylinders.

5.2.1. Allocate Routine

The allocate routine assigns space to a new file or an existing file. After the validity of the request is ensured, the allocate routine locates space on the disk by using a format 5 label in the VTOC that satisfies the request. The routine then removes the definition of the available space from the format 5 label and assigns it to the requesting file. If the requesting file is new, a format 1 label and format 2 label, along with any needed format 3 labels, are created and placed in the VTOC.

For a new split cylinder file, the allocation of the primary member results in the creation of a format 1 label, a format 2 label, a format 3 label (all physical extent information for a split file is kept in a common format 3 label), a format 6 label (all internal bookkeeping for a split file; e.g., available heads and format 1 pointers, are kept in a common format 6 label). Allocation of subsequent members to the split file results in only the creation of format 1 and format 2 labels. Note that because all physical extent information about a split file is kept in a common format 3 label, the file limit is 13 extents (the limit is 16 for a nonsplit cylinder file). Note also that the acquisition of more space for an already existing split file, via the ALLOC macro instruction with the OLD parameter, must be requested for the primary member of the file.

The services of the allocation routine are requested through the ALLOC macro instruction. Allocation is performed on a volume-by-volume basis; the inputs to the allocate routine for each file are the file control blocks (FCBs) of the file, or 8-byte file name. In the case of multivolume files, the volume sequence number must be set in register O by your calling program. The FCB must include a pointer to the extent request block (ERB); in a multivolume file, there is one ERB for each volume. The first FCB has 22 ERB pointers. When more than 22 volumes are allocated to this file, the first FCB has a next FCB pointer.

Special considerations are given to allocations for run libraries which are created by job control for each job. The FCB and accompanying ERB are in main storage with the pointer to the ERB filled with zeros. The disk space management then assumes that the ERB immediately follows the FCB in storage.

5.2.2. Extend Routine

The extend routine assigns additional space to a file after that file's initial space allocation has been exhausted. This secondary allocation (or extension) is handled automatically for data management by the system access technique (SAT), which provides a common interface to all disk subsystem types. When split files are extended, all files which belong to the same group are also extended. Any member of the split file, primary or subsequent, may be dynamically extended. The dynamic extension of a split file results in a new physical extent entry in the appropriate format 3 label. To avoid exceeding the limit of 13 extent entries, a secondary allocation increment greater than 1 should be considered for heavily used (and extended) split files.

The request for extension is made through the EXTEND macro instruction. When the file exhausts its initial allocation of space, data management calls the extend transient routine and the space (if available) is allocated in amounts specified by the secondary allocation of the format 1 label of that file.

The extend routine always tries to assign space contiguous to the last space assigned to a file. This minimizes the number of separate extents required. If this attempt fails or if insufficient space is available, then space is assigned by first fit. For example, if four cylinders are requested, the space is assigned from the first format 5 extent encountered that has at least four cylinders.

In the event that there is not sufficient space available to satisfy the extension increment specified in the format 1 label with contiguous space, disk space management will allocate the largest extent possible. If more space is required after a partial extension, additional extent requests can be made, as required.

5.2.3. Scratch Routine

The scratch routine deallocates disk space from a file and makes it available for future use. Scratch, after ensuring that the request is valid, removes the extents from format 1 or format 3, and records and updates format 5 records. The extent freed is placed in the correct position of the format 5 records, which are in ascending sequence. Format 1, 2, and 3 records are deleted from the VTOC and replaced with format 0 records if a file is scratched.

Scratching a member file of a split cylinder file results in the deletion of the corresponding format 1 and format 2 labels. Because the physical space allocated to the file is common to all its members, the space is returned to free space only when the last member of the file is scratched. At that point, the format 5 and format 6 labels are also deleted from the VTOC. Note that scratching the primary member of the split file will prevent the acquisition of new space for the file via ALLOC OLD.

There are three basic scratch routines available:

- scratch entire file;
- scratch files by prefix; and
- scratch all expired files.

5.2.3.1. Scratch Entire File

The VTOC is searched for a format 1 label that matches the 44-byte physical ID retrieved from the FCB. That entire file is then scratched.

5.2.3.2. Scratch by Prefix

The ability to scratch certain files which have identification fields beginning with the first four characters specified in a SCRTCH macro instruction parameter also is available. This allows you to deallocate temporary work files by a single call on space management routines. The characters \$Y\$ cannot be included in this prefix; this prevents you from scratching system files by mistake.

5.2.3.3. Scratch All by Date

This function permits you to scratch all files which have passed an expiration date that was included in the format 1 label expiration date field. The expiration date is compared to the date supplied by you; each format 1 label expiration date field is compared to this date and if it has expired, the file is scratched and that space is now available for reallocation.

5.2.4. Rename Routine

The rename routine allows you to apply a new file ID to a format 1 label. You can rename any file except a system scratch file (\$SCR). The RENAME macro instruction initiates the rename routine.

5.2.5. Obtain Routine

The obtain routine allows you to access any block of the VTOC. The obtain routine is initiated by using the OBTAIN macro instruction; you have to provide the 8-byte file name, the absolute disk address of the block to be accessed, or have the FCB in main storage.

5.3. DISK MACRO INSTRUCTIONS

The imperative macro instructions available to you with disk space management are: ALLOC, EXTEND, SCRTCH, RENAME, and OBTAIN. The ALLOC, EXTEND, and SCRTCH macro instructions are concerned with basic space accounting (allocating, extending, and deleting file space), while the RENAME and OBTAIN macro instructions provide support operations to allow you to change file ID and to access certain VTOC information. The following paragraphs give you a detailed description of each macro instruction.

5.3.1. Assign Space to a New Disk File or to an Existing Disk File (ALLOC)

Function:

The ALLOC macro instruction assigns space to a new file or to an existing file. Allocation is performed on a volume-by-volume basis. For each volume of a file, the inputs to the allocate routine are the file control block (FCB) of the file and the corresponding extent request block (ERB), if required.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|--|
| [symbol] | ALLOC | $\left\{ \begin{array}{l} \text{FCB-name} \\ \text{filename-addr} \\ (1) \end{array} \right\} \left[\left\{ \begin{array}{l} \text{error-addr} \\ (r) \end{array} \right\} \right]$ $\left[\left\{ \begin{array}{l} \text{vol-seq-no,OLD,NOFCB} \\ (0) \end{array} \right\} \right]$ |

Positional Parameter 1:

FCB-name

Specifies the symbolic address of the file control block.

filename-addr

Specifies the symbolic address of an 8-byte area in main storage in which you have stored the file name (as listed on the LFD job control card of the file). NOFCB must be entered as positional parameter 5.

(1)

Indicates that register 1 has been preloaded with the address of the file control block, or the address of the file name if NOFCB has been entered as positional parameter 5.

Positional Parameter 2:

error-addr

Specifies the symbolic address to which control is transferred if an error is encountered.

(r) Indicates that a register (other than 0 or 1) has been preloaded with the error address.

If omitted, the calling task will be abnormally terminated if an error occurs.

Positional Parameter 3:

vol-seq-no

Specifies the volume number of a multivolume file to be allocated.

If omitted, the value 1 is assumed.

Positional Parameter 4:

OLD

Specifies that an old file is extended with information contained in the specified FCB and ERB rather than with the EXTEND macro instruction.

If omitted, a new file is assumed.

Positional Parameter 5:

NOFCB

Specifies that positional parameter 1 refers to a file name instead of an FCB. In this case, space management will issue an RDFCB macro instruction to read the FCB from the run library into the transient area.

If omitted, it is assumed that positional parameter 1 refers to an FCB and that you have issued an RDFCB macro instruction for this file.

(0)

Indicates that register 0 has been preloaded with the information for positional parameters 3, 4, and 5:

Bit

22 1 = NOFCB

23 1 = OLD (See positional parameter 4.)

24—31 Volume sequence number.

Examples:

| 1 LABEL | 10 △OPERATION△ | 16 OPERAND | △ |
|------------|-------------------|---------------|---|
| | ALLOC | FCB1,(3) | |
| | ALLOC | (1),(3),2,OLD | |

5.3.2. Assign Additional Space to an Existing Disk File (EXTEND)

Function:

The EXTEND macro instruction allows you to assign additional space to a Sequential Access Method (SAM) or a System Access Technique (SAT) file after its initial allocation of space has been exhausted. The extend routine is called by data management, or any user, only after the file runs out of space; the additional space, if available, is allocated in increments specified at the time of primary allocation.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|---|
| [symbol] | EXTEND | $\left\{ \begin{array}{l} \text{FCB-name} \\ \text{filename-addr} \\ (1) \end{array} \right\} \left[, \left\{ \begin{array}{l} \text{error-addr} \\ (r) \end{array} \right\} \right]$ $\left[, \left\{ \begin{array}{l} \{01\} \\ \{80\} \end{array} \right\} , \left\{ \begin{array}{l} \text{vol-seq-no} \\ 1 \\ (0) \end{array} \right\} , [\text{FCBCORE}] \right]$ |

Positional Parameter 1:

FCB-name

Specifies the symbolic address of the file control block.

filename-addr

Specifies the symbolic address of an 8-byte area in main storage in which you have stored the file name (as listed on the LFD job control card) of the file to be extended.

(1)

If FCBCORE is entered as positional parameter 5 or if bit 6 of register 0 is set to 1, indicates that register 1 has been preloaded with the address of the file control block.

If positional parameter 5 is omitted, indicates that register 1 has been preloaded with the address of the file name.

Positional Parameter 2:

error-addr

Specifies the symbolic address to which control is transferred if an error is encountered.

(r)

Indicates that a register (other than 0 or 1) has been preloaded with the error address.

If omitted, the calling task will be abnormally terminated if an error occurs.

Positional Parameter 3:

- 01 Specifies that the file is a SAM file.
- 80 Specifies that the file is a SAT file.

Positional Parameter 4:

- vol-seq-no**
Specifies the volume number of a multivolume file to be extended.
- If omitted, the value 1 is assumed.

Positional Parameter 5:

FCBCORE

Specifies that positional parameter 1 refers to the address of an FCB. This assumes you have issued an RDFCB macro instruction for this file.

If omitted, it is assumed that positional parameter 1 refers to the address of a file name. In this case, space management will issue an RDFCB macro instruction to read the FCB from the run library into the transient area.

- (0) If filename-addr was specified as positional parameter 1, indicates that register 0 has been preloaded with the following information:

| <u>Bit</u> | |
|------------|------------------------|
| 16—23 | File type |
| 24—31 | Volume sequence number |

If FCB-name was specified as positional parameter 1, indicates that register 0 has been preloaded with the following information:

| <u>Bit</u> | |
|------------|------------------------|
| 6 | 1 = FCBCORE |
| 16—23 | File type |
| 24—31 | Volume sequence number |

This is an alternative to entering FCBCORE as positional parameter 5, and assumes that you have issued an RDFCB macro instruction for this file.

Examples:

| 1 | LABEL | Δ OPERATION Δ | OPERAND | Δ |
|---|-------|---------------|-----------------------|---|
| | | 10 | 16 | |
| | | EXTEND | (1), (15), (0) | |
| | | EXTEND | FCBINME, , 2, FCBCORE | |

5.3.3. Scratch a Disk File (SCRATCH)

Function:

The SCRATCH macro instruction allows you to deallocate one or more files, identified by the 44-byte file ID, and make that space available for future use. Do not issue the SCRATCH macro instruction to a file that is currently open.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|---|
| [symbol] | SCRATCH | { FCB-name } [{ PREFIX }] [{ error-addr }] (1) [ALL] [(0)] [(r)] |

Positional Parameter 1:

FCB-name

Specifies the symbolic address of the file control block (FCB) in main storage.

(1)

Indicates that register 1 has been preloaded with the address of the FCB in main storage.

Positional Parameter 2:

ALL

Specifies that all files whose expiration date has been exceeded are to be deallocated. The expiration date must be included in the 3-byte expiration date field of the FCB.

PREFIX

Specifies that all files that have the specified 4-byte prefix are to be deallocated. The 4-byte prefix must be placed in bytes 76—79 of the FCB.

(0) Indicates that register 0 has been preloaded with the following information:

| | |
|-------------|----------------------------|
| <u>Bit</u> | |
| 0—7 | Hexadecimal function code: |
| | |
| <u>Code</u> | <u>Interpretation</u> |
| 00 | Scratch file. |
| 82 | Scratch all by date. |
| 83 | Scratch by prefix. |

If omitted, the file specified by the 44-byte file ID in the FCB is scratched.

Positional Parameter 3:

error-addr

Specifies the symbolic address that receives control if an error is encountered.

(r)

Indicates that a register (other than 0 or 1) has been preloaded with the address of the error routine.

If omitted, the calling task is abnormally terminated if an error occurs.

Examples:

| 1 | LABEL | Δ OPERATION Δ 10 16 | OPERAND | Δ |
|---|-------|------------------------|---------------------|---|
| | | SCRATCH | (1), (0), (15) | |
| | | SCRATCH | FCBNAME, ALL, ERROR | |

5.3.4. Rename a Disk File (RENAME)

Function:

The RENAME macro instruction permits you to assign a new physical file name to any file except a system scratch file. This is accomplished by specifying the new name to be used and the file ID as contained in the format 1 label. Do not issue the RENAME macro instruction to a file that is currently open.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|--|
| [symbol] | RENAME | { param-list } (1) [{ error-addr } (r)] [{ vol-seq-no } 1] |

Positional Parameter 1:

param-list

Specifies the symbolic address of a parameter list containing the 8-byte file name (as listed on the LFD job control card) and a new 44-byte file identifier.

(1)

Indicates that register 1 has been preloaded with the address of the parameter list.

Positional Parameter 2:

error-addr

Specifies the symbolic address to which control is transferred if an error is encountered.

(r)

Indicates that a register (other than 0 or 1) has been preloaded with the error address.

If omitted, the calling task will be abnormally terminated if an error occurs.

Positional Parameter 3:

vol-seq-no

Specifies the volume number of a multivolume file to be renamed.

If omitted, the value 1 is assumed.

Examples:

| 1 | LABEL | △ OPERATION △ | OPERAND | △ |
|---|-------|---------------|-----------|---|
| | | 10 | 16 | |
| | | RENAME | (1), (12) | |
| | | RENAME | PLIST, 2 | |

5.3.5. Access VTOC User Block (OBTAIN)

Function:

The OBTAIN macro instruction allows you to access any user block in the VTOC. You must first construct the parameter list which specifies the file, the particular area of the VTOC that is of interest to you, and the address of a buffer in main storage where you want the retrieval data stored.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|---|
| [symbol] | OBTAIN | { param-list } [, { error-addr }] [, { vol-seq-no }] (1) (r) 1 [,FCBCORE] |

Positional Parameter 1:

param-list

Specifies the symbolic address of a parameter list containing the following:

Bytes 0—7

An 8-byte file name (as listed on the LFD job control card).

Byte 8

Function code of the requested service for the disk pack containing the volume sequence number specified by positional parameter 3:

| <u>Code</u> | <u>Interpretation</u> |
|-------------|--|
| 00 | VOL1 address in form Occchrr |
| 01 | Format 1 address in form Occchrr |
| 02 | Format 2 address in form Occchrr |
| 03 | Format 3 address in form Occchrr |
| 04 | Format 4 address in form Occchrr |
| 05 | Format 5 address in form Occchrr |
| 06 | Format 6 address in form Occchrr |
| 80 | Contents of VOL1 label |
| 81 | Contents of format 1 label |
| 82 | Contents of format 2 label |
| 83 | Contents of format 3 label |
| 84 | Contents of format 4 label |
| 85 | Contents of format 5 label |
| 86 | Contents of format 6 label |
| 87 | Contents of label record located at the disk address which is in the first word of the buffer in the form Occchrr. |

NOTE:

Addresses in the form Occchrr are in discontinuous binary, where ccc is the cylinder number, hh is the head number, and rr is the record number.

Bytes 9—11

Buffer address of the storage area into which the addresses or label contents requested through byte eight are loaded. For codes 00 through 06, the first word of the buffer contains the disk address of the label record. For code 87, you must store the disk address (in the form Occchrrr) of the label desired in bytes 0 through 3 of this buffer area.

Bytes 12—15

Symbolic address of the FCB in main storage. This field is required only if positional parameter 4 is specified.

(1)

Indicates that register 1 has been preloaded with the address of the parameter list.

Positional Parameter 2:

error-addr

Specifies the symbolic address to which control is transferred if an error is encountered.

(r)

Indicates that a register (other than 0 or 1) has been preloaded with the error address.

If omitted, the calling task will be abnormally terminated if an error occurs.

Positional Parameter 3:

vol-seq-no

Specifies the volume number of a multivolume file from which you retrieve the VTOC information.

If omitted, a value of 1 is assumed.

Positional Parameter 4:

FCBCORE

Specifies that the FCB is in main storage. The address of the FCB is contained within bytes 12—15 of the parameter list whose address is specified by positional parameter 1.

If omitted, space management reads the FCB from disk, using the 8-byte file name contained in the parameter list.

Examples:

| 1 | LABEL | △OPERATION△ 10 16 | OPERAND | △ |
|---|-------|----------------------|-----------------|---|
| | | OBTAIN | (1),(4),2 | |
| | | OBTAIN | RECOVER1,ERRRTN | |

5.4. DISKETTE SPACE MANAGEMENT ROUTINES

The diskette space management routines are transient service routines. Space management is accomplished by maintaining information on the index track about the volume and files on the diskette. The index track on physical track 0 has a fixed format. This track is divided into 26 sectors with each sector 128 bytes long. Sectors 1 through 6 are reserved for physical information. Sector 7 is referred to as the volume label and is used to describe the diskette volume. Sectors 8 through 26 are referred to as file labels and define the files recorded on cylinders 1 through 74 of the diskette. Refer to the data management user guide, UP-8068 (current version) for the format and description of the diskette file labels.

5.5. DISKETTE MACRO INSTRUCTIONS

Of the five macro instructions available for space management, only ALLOC, SCRTCH, and OBTAIN may be used for diskette space management. Also, you will note some variations in the parameter specifications. For example, the fourth parameter of the ALLOC macro instruction and the second parameter of the SCRTCH macro instruction are not used, and you can choose from only two types of label information in the first parameter of the OBTAIN macro instruction.

5.5.1. Assign Space to a New Diskette File (ALLOC)

Function:

The ALLOC macro instruction assigns space on the diskette to a new file. The increments of allocation are sectors (128 bytes per sector). After ensuring the request is valid, the allocate routine locates space on the diskette by reading the index track and calculates the available space on the volume from user file labels 8 through 26. The routine then writes a new file label on the index track to allocate space to the file.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|--|
| [symbol] | ALLOC | $\left\{ \begin{array}{l} \text{FCB-name} \\ \text{filename-addr} \\ (1) \end{array} \right\} \left[, \left\{ \begin{array}{l} \text{error-addr} \\ (r) \end{array} \right\} \right]$ $\left[, \left\{ \begin{array}{l} \text{vol-seq-no} \\ (0) \end{array} \right\} \text{,, NOFCB} \right]$ |

Positional Parameter 1:

FCB-name

Specifies the symbolic address of the file control block.

filename-addr

Specifies the symbolic address of an 8-byte area in main storage in which you have stored the file name (as listed on the LFD job control card of the file). NOFCB must be entered as positional parameter 5.

(1)

Indicates that register 1 has been preloaded with the address of the file control block, or the address of the file name if NOFCB has been entered as positional parameter 5.

Positional Parameter 2:

error-addr

Specifies the symbolic address to which control is transferred if an error is encountered.

(r)

Indicates that a register (other than 0 or 1) has been preloaded with the error address.

If omitted, the calling task will be abnormally terminated if an error occurs.

Positional Parameter 3:

vol-seq-no

Specifies the volume number of a multivolume file to be allocated.

If omitted, the value 1 is assumed.

Positional Parameter 4:

This parameter is not applicable, but a comma must be entered in this position if positional parameter 5 is used.

Positional Parameter 5:

NOFCB

Specifies that positional parameter 1 refers to a file name instead of an FCB. In this case, space management will issue an RDFCB macro instruction to read the FCB from the run library into the transient area.

If omitted, it is assumed that positional parameter 1 refers to an FCB and that you have issued an RDFCB macro instruction for this file.

(0)

Indicates that register 0 has been preloaded with the information for positional parameters 3, 4, and 5:





- Bit
- 22 1 = NOFCB
- 23 0 = new allocation
- 24—31 Volume sequence number

5.5.2. Scratch a Diskette File (SCRATCH)

Function:

The SCRATCH macro instruction deallocates diskette space for a file and makes it available for future use. After ensuring that the request is valid, the scratch routine searches the file labels for a file identifier (17 bytes) that matches the first 17 bytes of the 44-byte file ID retrieved from the FCB. If a match occurs, the file's extent is scratched by marking the file label 'deleted'. Do not issue the SCRATCH macro instruction to a file that is currently open.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|--|
| [symbol] | SCRATCH | $\left\{ \begin{array}{c} \text{FCB-name} \\ (1) \end{array} \right\} [,] \left[, \left\{ \begin{array}{c} \text{error-addr} \\ (r) \end{array} \right\} \right]$ |

Positional Parameter 1:

FCB-name

Specifies the symbolic address of the file control block (FCB) in main storage.

(1)

Indicates that register 1 has been preloaded with the address of the FCB in main storage.

Positional Parameter 2:

This parameter is not applicable, but a comma must be entered in this position if positional parameter 3 is used.

Positional Parameter 3:

error-addr

Specifies the symbolic address that receives control if an error is encountered.



(r)

Indicates that a register (other than 0 or 1) has been preloaded with the address of the error routine.

If omitted, the calling task is abnormally terminated if an error occurs.

5.5.3. Obtain Diskette Label Information (OBTAIN)

Function:

The OBTAIN macro instruction retrieves the volume label or any file label on the index track. After ensuring that the request is valid, the obtain routine locates the requested label and returns it in a buffer area in main storage. You must construct a parameter list which specifies the type of label requested and gives the address of your buffer.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|---|
| [symbol] | OBTAIN | { param-list } [, { error-addr }] [, { vol-seq-no }] (1) (r) 1 [,FCBCORE] |

Positional Parameter 1:

param-list

Specifies the symbolic address of a parameter list containing the following:

Bytes 0—7

An 8-byte file name (as listed on the LFD job control card).

Byte 8

Function code specifying the type of label requested.

| Code | Interpretation |
|------|----------------|
|------|----------------|

| | |
|----|---------------------------------|
| 80 | Contents of index track label 7 |
|----|---------------------------------|

| | |
|----|--|
| 81 | Contents of index track label for the file name specified in bytes 0—7 |
|----|--|

Bytes 9—11

Buffer address of the storage area into which the label contents are to be loaded. This buffer must be at least 128 bytes.

Bytes 12—15

Symbolic address of the FCB in main storage. This field is required only if positional parameter 4 is specified.



(1)

Indicates that register 1 has been preloaded with the address of the parameter list.

Positional Parameter 2:

error-addr

Specifies the symbolic address to which control is transferred if an error is encountered.

(r)

Indicates that a register (other than 0 or 1) has been preloaded with the error address.

If omitted, the calling task will be abnormally terminated if an error occurs.

Positional Parameter 3:

vol-seq-no

Specifies the volume number of a multivolume file.

If omitted, a value of 1 is assumed.

Positional Parameter 4:

FCBCORE

Specifies that the FCB is in main storage. The address of the FCB is contained within bytes 12—15 of the parameter list whose address is specified by positional parameter 1.

If omitted, space management reads the FCB from disk, using the 8-byte file name contained in the parameter list.



5.6. SPACE MANAGEMENT ERROR CODES

Errors that occur during processing of your disk and diskette space management macro instructions cause a transient routine to be called into main storage. This error transient overlay routine places an appropriate error code into register 0, depending upon the type of error. If the error is not catastrophic (one that necessitates termination of your program), control is then switched to your error-handling routine (through the error-addr parameter of your macro instructions). If you do not include an error handling routine in your program, your task is terminated and control is returned to the supervisor.

The system messages programmer reference, UP-8076 (current version) contains a list of space management error codes and their interpretation.

6. System Access Technique

6.1. GENERAL

The OS/3 includes several data management packages that allow you to process a wide variety of file types in several different ways. System access technique (SAT) is a specialized block level device handler that provides great efficiency in handling disk and tape files.

This section provides you with a brief functional description of SAT operation techniques, and an explanation of the interface that is available to modify SAT operation or to construct your own handler modules.

SAT techniques and macro instructions to define and control disk files are described starting at 6.2. SAT techniques and macro instructions to define and control tape files are described starting at 6.5.

6.2. DISK SAT FILE ORGANIZATION AND ADDRESSING METHODS

SAT files may be segmented into logical parts called partitions with each partition having distinct physical and logical characteristics. Each partition is defined by a PCA macro instruction, which generates a partition control appendage to the DTF file table. Up to seven partitions may be defined within a single file.

6.2.1. PCA Table Entries Used in Addressing

The addressing of physical blocks being accessed from a partition is controlled by two entries in the partition control appendage (PCA) table in main storage. A PCA table (Figure 6-1) is created for each partition processed and is used as a reference by the program. The two entries in the PCA table that affect addressing are:

- Current ID
- End of data ID

The current ID is the starting address of the logical partition or the address of the current block being processed.

The end of data ID is the last logical block of the partition.

When you open your file with the OPEN macro instruction, the current ID and end of data ID for each partition in the file referenced are initialized to the start and end of that partition. When sequential processing (SEQ keyword parameter) is performed, successful completion of the GET and PUT macro instructions results in the current ID being incremented to the next physical block of the partition. This incrementation, which occurs after the wait, continues until the end of data ID is encountered; this indicates that all blocks in a file have been processed.

Provisions are also made to allow you to access blocks in other than sequential method. The current ID is the same address as the label of your PCA (partition). This is a 4-byte field containing a right-justified hexadecimal number representing the block to be referenced relative to the first block of the partition.

When first initialized, this field contains a 1 corresponding to the first block of the partition. If you wish to access a particular individual block, you must load the relative block number into the PCA address; this causes the current ID to reflect the block you want to access.

| | | | | |
|------|------------------------|----------------|---------------|---------------|
| Byte | 0 | 1 | 2 | 3 |
| 0 | current ID | | | |
| 4 | max relative block | | | |
| 8 | logical blocks/track | | | |
| 12 | PCA ID | EOD ID | | |
| 16 | I/O count | IOAREA/address | | |
| 20 | block size | | reserved | sectors/block |
| 24 | lace factor/key length | | unit of store | |
| 28 | DTF address | | | |
| 32 | PCA flags | EOD address | | |

PCA FLAGS

| | | | |
|------------|--------------|------------|------------------------------------|
| <u>Bit</u> | | <u>Bit</u> | |
| 0 | Format write | 4 | Verify required/initial allocation |
| 1 | Interlace | 5 | No extension permitted |
| 2 | SEQ = Yes | 6 | Interlace adjust/keyed data |
| 3 | Write verify | 7 | LBLK specified |

Figure 6-1. Partition Control Appendage (PCA) Table Format

When searching by key (READE and READH macro instructions) you must know the relative address of at least one block on the track you wish to search. Once again, when you open the file, the current ID and end of data ID of the partition are initialized. However, you must initialize the current ID to the relative block address of a block on the track you wish to search. Next, you place the key for which you want a match (or match and higher) into the first key length bytes of the I/O buffer area. When you issue the READE or READH macro instruction, a search of the track begins. A successful search results in the current ID field being loaded with the address of the block retrieved by the match. If the SEQ keyword parameter was specified in the PCA macro instruction, the address contained in the current ID field will be the block just read plus 1.

When using the SEEK macro instruction, there is no updating of the PCA table entries. In this case, after the file is opened, place the relative block number of any block on the track you want to access into the current ID field of the PCA.

6.2.2. Block Addressing by Key

Blocks are addressed either by key or by relative ID. You create a partition using keyed data blocks (Figure 6—2) by specifying the KEYLEN keyword parameter of the PCA declarative macro instruction. The key is placed in the first part of the I/O buffer area and is left-justified; when the PUT macro instruction is issued, the block is then written from the I/O area and to disk by PIOCS. To read data blocks by key, place the key ID into the first key length area of I/O buffer area. The instruction to read allows you two options. First, you can access a specific block by using the READE macro instruction which searches for a matching (equal) key; this block is then read into the I/O area for you to process. You can use the READH macro instruction where the key is placed in the first part of the I/O buffer area. As the block with the matching key or higher is located, that block is read into the I/O buffer area.

6.2.3. Block Addressing by Relative Block Number

When you address by relative block number, the current ID field of the PCA will contain the relative block number of the current block being referenced. (The first block of each partition is relative block 1, the second is 2, etc.) Load the relative block number of the block you wish to access, then issue a GET macro instruction to read the block or a PUT macro instruction to write the block.

WITH KEYS

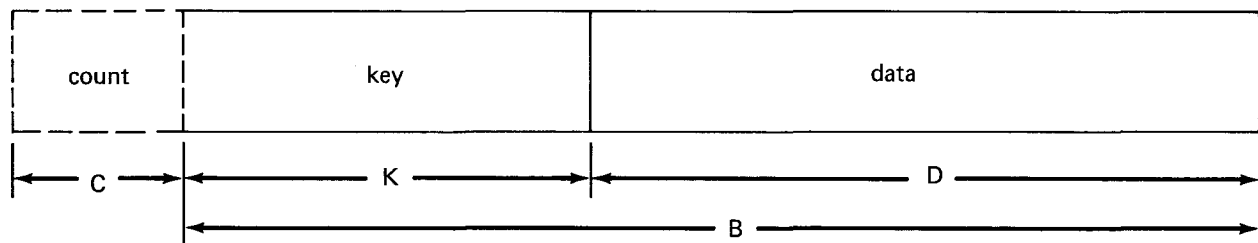
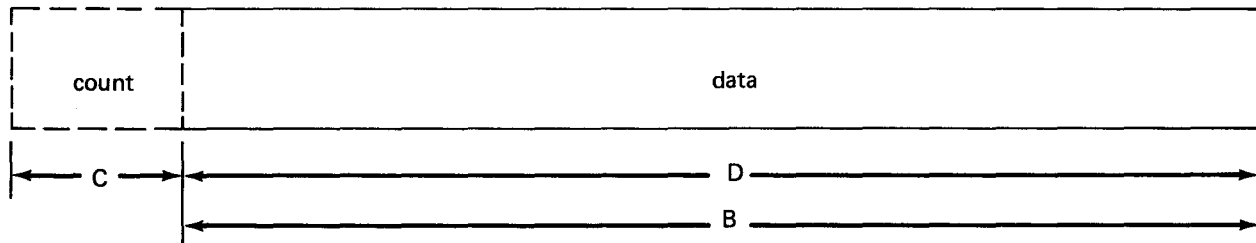


Figure 6—2. Record Formats for Disk Devices (Part 1 of 2)

WITHOUT KEYS



LEGEND:

- C = Count field length (8 bytes). Count field is used only by data management.
- K = Key field length (3—255 bytes)
- D = Data field length
- B = Block length (< track length and cannot span track boundaries)

Figure 6—2. Record Formats for Disk Devices (Part 2 of 2)

6.2.4. Disk Space Control

Space required for new files is allocated and scratched using the standard disk space management routines. Requests for temporary disk space are handled through job control; space allocated in this manner is released at the end of job step.

Allocation of disk space to your partitions is on a serial basis; first, the partition 1 space requirements are filled from the first available tracks of the extents, then the other partitions are satisfied in sequence.

Specify the initial space allocation to a partition using the SIZE keyword parameter of the PCA macro instruction. This is represented as a percentage value of the overall file.

To calculate the SIZE entry, use the following formula:

$$\text{SIZE} = \frac{\text{BLKSIZE} \times \text{Percentage}}{\text{Total}}$$

For example, if you have a file requiring three partitions, as follows:

Partition 1

Block size is 1024 bytes. Approximately 40% of the blocks in the file are this size.

Partition 2

Block size is 256 bytes. Approximately 50% of the blocks in the file are this size.

Partition 3

Block size is 768 bytes. Approximately 10% of the blocks in the file are this size.

NOTE:

Block size is specified for each partition by the BLKSIZE keyword parameter in the PCA macro instruction.

Then, divide the result of each BLKSIZE times the percentage value by the total for all the partitions. If necessary, round the results so that the total for all partitions does not exceed 100 percent. Use this value as the specification for the SIZE keyword parameter in the PCA macro instruction for the partition.

| <u>Partition No.</u> | <u>BLKSIZE</u> | <u>Percentage</u> | <u>Result</u> | <u>SIZE</u> |
|----------------------|----------------|-------------------|---------------|-------------|
| 1 | 1024 | 40 | 40960 | 67 |
| 2 | 256 | 50 | 12800 | 21 |
| 3 | 768 | 10 | <u>7680</u> | <u>12</u> |
| | | Total | 61440 | 100 |

If all the blocks in your file are of equal size and each partition will contain the same number of words, you would simply use the percentage of the overall file with the SIZE keyword entry. For example, if your file consisted of three partitions, each containing the same number of blocks of the same size, the entry in the PCA macro instruction for each partition of the file would be SIZE=33.

Dynamic allocation is given as a unit of store (UOS keyword parameter). The unit of store is a percentage of secondary allocation and cannot exceed 100 percent. The total of secondary allocation is given by an EXT job control statement. If you do not use the SIZE keyword parameter to specified initial space allocation, the initial allocation to the partition is equal to the percentage specified in the UOS keyword parameter. When the UOS keyword parameter is not specified, no extension to your file can be made. When you do not specify either the SIZE or UOS keyword parameter, an amount of disk space equal to 1 percent of your files is allocated to the partition.

Once the file is established and you have specified a UOS, the partition can be extended by this percentage. This occurs each time your PUT macro instruction references a block beyond the current maximum block address for the partition. If the new allocation cannot satisfy the current PUT macro instruction demands, an error will be indicated. However, partitions will not be extended beyond the volume on which the file resides.

6.2.5. Record Interlace

Record interlace is a technique available to you that reduces the effects of rotational delay when processing partitioned files, accessed sequentially. The interlace function is optional and, when specified, is completely controlled by SAT.

During file creation, the interlace function automatically arranges the physical records (blocks) in the file so that several blocks can be accessed during one disk rotation and, at the same time, provides the necessary interval between block accesses (time frame). This time frame is based on a lace factor specified in the LACE keyword parameter when you define a partition by using the PCA macro instruction (6.3.2). When the file is opened by the OPEN macro instruction, this lace factor is applied to the performance of the particular device type being used.

The lace factor determines the spacing of sequential blocks on the track; a lace factor of 4 results in the next logical block occurring at a minimum interval of 4 blocks. Calculation of the lace factor is described in 6.2.5.2.

Figure 6—3 illustrates some of the factors involved in accomplishing interlace:

- Number of physical blocks on each track
- I/O time (time required to input or output a block)
- Sector time (average interval available to each block)
- Time frame (time between block accesses)

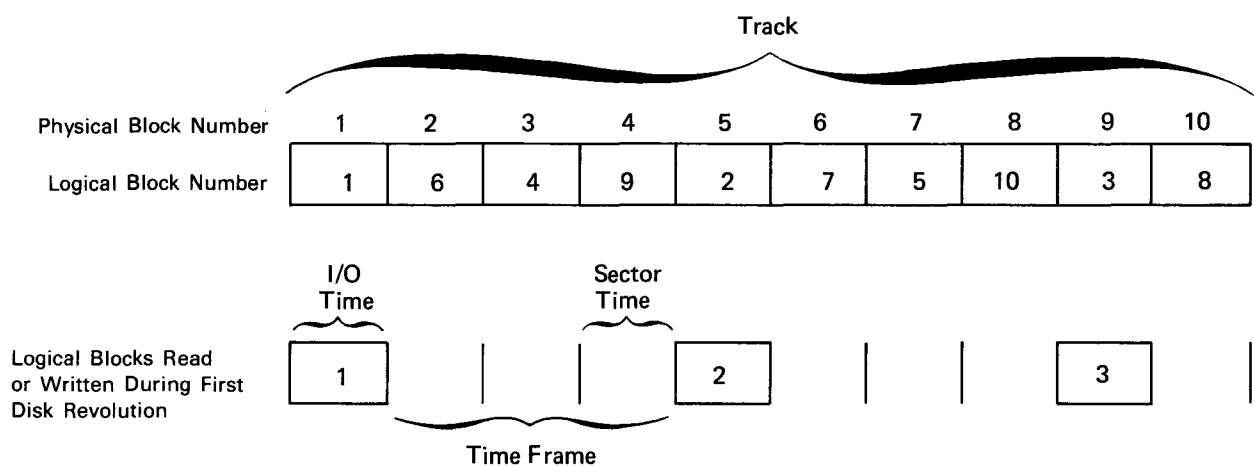


Figure 6—3. Definition of Interlace Variables

6.2.5.1. Interlace Operation

Figure 6—4 illustrates the advantage of interlace accessing. For example, assume that a file contains ten 1024-byte blocks per track and the disk subsystem being used has a rotational speed of 21.4 ms per revolution. If the blocks were stored sequentially on the track in contiguous locations, it would require ten revolutions to sequentially access all ten blocks, or a total of 214 ms (exclusive of head positioning and latency for initial access). However, using an interlace factor of 4, all ten blocks could be accessed in 81.32 ms because the last block would be retrieved before completion of the fourth disk revolution. This performance can be obtained only if your required time between block accesses is not more than the actual time frame.

| | | Without Interface | | | | | | | | | | With Interface | | | | | | | | | | | | | | | | |
|---|-------------------|-------------------|---|---|---|---|---|---|---|---|----|----------------|---|---|---|---|---|---|----|---|----|--|---|---|--|--|--|--|
| Physical Block No. | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | | | | | | | |
| Logical Block No. | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 1 | 6 | 4 | 9 | 2 | 7 | 5 | 10 | 3 | 8 | | | | | | | |
| Logical Blocks Read or Written During Each Disk Revolution | Revolution No. | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | 1 | 1 | | | | | | | | | | 1 | | | | | 2 | | | | | | 3 | | | | | |
| | 2 | | | 2 | | | | | | | | | | | 4 | | | | | 5 | | | | | | | | |
| | 3 | | | | 3 | | | | | | | | | | 6 | | | | 7 | | | | | 8 | | | | |
| | 4 | | | | | 4 | | | | | | | | | | | 9 | | | | 10 | | | | | | | |
| | 5 | | | | | | 5 | | | | | | | | | | | | | | | | | | | | | |
| | 6 | | | | | | | 6 | | | | | | | | | | | | | | | | | | | | |
| | 7 | | | | | | | | 7 | | | | | | | | | | | | | | | | | | | |
| | 8 | | | | | | | | | 8 | | | | | | | | | | | | | | | | | | |
| | 9 | | | | | | | | | | 9 | | | | | | | | | | | | | | | | | |
| 10 | | | | | | | | | | | 10 | | | | | | | | | | | | | | | | | |

Figure 6-4. Interlace Accessing

Successful interlace operation requires that the I/O orders must be issued within a specific time frame. The lace factor, therefore, determines how blocks are to be spaced on the track to ensure that the actual time frame (which includes both user and SAT overhead) is equal to or greater than your estimate of required time between block accesses.

A lace factor of 4 means that the blocks will be spaced in sufficient intervals (every 4th block) to produce an actual time frame that is equal to or greater than the estimated required time frame.

To calculate the lace factor, use the formula described in 6.2.5.2. Although the formula is based on the use of the 8416 disk subsystem, all lace factor calculations must be performed by using this formula, regardless of the actual disk subsystem being used. When the file is opened by the OPEN macro instruction, the specified lace factor will be applied to the performance of the particular disk subsystem being accessed. If necessary, SAT will adjust the lace factor to the capacity and speed of the specific device so that a similar time frame will be maintained for interlaced files processed on all supported disk subsystems.

6.2.5.2. Lace Factor Calculation

The lace factor is calculated in two steps by using the following formula:

1. $\frac{\text{BLKSIZE}}{256} \times .535 = \text{Calculated Sector Time}$
2. $\frac{\text{Required Time Frame}}{\text{Calculated Sector Time}} + 1 \text{ (rounded high)} = \text{Lace Factor}$

For example, if you are using a block size of 1024 bytes, first calculate the sector time in milliseconds:

1. $\frac{1024}{256} \times .535 = 2.14 \text{ ms}$

Then calculate the lace factor using an estimate of the processing time required between block accesses. For this example, let us use a required time frame estimate of 7.48 ms:

2. $\frac{7.48}{2.14} = 3.49 + 1 = 4.49 \text{ rounded to } 4$

The result is a lace factor of 4. In the PCA macro instruction statement for this partition, enter the keyword parameter LACE=4.

NOTE:

When the time frame exceeds 21.4 ms, it should be divided by 21.4 and the remainder should be used as the time frame in the foregoing calculation.

6.2.6. Accessing Multiple Blocks

When you are engaged in sequential processing (SEQ=YES specified in PCA macro instruction), you can read or write more than one block with each SAT imperative macro instruction that is issued. This is done by specifying the number of blocks you wish to access together by using the LBLK keyword parameter of the PCA macro instruction. However, when you use multiple buffer accessing, be certain that your I/O buffer area has enough contiguous space to contain the blocks. Also, if you are creating the partition by using the format write option, (FORMAT=NO), an additional 8-byte area, used to construct the count field, must immediately precede the first buffer area. During input operations, fewer than the requested number of blocks may be read if the end of data ID is encountered. The I/O count field (bytes 44 and 45) of the DTF (Figure 6—5) will contain the number of buffers not acted upon.

Normally, SAT makes a single reference to physical IOCS for the number of blocks requested. If an end-of-track condition is encountered for any block other than the last block of the request, SAT makes an additional reference to physical IOCS to access the next track. For interlaced files, SAT makes one reference to physical IOCS for each block requested. If an end-of-block condition is encountered on the last, or only, block requested, an information bit will be set in the error status field (byte 50, bit 0, of the DTF) to indicate the last block on that track has been accessed.

The LBLK keyword parameter specifies the number of blocks required, within a range from 1 to 255; however, the total size of the buffer cannot exceed 32,767 bytes.

| Byte | 0 | 1 | 2 | 3 |
|------|---------------------------|-------------------------------|-------------------|-----------------------------|
| 0 | control 1 | I/O error count | transmission byte | control 2 |
| 4 | next CAW | | | |
| 8 | residual byte count | | reserved | |
| 12 | CCW address | | | |
| 16 | PIOCB address | | | |
| 20 | sense byte 0 | sense byte 1 | sense byte 2 | sense byte 3 |
| 24 | sense byte 4 | sense byte 5 | device status | channel status |
| 28 | filename | | | |
| 36 | module flags | | number of vols | current vol no. |
| 40 | current PCA address | | | |
| 44 | I/O count | | DTF type code | |
| 48 | DTF type code (cont) | function code | error flags | |
| 52 | IOCS module address | | | |
| 56 | err msg code | error exit address | | |
| 60 | command code | current I/O address | | |
| 64 | current block size | | reserved | sectors/block |
| 68 | reserved | | current head | reserved |
| 72 | current cylinder | | current sector | reserved |
| 76 | address of extent storage | | | |
| 80 | PCA count | allocation incr | share flags | ext table entries available |
| 84 | tracks per cylinder | | | |
| 88 | file low head | | file high head | |
| 92 | PCA ID 1 | address of PCA 1 | | |
| 96 | ~ ~ ~ ~ ~ | | | |
| 100 | PCA ID 7 | address of PCA 7 (if present) | | |



Figure 6-5. Define the File (DTF) Table Format (Part 1 of 2)

| MODULE FLAGS | | | ERROR FLAGS | | |
|--------------|-------|----------------------------|-------------|-----------------|--------------------------------|
| Byte 1 | Bit 0 | Open | Byte 1 | Bit 0 | Access to last record on track |
| | 1 | Wait required | | 1 | Invalid ID |
| | 2 | WAIT = Yes | | 2 | Invalid PCA |
| | 3 | Sector type disk | | 3 | Hardware error |
| | 4 | F2 active | | 4 | Reserved |
| | 5 | No extension made | | 5 | Reserved |
| | 6 | FCB not found | | 6 | Reserved |
| | 7 | Multiple I/O permitted | 7 | Reserved | |
| Byte 2 | Bit 0 | Search wait required | Byte 2 | Bit 0 | I/O complete |
| | 1 | Cylinder alignment | | 1 | Unrecoverable error |
| | 2 | Format entered by extend | | 2 | Unique unit error |
| | 3 | Reserved | | 3 | No record found |
| | 4 | Library lock required | | 4 | Unit exception |
| | 5 | FCB in core | | 5 | Reserved |
| | 6 | Single mount | | 6 | End of track |
| | 7 | Unassigned space available | 7 | End of cylinder | |

Figure 6—5. Define the File (DTF) Table Format (Part 2 of 2)

6.3. DISK SAT FILE INTERFACE

Interface with SAT files is through declarative and imperative macro instructions. The DTFFP declarative macro instruction is used to define your overall file structure, while a separate PCA declarative macro instruction is required to define each of the partitions which make up a particular file.

The imperative macro instructions allow you to control file activity; the set of imperative macro instructions varies slightly, depending upon the type of accessing you specify. The following paragraphs describe these interfaces in detail.

6.3.1. Define a New File (DTFFP)

When organizing your partitioned file, you must assign a unique name (filename) to the file and describe certain operating characteristics as well as physical characteristics of your file. This is accomplished by the define the file partitioned file (DTFFP) macro instruction which creates a table in main storage (Figure 6—5) that can be referenced by the system.

This is a declarative macro instruction and must not appear in a sequence of executable code.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|--|
| filename | DTFFP | PCA1=partition-name [: :] ,PCA7=partition-name [: :] ,ACCESS= { EXC EXCR SRDO SRD SUPD SADD } [,ALINE=YES] [,ERROR=symbol] [,EXTENTS=n] [,FCB=YES] [,LIBUP=YES] [,WAIT=YES] |

The DTFFP macro instruction provides up to six operating/physical characteristic specifications and allows you to name from 1 to 7 file partitions. In its most abbreviated form, the DTFFP macro instruction contains only the required partition names (one for each PCA macro instruction) supplied by using the PCA1 through PCA7 keyword parameters. For file operation, these keywords must be specified in sequence with no intervening keywords missing. The remaining six keyword parameters, when not specifically listed in your DTFFP macro instruction, assume a predetermined value or condition (default). These keyword parameter defaults are as follows:

| <u>Keyword</u> | <u>Default</u> |
|--------------------|---|
| ALINE | PCAs start on track boundaries. |
| ERROR | Program will terminate when a major file error occurs. |
| EXTENTS | No extent table will be generated with the DTFFP macro instruction. |
| FCB | The file control block (FCB), which controls file I/O, is placed into the transient area of main storage during file open operations. |
| LIBUP or ACCESS | The file being accessed cannot be written into. This is a read-only lock for the file. ← |
| WAIT | You must issue a WAITF macro instruction after each I/O operation (GET, PUT, READE, or READH). |

The default values and characteristics applied to your file partition represent the most common usage. However, you have the option of specifying your own parameters for these keywords. This enables tailoring the file to suit your own particular needs. For example, you may want to use your own error routine to handle file errors. The following options are available:

- When creating a file, you can have your PCAs start and end on cylinder boundaries by specifying ALINE=YES in your DTFFP macro instruction.
- When you want the program to branch to your own error routine when a file error occurs, provide the address (symbol) of the error routine by specifying ERROR=symbolic address.
- An extent table can be generated for you if you specify the EXTENTS keyword parameter. When your DTFFP macro instruction references one of the standard system library files (\$Y\$LOD, \$Y\$SRC, \$Y\$MAC, \$Y\$OBJ, or \$Y\$JCS), you must use the EXTENTS keyword parameter to specify the number of extent entries you want to be allocated. The number of extents required is calculated by adding the number of extents allocated (to the file) to the number of partitions in the file.

When standard system libraries are being accessed, 18 extents are recommended to be specified as EXTENTS=18.

- File control blocks (FCBs) are used to make information available about a file or partition to the system. Normally, the FCB is placed in the transient area when the OPEN macro instruction is issued. However, you may place an FCB in the I/O area specified in the PCA macro instruction for the first or only partition of the file. This area address is specified by the IOAREA1 keyword parameter of the PCA macro instruction (6.3.2).



- There are several ways to request a specific type of filelock. If you use LIBUP=YES, when the file is opened, it is reserved for exclusive use of the job step until it is closed. No access by any other task will be permitted.

You can request the same type of filelock using the ACCESS=EXC keyword parameter entry instead of LIBUP=YES. The ACCESS parameter provides an expanded filelock capability with more options available (see 6.3.1.1).



- Normally, you must issue a WAITF macro instruction after each I/O function to assure completion of the input or output operation and to set particular status bytes in the DTFFP reference table. However, you can have SAT initiate this waiting period by specifying WAIT=YES. When specifying the WAIT keyword parameter, you don't have to use the WAITF macro instruction.



6.3.1.1. Filelocks

The use of filelocks enables you to restrict access to your files. A filelock is applied when a file is opened and remains in effect until it is closed. You can choose the specific type of restriction you want for a file during the execution of your job step. For example, you may want exclusive use of the file, or you may want to permit other tasks to read but not write.

The files that may be locked and the type of filelock processing performed are determined by a combination of system generation, job control, and SAT options. The FILELOCK parameter at system generation (refer to the system installation user guide, UP-8074 (current version) specifies the type of filelocks available and the types of files affected. The LIBUP (6.3.1) or ACCESS (6.3.1.2) parameter in your DTFFP macro instruction, specifies the type of lock you want to be applied to that file. The LBL job control statement assigns a lock ID to your user file (refer to the job control user guide, UP-8065 (current version)), and the ACCESS parameter in the DD job control statement at run time adds or changes the ACCESS parameter in the DTFFP.



6.3.1.2. Shared Filelock Capability

The ACCESS parameter provides a greater filelock capability than the LIBUP parameter. They should not be used together. If both appear in the same DTFFP, the ACCESS parameter supersedes LIBUP. The ACCESS options can only be used if FILELOCK=SHARED was specified at system generation. The filelock options available with ACCESS are:

ACCESS= EXC

Requests exclusive use of the file. You may read, update, and extend the file. No access is permitted by any other task. This type of filelock is the same as that requested by the LIBUP=YES parameter entry.

ACCESS=EXCR

Requests exclusive-read use of the file. You may read, update, and extend the file. Other tasks may also read the file, but may not write.

ACCESS=SRDO

Requests shared-read-only access to the file. You intend only to read the file. Other tasks may also read the file. No writing is permitted. This type of filelock is the same as the default of LIBUP.

ACCESS=SRD

Requests shared-read access to the file. You intend to read the file. Other tasks may read, update, or extend the file.

ACCESS=SUPD

Requests shared-update access to the file. You intend to read and update the file, but will not be extending it. Other tasks may only read the file.

ACCESS=SADD

Requests shared-add access to the file. You intend to read, update, and extend the file. Other tasks may only read the file.

6.3.2. Defining a Partition (PCA)

Once your file is defined and each file partition is listed by using the PCA1 through PCA7 keyword parameters of the DTFPF macro instruction, the characteristics of each partition appendage must be described. This is done by using the partition control appendage (PCA) macro instruction.

This is a declarative macro instruction and must not appear in a sequence of executable code.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------------|---------------|--|
| partition-name | PCA | BLKSIZE=n ,IOAREA1=symbol [,EODADDR=symbol] [,FORMAT=NO] [,KEYLEN=n] [,LACE=n] [,LBLK=n] [,SEQ=YES] [,SIZE=n] [,UOS=n] [,VERIFY=YES] |

The partition name for a particular PCA macro instruction is the same as that assigned by the PCAn keyword parameter in the DTFPF macro instruction. The keywords allow you to specify up to 10 operating and physical characteristics for each partition; these characteristics are placed in a PCA table in main storage together with a current ID and end of data ID. In its most abbreviated form, it is required only that you specify the size, in bytes, of the blocks in the partition (BLKSIZE=n) and the address of an input/output area where the blocks are going to be processed (IOAREA1=symbol). The size of the I/O area is the same as the BLKSIZE specification. The remaining keywords, when not specifically listed, assume their default conditions as follows:

| <u>Keyword</u> | <u>Default</u> |
|----------------|--|
| EODADDR | When the GET macro instruction accesses the block with the relative block number equal to the end of data ID for that partition, SAT assumes there is no end of data routine for this partition and indicates that an invalid ID has been requested. |
| FORMAT | Space allocated to the partition on 8411, 8414, and 8430 disk subsystems is preformatted. This is used when writing new files in which each block is written in format (count field followed by either a data field or a key field and data field). |
| KEYLEN | Assumes blocks will not be referenced by key. |
| LACE | Assumes that no interlace is to be applied. LACE and FORMAT keyword parameters are mutually exclusive. |
| LBLK | One block (the size as specified in the BLKSIZE keyword parameter) comprises one logical block (LBLK=1). |
| SEQ | The file is not treated as a sequential file and you must provide the 4-byte current ID field at the address of the PCA being referenced for each I/O request (WRITE ID and READ ID macro instructions). |
| SIZE | The new file partition being defined requires one percent of the total file allocation (SIZE=1). |
| UOS | The Unit Of Store (secondary allocation of disk space) has a value of 1. |
| VERIFY | No verification (parity check) of block writing is performed. |

The default values of the PCA macro instruction represent the most common usage. However, you have the option of specifying your own parameters for these keywords. Certain keywords are interrelated; the following are some examples:

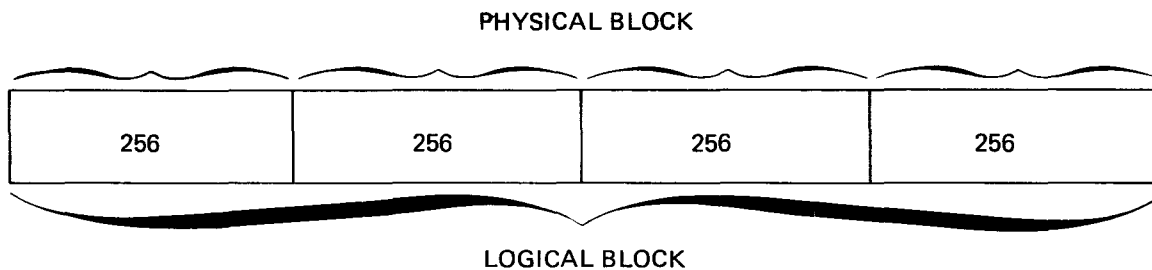
- Some of these are mutually exclusive, like the FORMAT and LACE keyword parameters since you cannot use format write (WRITE ID) and interlace simultaneously.
- Some are required together, like the SEQ and EODADDR keyword parameters.
- The LACE keyword parameter must be specified for interlace files. The lace factor is based upon the 8416 disk subsystem and is adjusted by SAT for all other disk subsystems.
- The SIZE keyword parameter is applicable only to files being created.
- The BLKSIZE, IOAREA1, and the LBLK keyword parameters are also interrelated.

User-supplied options to the PCA macro instruction keywords are as follows:

- When specifying blocksize (BLKSIZE keyword parameter), also specify the size of the I/O area. When using 8416 disk subsystem, specify this value in multiples of 256 since this is the size of the fixed sectors on that device. The multiple buffer keyword parameter (LBLK) specifies the number of blocks that can fit within this I/O area.
- If specifying sequential file processing (SEQ=YES), inform the program at which point file processing should terminate. This is done by specifying the end of data (EODADDR) keyword parameter address. When the GET macro instruction accesses the block with the relative block number equal to the end of data ID for that partition, SAT transfers control to the address specified by the EODADDR keyword parameter.
- If loading your file on a device where the space allocated is not preformatted (FORMAT=NO), a format write command is issued by SAT for each PUT macro instruction that references a relative block number equal to the end-of-data address of the partition being accessed. A data write command is issued by SAT for each PUT macro instruction that references relative block numbers less than the current end of data address.

This means that data written in the area outside the existing file partition area is written as a new file while those within the existing file partition are written as update records.

- The address of the input/output area needed to process records is specified by the IOAREA keyword parameter. The length of this area is specified by the BLKSIZE keyword parameter.
- When you have interlaced creation or retrieval of sequential files, specify the LACE keyword parameter to achieve most efficient processing. This value is computed for the 8416 disk subsystem and is modified by SAT to make other disk subsystems conform to a similar access pattern. A thorough discussion of interlace operation and computation is provided in 6.2.5.
- Under certain circumstances, you may desire to retrieve more than one physical block to construct one logical block. In this case, specify the block size through the BLKSIZE keyword parameter. The LBLK keyword parameter would then specify the number of physical blocks within the logical block. For example, assume that your physical blocks are 256 bytes long and that you must have four of these to make up your logical block.



The following would be specified:

```
BLKSIZE=256
LBLK=4
```

- When you wish to process a file sequentially, you can specify SEQ=YES. When the OPEN macro instruction is issued, the open transient routine sets the current ID field to relative block 1 of the partition. Each subsequent GET or PUT macro instruction that is issued will transfer the next block in sequence to or from main storage. The current ID is updated after each GET or PUT macro instruction has been waited.

Random processing of the sequential file can be achieved as well as sequential processing of random portions of the file by supplying the new value in the current ID field before any GET or PUT macro instruction is issued.

- At the time that you are organizing your file, specify the space required for the partition in the terms of a percentage of the overall file allocation. For example, if your file contained four partitions of equal size, you would specify SIZE=25.
- If you feel that additional space may be needed to expand your file partition, specify this space in increments called units of store (UOS). A unit of store is a percentage of secondary allocation.

Each time an attempt is made to write a block with a relative block number larger than the current maximum for the partition, a unit of store is added to the partition. For example, suppose that you had a secondary allocation of 10 cylinders and you wished to add 2 cylinders to your partition each time you needed more space. You would specify: UOS=20 since 2 cylinders are 20 percent of your secondary allocation.

If the block chosen to be added to the partition exceeds the unit of store, an invalid ID indication would be returned to the error field in your DTFPF table in main storage.

- If writing records to disk and you wish to be certain that the block written is complete and accurate, use the VERIFY=YES option. The blocks are check-read for parity. An additional disk rotation must be allowed for the verification process.
- If blocks are to be addressed by key, use the KEYLEN keyword parameter to specify the length (3 to 255 bytes) of the key field in formatted records.

6.3.3. Processing Partitioned SAT Files

Once you have established your file on disk (that is, you have issued DTFPF and PCA macro instructions to describe and name your file), use the imperative macro instructions to open, control, and close your file processing. These macro instructions are universal, but are normally grouped according to their use as follows:

- Processing Blocks by Key — OPEN, PUT, WAITF, READE/READH, SEEK, CLOSE
- Processing Blocks by Relative Number — OPEN, GET, PUT, WAITF, SEEK, CLOSE

The following paragraphs give a brief functional description of these imperative macro instructions. This description is followed by listing these macro instructions in 6.4 and includes a detailed description of their parameters and characteristics.

6.3.3.1. Processing Blocks by Key

| <u>Macro Instruction</u> | <u>Function</u> |
|--------------------------|---|
| OPEN | Initiates the open transient routine and identifies the file (as listed in the DTFPF macro instruction) to be processed. |
| PUT | Identifies the file and partition to be accessed. Issues the write for the indicated block. |
| WAITF | Identifies the file and ensures completion of the current I/O. If the current I/O was a successful READE or READH, it places the ID of the block accessed in the current ID field. Updates the current ID by 1 if the SEQ=YES keyword parameter was specified. |
| READE | Initiates the search for a block by key of a particular track. You must place a relative block number, that is on the track to be searched, in the current ID field of the PCA table. You must also place the key of the block to be accessed in first key length bytes of the buffer area. |
| READH | Same as for READE except that the search is for a block that is equal to the key specified or higher than the key. |
| SEEK | Initiates movement of the disk heads to a particular track or disk. It is your responsibility to place the relative address of a block on that track in the current ID field of the PCA table. |
| CLOSE | Identifies the file. After the file processing has been completed or when the end of data ID has been detected, it initiates the transient file close routine. |

6.3.3.2. Processing by Relative Block Number

| <u>Macro Instruction</u> | <u>Function</u> |
|--------------------------|--|
| OPEN | Initiates the open transient routine and identifies the file (as listed in the DTFPF macro instruction) to be processed. Initializes the start ID entry in the PCA tables of the file. |
| GET | Identifies the file and partition to be accessed. Issues the read for the indicated block. |
| PUT | Identifies the file and partition to be accessed. Issues the write for the indicated block. |
| WAITF | Identifies the file and assures completion of the current I/O. Updates the current ID by 1 if the SEQ=YES keyword parameter was specified. |
| SEEK | Initiates movement of the disk heads to a particular track on disk. It is your responsibility to place the relative address of a block on that track in the current ID field of the PCA table. |
| CLOSE | Identifies the file. After the file processing has been concluded or when the end of data ID has been detected, it initiates the transient file close routine. |

6.4. CONTROLLING YOUR DISK FILE PROCESSING

After you have specified the details of the file and partition you wish to access through the declarative macro instructions, the imperative macro instructions described in the following paragraphs actually control your file accessing. The sequence of these macro instructions for a particular type of processing is listed in 6.3.3.1 and 6.3.3.2, together with a brief description of their function.

6.4.1. Open a Disk File (OPEN)

Function:

The OPEN macro instruction opens a file defined by the DTFPF and PCA macro instructions so that it can be accessed by the logical IOCS.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|---|
| [symbol] | OPEN | { filename-1 [, ..., filename-n] } (1) |

Positional Parameter 1:

filename-1

Specifies the symbolic address of the DTFPF macro instruction in the program corresponding to the file to be opened.

(1)

Indicates that register 1 has been preloaded with the address of the DTFPF macro instruction.

Positional Parameter n:

filename-n

Successive entries specify the symbolic addresses of the DTFPF macro instructions in the program corresponding to the additional files to be opened.

Use this form (for example, OPEN FILE1, FILE2) when more than one lockable file is to be accessed by a single task. This opens all the files named and applies the required read or write locks at the same time. In this way you can avoid the possibility of two jobs locking each other out with each one waiting for the other to give up its file. The operator would then have to cancel one of the jobs to remove the stalemate and continue processing.

After the file has been defined by the DTFPF and PCA macro instructions, you must issue an OPEN macro instruction to initialize the file before any other access can be made. Use the GET macro instruction to access the first (or next) data block.

The transient routine called by the OPEN macro instruction allocates disk space to each of the partition control appendages from the VTOC file extents; these areas are then preformatted if necessary. If too little disk space has been allocated to a file to satisfy all PCA requirements, partitions requiring space may be extended during processing.

6.4.2. Retrieve Next Logical Block (GET)

Function:

The GET macro instruction reads a logical block from disk into main storage and makes it accessible for processing. The address into which the data is read is specified in the associated PCA macro instruction by the keyword parameter IOAREA1.

Format:

| LABEL | Δ OPERATION Δ | OPERAND | |
|----------|---------------|---------------------|---------------------|
| [symbol] | GET | { filename } (1) | { PCA-name } (0) |

Positional Parameter 1:

filename

Specifies the symbolic address of the DTFPF macro instruction in the program corresponding to the file being read.

(1)

Indicates that register 1 has been preloaded with the address of the DTFFP macro instruction.

Positional Parameter 2:

PCA-name

Specifies the symbolic address of the PCA macro instruction associated with the partition to be accessed.

(0)

Indicates that register 0 has been preloaded with the address of the PCA macro instruction.

PCA Table Content:

The OPEN macro instruction initializes the current ID field in the PCA table to the start ID of the partition. If the SEQ keyword parameter in the PCA macro instruction is used, the current ID field will be updated after each GET macro instruction has been waited.

If the SEQ keyword is not used, or random access is desired, it is your responsibility to preload the current ID field with the relative ID of the data block to be read. The current ID field is located at the address (label) of the PCA being referenced. This is a 4-byte field and contains a right-justified hexadecimal number representing the number of the block (relative to the first block in the partition) to be read.

6.4.3. Output a Logical Block (PUT)

Function:

The PUT macro instruction writes a logical block from main storage to disk. The main storage address from which the data is written is specified in the associated PCA macro instruction by the keyword parameter IOAREA1.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|---|
| [symbol] | PUT | { filename } (1) , { PCA-name } (0) |

Positional Parameter 1:

filename

Specifies the symbolic address of the DTFFP macro instruction in the program corresponding to the file being written.

(1)

Indicates that register 1 has been preloaded with the address of the DTFFP macro instruction.

Positional Parameter 2:

PCA-name

Specifies the symbolic address of the PCA macro instruction associated with the partition to be written.

(0)

Indicates that register 0 has been preloaded with the address of the PCA macro instruction.

PCA Table Content:

The OPEN macro instruction initializes the current ID field in the PCA table to the start ID of the partition. If the SEQ keyword parameter in the PCA declarative macro instruction is used, the current ID field will be updated after each PUT macro instruction has been waited.

If the SEQ keyword is not used, or random access is desired, it is your responsibility to preload the current ID field with the relative ID of the data block to be written. The current ID field is located at the address (label) of the PCA being referenced. This is a 4-byte field and contains a right-justified hexadecimal number representing the number of the block (relative to the first block in the partition) to be written.

6.4.4. Wait for Block Transfer (WAITF)

Function:

The WAITF macro instruction ensures that a command initiated by a preceding GET, PUT, READE, or READH macro instruction has been completed. When completed, the error status field contains the error status information pertaining to the I/O request. It is your responsibility to check these bits, which are in bytes 50 and 51 of the DTF table.

If the keyword parameter WAIT=YES was not specified in the DTFPF macro instruction, the WAITF macro instruction must be issued after a GET, PUT, READE, or READH macro instruction and before another imperative macro instruction is issued for that file.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|---------------------|
| [symbol] | WAITF | { filename } (1) |

Positional Parameter 1:

filename

Specifies the symbolic address of the DTFPF macro instruction in the program corresponding to the file being accessed.

(1)

Indicates that register 1 has been preloaded with the address of the DTFFP macro instruction.

6.4.5. Read by Key Equal/Read by Key Equal or Higher (READE/READH)

Function:

The READE and READH macro instructions initiate a search by key for a block having a key equal to or equal and higher to the key specified.

Format:

| LABEL | Δ OPERATION Δ | OPERAND | |
|----------|------------------------|-----------------------|-----------------------|
| [symbol] | { READE } { READH } | { filename } (1) | { PCA-name } (0) |

Positional Parameter 1:

filename

Specifies the symbolic address of the DTFFP macro instruction in the program corresponding to the file being processed.

(1)

Indicates that register 1 has been preloaded with the address of the DTFFP macro instruction.

Positional Parameter 2:

PCA-name

Specifies the symbolic address of the PCA macro instruction associated with the partition to be accessed.

(0)

Indicates that register 0 has been preloaded with the address of the partition to be accessed.

PCA Table Content:

After a successful search, the current ID entry in the PCA table is updated to reflect the relative number of the record retrieved. However, if SEQ=YES has been specified in the PCA macro instruction, the current ID field in the PCA table will be the relative block number plus 1.

6.4.6. Access a Physical Block (SEEK)

Function:

The SEEK macro instruction initiates movement of the disk read/write head to the position specified in the current ID field of the PCA. This is a 4-byte field which contains a right-justified hexadecimal number representing any block number on the track (relative to the first block in the partition) to which head movement will be initiated. It is your responsibility to store the desired relative block number in this field.

Format:

| LABEL | Δ OPERATION Δ | OPERAND | |
|----------|---------------|---------------------|---------------------|
| [symbol] | SEEK | { filename } (1) | { PCA-name } (0) |

Positional Parameter 1:

filename

Specifies the symbolic address of the DTFPF macro instruction in the program corresponding to the file being accessed.

(1)

Indicates that register 1 has been preloaded with the address of the DTFPF macro instruction.

Positional Parameter 2:

PCA-name

Specifies the symbolic address of the PCA macro instruction associated with the partition to be accessed.

(0)

Indicates that register 0 has been preloaded with the address of the PCA macro instruction.

6.4.7. Close a Disk File (CLOSE)

Function:

The CLOSE macro instruction performs the required termination operations for a file. Once the CLOSE macro instruction has been issued for a file, only the OPEN macro instruction may reference that file.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|---|
| [symbol] | CLOSE | { filename-1 [,,,filename-n] } (1) *ALL |

Positional Parameter 1:

filename-1

Specifies the symbolic address of the DTFFP macro instruction in the program corresponding to the file to be closed.

(1)

Indicates that register 1 has been preloaded with the address of the DTFFP macro instruction.

***ALL**

Specifies that all files currently open in the job step are to be closed.

Positional Parameter n:

filename-n

Successive entries specify the symbolic addresses of the DTFFP macro instructions in the program corresponding to the additional files to be closed.

6.5. SAT FOR TAPE FILES

The OS/3 tape system access technique (TSAT) is a generalized input/output control system that provides a standard interface to physical IOCS for magnetic tape subsystems. It performs the basic functions of a tape data management system and provides block level I/O for sequential tape files.

Interface with TSAT files is through declarative and imperative macro instructions. You use the SAT and TCA declarative macro instructions to define the characteristics of the file and the data management technique to be used to process the file. The SAT macro instruction creates the DTF table for the file, and the TCA macro instruction creates the appendage to the table. These macro instructions are described in 6.8. You use the OPEN, GET, PUT, CNTRL, WAITF, and CLOSE imperative macro instructions to control file processing. These are described in 6.9.

All files processed by TSAT are written in a forward direction, and can be read forward and backward. The CNTRL macro instruction initiates nondata operations on the device and can be issued whether or not the file is open.

To use TSAT, you must observe tape label conventions (described in 6.6) and tape volume and file organization conventions (described in 6.7).

If you are processing block numbered tapes, you must also observe the special conventions applicable to these tapes. Requirements and processing for block numbered tapes are summarized in 6.10.

6.6.1. Volume Label Group

A volume label group consists of a single volume label (VOL1). The VOL1 label identifies the tape reel and its owner, and it is used to check that the proper reel is mounted. When a tape is first used at an installation, its volume serial number (VSN) and other volume information, as shown in Figure 6—6, are specified by parameter cards supplied to a standard utility routine that writes the label. The serial number is also written on the exterior of the reel for visual identification.

If you want logical IOCS to prep the volumes of a standard labelled file, NEW must be specified as a parameter of the LFD job control statement associated with that file. Logical IOCS will then prep the volumes from the information supplied on the associated VOL and LBL job control statements.

When you issue an OPEN macro instruction to an output tape, its open-and-rewind options are executed first, and then the tape is checked to see if it is at the load point. If it is at the load point, the VOL1 label is read (if in a nonprepping mode) and the volume serial number is checked and saved for use in the file header labels (HDR1 and HDR2). The tape is then positioned so that the volume labels are not destroyed, and no further volume label processing is performed.

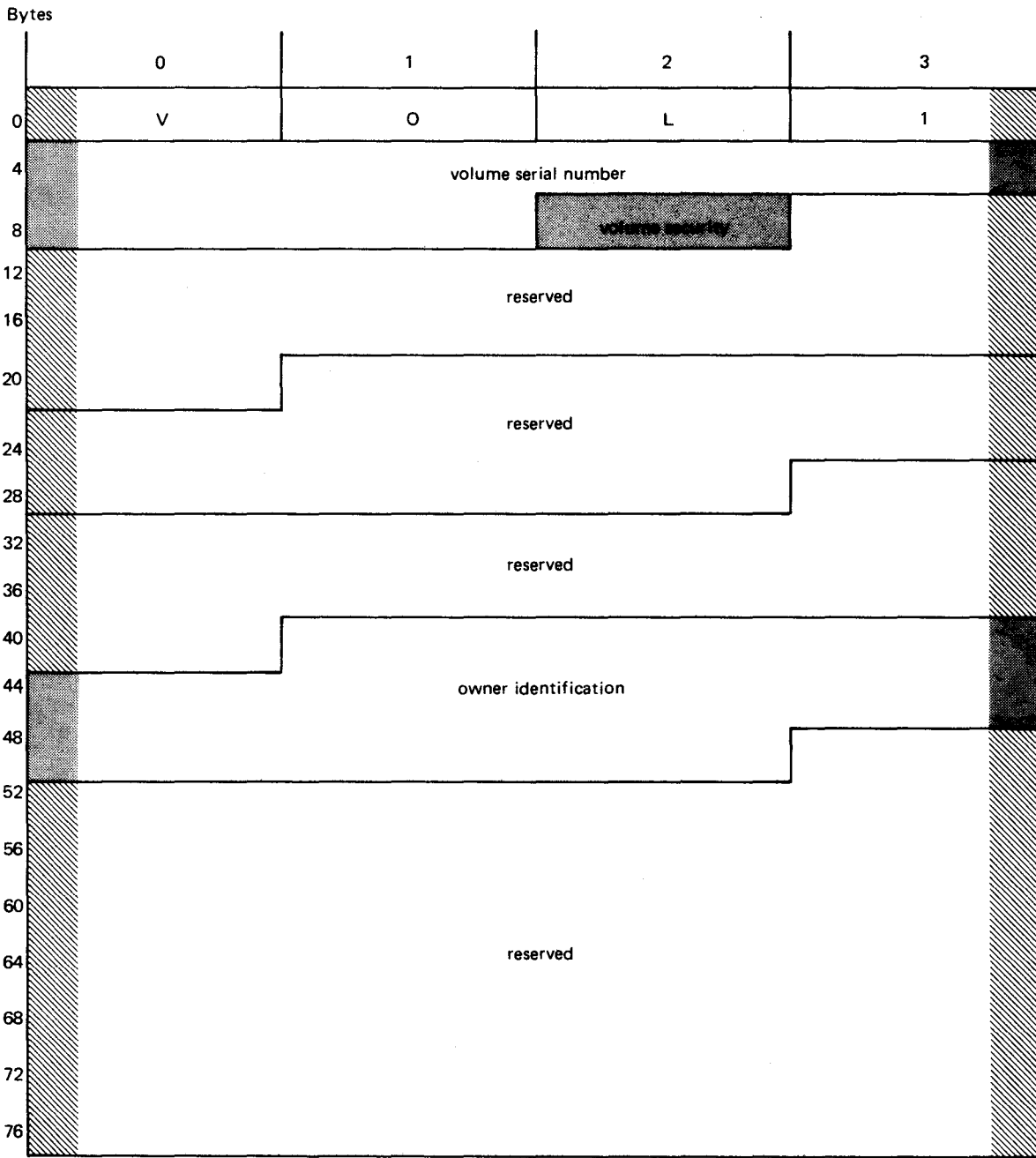
If the output tape is not at the load point after the open-and-rewind options are performed, TSAT assumes that the tape is positioned between the two ending tape marks of the previous file or just prior to the HDR1 label of an existing file. In either case, no volume label checking or creation is performed.

For an input tape, the OPEN transient first executes the open-and-rewind options and then checks to see whether the tape is at the load point. If it is, the VOL1 label is read and the volume serial number is used to check the file serial number in the appropriate file header or trailer label. The tape is then positioned to the proper file header or trailer label as specified in the file sequence number field of the associated LBL job control statement, and no further volume label processing is performed.

If the input tape is not at the load point after the open-and-rewind options are performed, TSAT assumes that the tape is positioned between the two ending tape marks of a previously created file or just prior to the HDR1 label of an existing file. In either case, no further volume label processing is performed.

When any volume label is encountered during the processing of a CLOSE macro instruction for an input tape and you have specified READ=BACK in the TCA macro instruction, the label is bypassed without processing.

The format of the volume label is shown in Figure 6—6. The fields are described in Table 6—1.



LEGEND:



-  Generated by TSAT or reserved for system expansion.
-  Written by TSAT from user-supplied data.

Figure 6-6. Tape Volume 1 (VOL1) Label Format for an EBCDIC Volume

Table 6-1. Tape Volume 1 (VOL1) Label Format, Field Description for an EBCDIC Volume

| Field | Initialized By | Bytes | Code | Description |
|----------------------|----------------|-------|--------|---|
| Label identifier | Tape prep | 0-2 | EBCDIC | Contains VOL to indicate that this is a volume label |
| Label number | Tape prep | 3 | EBCDIC | Always 1 for the initial volume label |
| Volume serial number | Tape prep | 4-9 | EBCDIC | Unique identification number assigned to this volume by your installation. TSAT expects 1- to 6-alphanumeric characters, the first of which is alphabetic |
| Volume security | TSAT | 10 | EBCDIC | Reserved for future use by installations requiring security at the reel level. Currently blank |
| (Reserved) | ----- | 11-20 | EBCDIC | Contains blanks (40 ₁₆) |
| (Reserved) | ----- | 21-30 | EBCDIC | Contains blanks (40 ₁₆) |
| (Reserved) | ----- | 31-40 | EBCDIC | Contains blanks (40 ₁₆) |
| Owner identification | Tape prep | 41-50 | EBCDIC | Unique identification of the owner of the reel: any combination of alphanumerics |
| (Reserved) | ----- | 51-79 | EBCDIC | Contains blanks (40 ₁₆) |

NOTE:

For ASCII files, Bytes 0-36 of a VOL1 label have the same significance as shown in the preceding example. Bytes 37-50 indicate the owner identification field. Bytes 51-78 are blank and are reserved for future standardization. Byte 79 indicates the label standard level, and when set to 1, indicates formats on this volume meet the American National Standard, X3.27-1969.

6.6.2. File Header Label Group

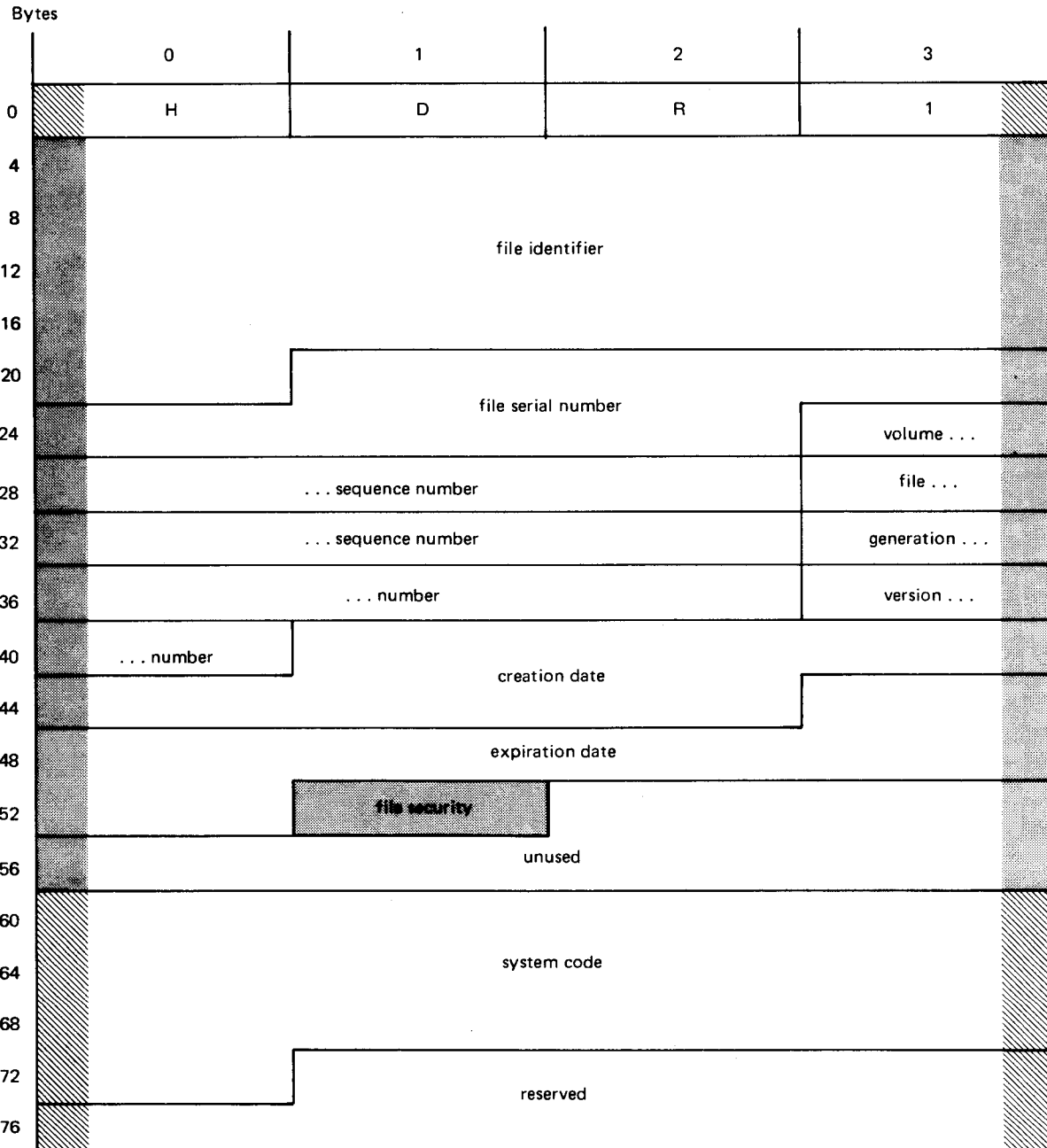
The file header label group consists of two labels: the file header 1 label (HDR1) and the file header 2 label (HDR2).

6.6.2.1. First File Header Label (HDR1)

The first file header label (HDR1), which identifies the file, is written at the beginning of each file. The HDR1 label is required and has the fixed format shown in Figure 6-7; its fields are described in Table 6-2. All fields in the HDR1 label may be specified in the job control stream.

For input files, all fields up to and including the system code in the HDR1 label are checked against values specified in the LBL job control statement. Only those fields for which values have been supplied are checked. However, if you specified READ=BACK in the TCA macro instruction, the HDR1 label is bypassed without processing. For multfile input volumes, you should specify the file sequence number in the LBL job control statement to ensure proper tape positioning.

For output files, the tape must be positioned properly before the files can be opened. On file open, the expiration date in the HDR1 label is checked against the current or actual calendar date to determine if the associated file has expired. If the file has expired, the tape is positioned so that the old HDR1 label is written over. The new HDR1 label is set up from values specified by the LBL job control statement and is written on the tape.



LEGEND:



-  Generated by TSAT or reserved for system expansion.
-  Written by TSAT from user-supplied data.

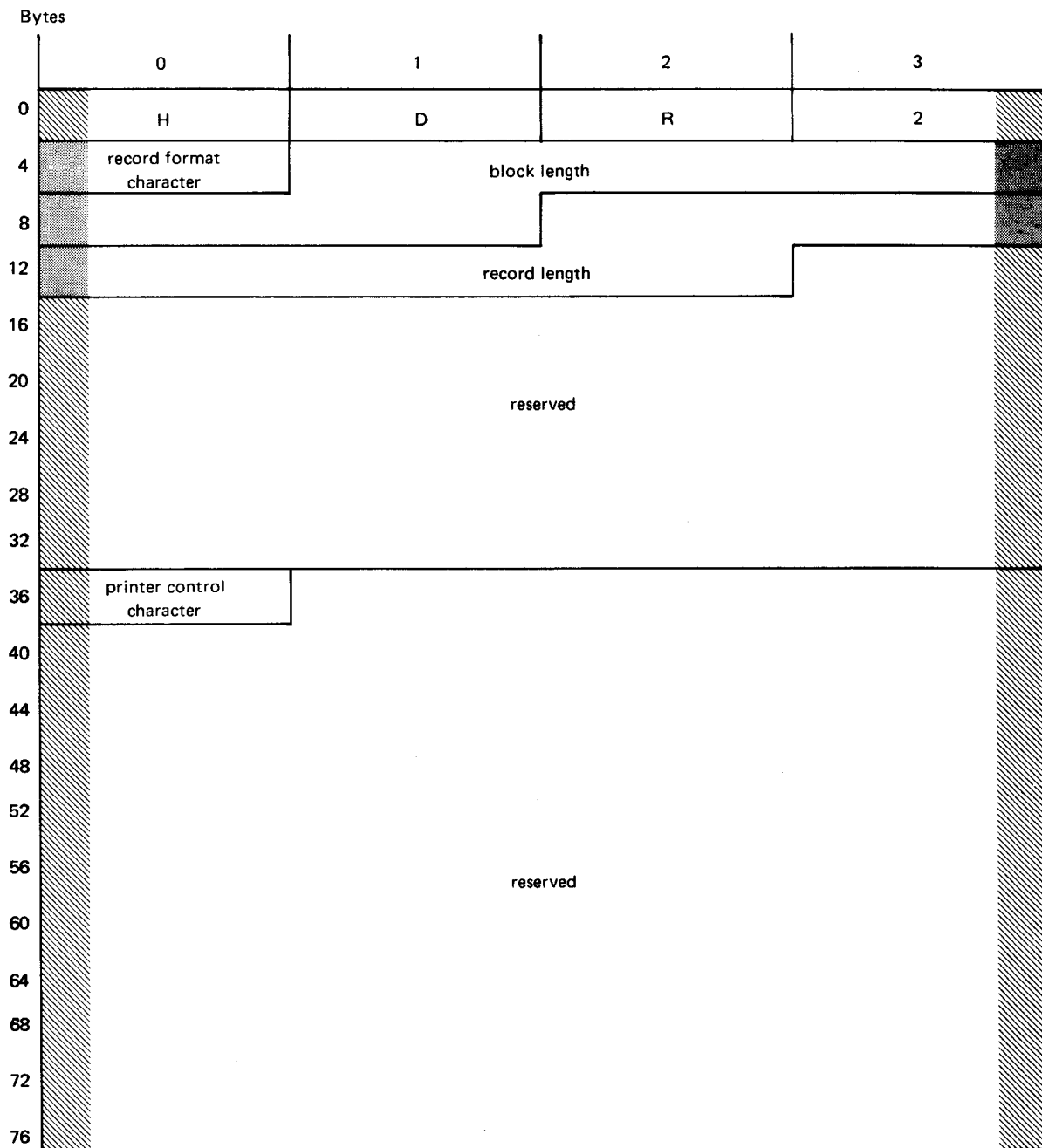
Figure 6-7. First File Header Label (HDR1) Format for an EBCDIC Tape Volume

Table 6-2. First File Header Label (HDR1), Field Description

| Field | Bytes | Description |
|------------------------------|-------|---|
| Label identifier | 0-2 | Contains HDR to indicate a file header label |
| Label number | 3 | Always 1 |
| File identifier | 4-20 | A 17-byte configuration that uniquely identifies the file. It may contain embedded blanks and is left-justified in the field if fewer than 17 bytes are specified. |
| File serial number | 21-26 | The same as the VSN of the VOL1 label for the first reel of a file or a group of multifile reels |
| Volume sequence number | 27-30 | The position of the current reel with respect to the first reel on which the file begins. This number is used with multivolume files only. |
| File sequence number | 31-34 | The position of this file with respect to the first file in the group |
| Generation number | 35-38 | The generation number of the file (0000-9999) |
| Version number of generation | 39-40 | The version number of a particular generation of a file |
| Creation date | 41-46 | The date on which the file was created, expressed in the form YYDDD and right-justified. The leftmost position is blank. |
| Expiration date | 47-52 | The date the file may be written over or used as scratch, in the same form as the creation date |
| File security indicator | 53 | Reserved for file security indicator. Indicates whether additional qualifications must be met before a user program may have access to the file. 0 = No additional qualifications are required. 1 = Additional qualifications are required. |
| (Unused) | 54-59 | Unused field, containing EBCDIC 0's |
| System code | 60-72 | Reserved for system code, the unique identification of the operating system that produced the file |
| (Reserved) | 73-79 | Reserved field, containing blanks (40_{16}). |

6.6.2.2. Second File Header Label (HDR2)

The second file header label (HDR2) acts as an extension of the HDR1 label and is a required label. Unless the HDR2 label was created by the OS/3 or OS/7 operating system as indicated in the system code field of the HDR1 label, the HDR2 label is ignored by TSAT. Figure 6-8 shows the format of the HDR2 label; Table 6-3 describes its fields.



LEGEND:



Generated by TSAT or reserved for system expansion.



Written by TSAT from user-supplied data.

Figure 6—8. Second File Header Label (HDR2) Format for an EBCDIC Tape Volume

Table 6-3. Second File Header Label (HDR2), Field Description

| Field | Bytes | Description | | | | | | | | | | | | |
|---------------------------|--|--|-----------|---------|---|--|---|--------------|---|---------|---|-----------|---|--|
| Label identifier | 0-2 | Contains HDR to indicate a file header label | | | | | | | | | | | | |
| Label number | 3 | Always 2 | | | | | | | | | | | | |
| Record format character | 4 | <table border="1"> <thead> <tr> <th>Character</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>D</td> <td>Variable-length (ASCII), with length fields specified in decimal</td> </tr> <tr> <td>F</td> <td>Fixed-length</td> </tr> <tr> <td>S</td> <td>Spanned</td> </tr> <tr> <td>U</td> <td>Undefined</td> </tr> <tr> <td>V</td> <td>Variable-length (EBCDIC), with length fields specified in binary</td> </tr> </tbody> </table> | Character | Meaning | D | Variable-length (ASCII), with length fields specified in decimal | F | Fixed-length | S | Spanned | U | Undefined | V | Variable-length (EBCDIC), with length fields specified in binary |
| Character | Meaning | | | | | | | | | | | | | |
| D | Variable-length (ASCII), with length fields specified in decimal | | | | | | | | | | | | | |
| F | Fixed-length | | | | | | | | | | | | | |
| S | Spanned | | | | | | | | | | | | | |
| U | Undefined | | | | | | | | | | | | | |
| V | Variable-length (EBCDIC), with length fields specified in binary | | | | | | | | | | | | | |
| Block length | 5-9 | Five EBCDIC characters specifying the maximum number of characters per block | | | | | | | | | | | | |
| Record length | 10-14 | Five EBCDIC characters specifying the record length for fixed-length records. For any other record format, this field contains 0's. | | | | | | | | | | | | |
| (Reserved) | 15-35 | Reserved for future system use | | | | | | | | | | | | |
| Printer control character | 36 | <p>One EBCDIC character indicating which control character set was used to create the data set.</p> <p>A=Special (ASA) control character present D=Device independent control character present M=IBM control character present U=SPERRY UNIVAC control character present</p> | | | | | | | | | | | | |
| (Reserved) | 37-79 | Reserved for future system use | | | | | | | | | | | | |

NOTE:

For ASCII files, bytes 0-14 of a HDR2 label have the same significance as shown in the preceding example. Bytes 50 and 51 indicate the buffer offset field which must be included in the block length. All other fields are recorded as ASCII spaces.

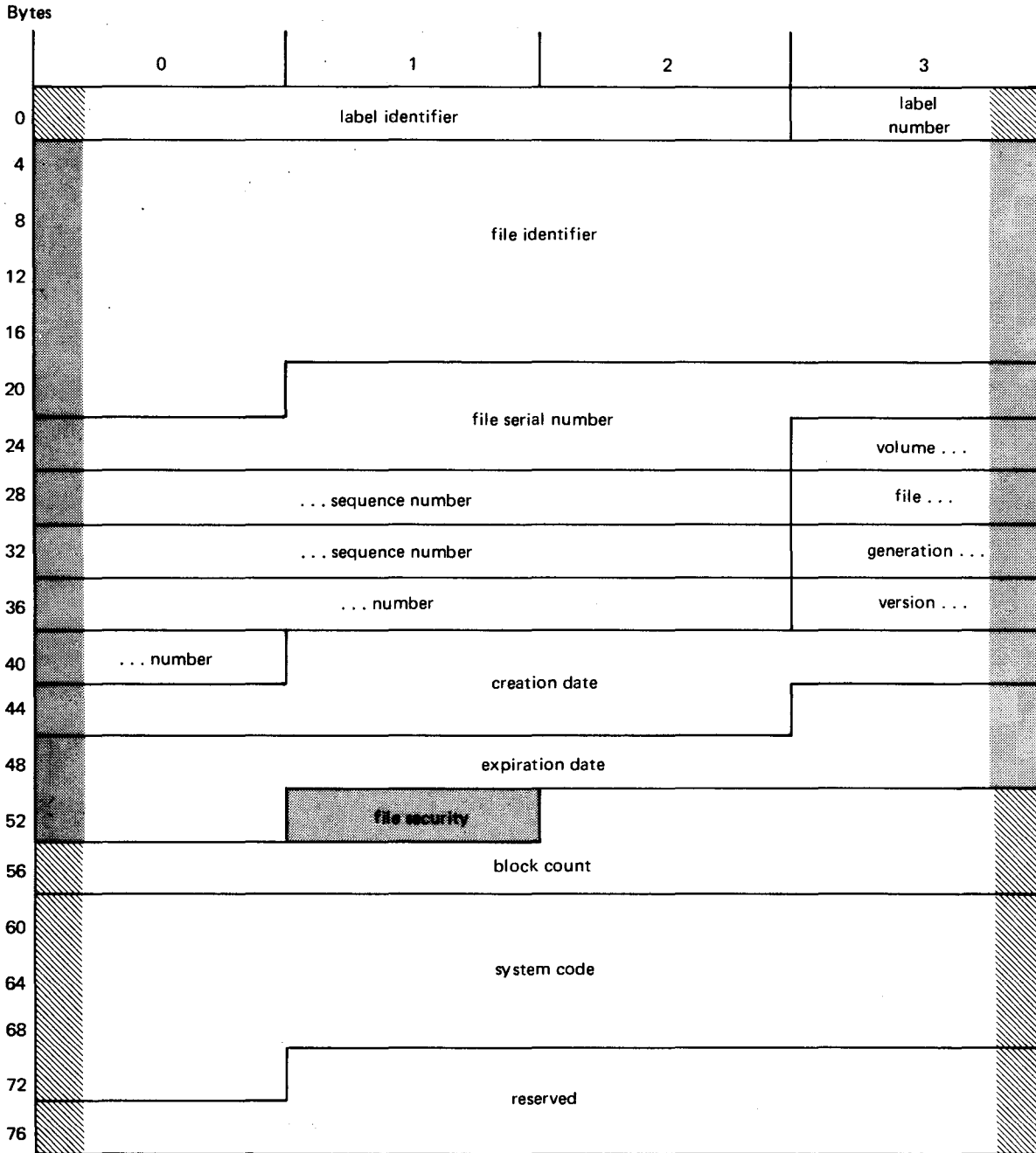
6.6.3. File Trailer Label Group

The file trailer label group comprises either of two pairs of labels, depending on whether the reel contains an end-of-file or an end-of-volume condition. In the first condition, the first label of the pair is the EOF1 label, in a format identical to the HDR1 label; the second label is the EOF2 label. Its format is identical to the HDR2 label. In the end-of-volume condition, these labels are the EOV1 and EOV2 labels; again, the formats of these labels are identical to their counterparts in the file header label group, HDR1 and HDR2.

The contents of the EOF1 and EOV1 labels are identical to the HDR1 label except for the label identifier, label number, and block count fields. The contents of the EOF2 and EOV2 labels are identical to the HDR2 label except for the label identifier and label number fields.

When you issue an OPEN macro instruction to an input tape file, with READ=BACK specified in the TCA macro instruction, the OPEN transient checks the fields in an EOF1 or EOV1 label against the values you have specified in the LBL job control statement. This processing is similar to that of the HDR1 label.

Figure 6—9 illustrates the format of the EOF1 or EOVI label, Table 6—4 summarizes the contents of its fields. Figure 6—10 illustrates the format of the EOF2 or EOVI2 label; Table 6—5 presents the contents of its fields.



LEGEND:



Generated by TSAT or reserved for system expansion.

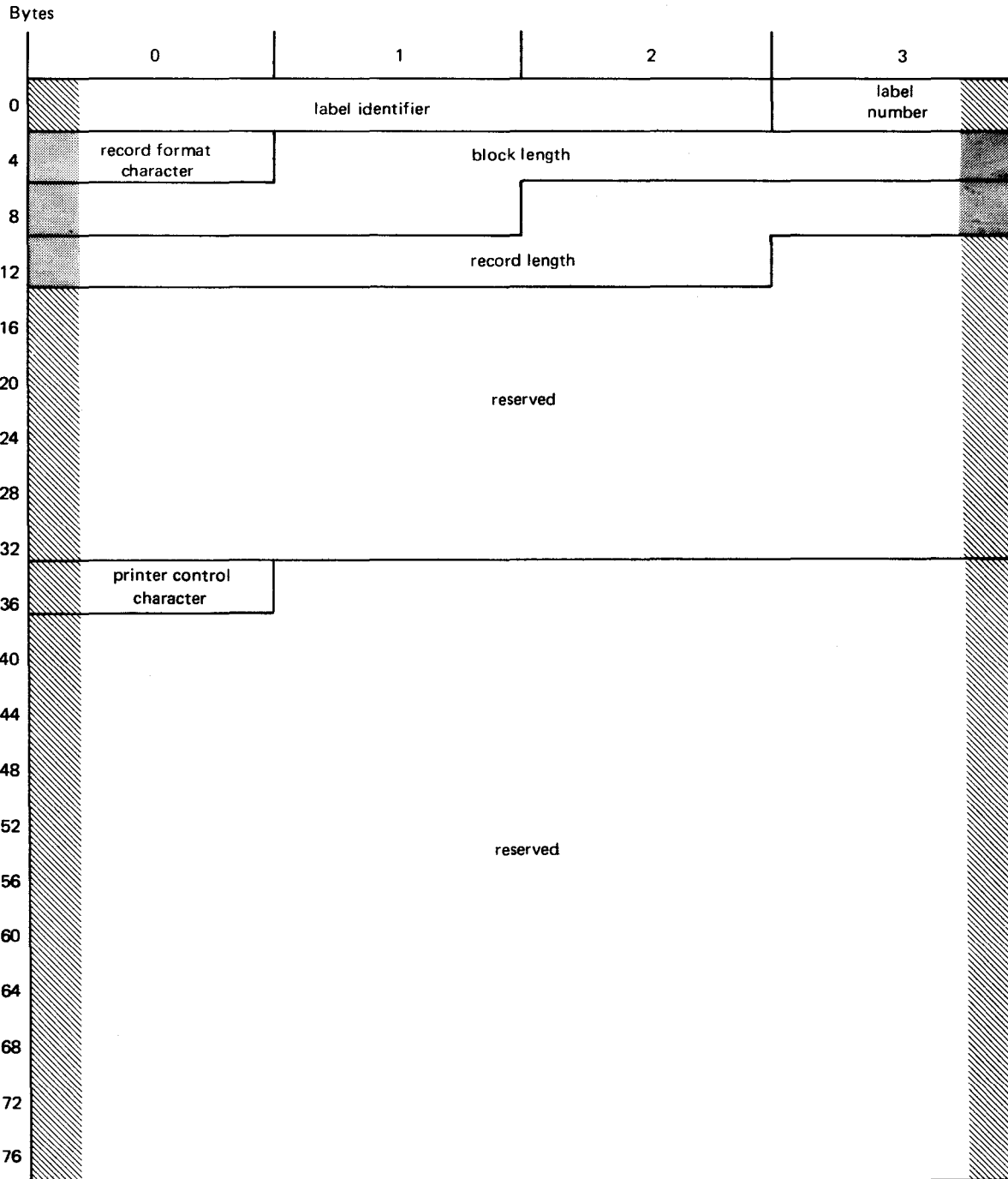


Written by TSAT from user-supplied data.

Figure 6—9. Tape File EOF1 and EOVI Label Formats for EBCDIC Tapes

Table 6-4 Tape File EOF1 and EOVI Labels, Field Description

| Field | Bytes | Description |
|------------------------------|-------|---|
| Label identifier | 0-2 | Indicates that this is a file trailer label; contains EOF for an end-of-file label, or EOVI for an end-of-volume label |
| Label number | 3 | Always 1 |
| File identifier | 4-20 | A 17-byte configuration that uniquely identifies the file. It may contain embedded blanks and is left-justified in the field if fewer than 17 bytes are specified. |
| File serial number | 21-26 | The same as the VSN of the VOL1 label for the first reel of a file or a group of multifile reels |
| Volume sequence number | 27-30 | The position of the current reel with respect to the first reel on which the file begins. This number is used with multivolume files only. |
| File sequence number | 31-34 | The position of this file with respect to the first file in the group |
| Generation number | 35-38 | The generation number of the file (0000-9999) |
| Version number of generation | 39-40 | The version number of a particular generation of a file |
| Creation date | 41-46 | The date on which the file was created, expressed in the form YYDDD and right-justified. The left-most position is blank. |
| Expiration date | 47-52 | The date the file may be written over or used as scratch, in the same form as the creation date |
| File security indicator | 53 | Reserved for file security indicator. Indicates whether additional qualifications must be met before a user program may have access to the file. 0 = No additional qualifications are required. 1 = Additional qualifications are required. |
| Block count | 54-59 | In the first file trailer label, indicates the number of data blocks: either in this file of a multifile reel, or on the current reel of a multivolume file. TSAT checks the block count for input files or writes the count for output files. |
| System code | 60-72 | Reserved for system code, the unique identification of the operating system that produced the file |
| (Reserved) | 73-79 | Reserved field, containing blanks (40 ₁₆) |



LEGEND:



Generated by TSAT or reserved for system expansion.



Written by TSAT from user-supplied data.

Figure 6-10. Tape File EOF2 and EO2 Label Formats for EBCDIC Tapes

Table 6-5. Tape File EOF2 and EO2 Labels, Field Description

| Field | Bytes | Description | | | | | | | | | | | | |
|---------------------------|--|--|-----------|---------|---|--|---|--------------|---|---------|---|-----------|---|--|
| Label identifier | 0-2 | Indicates that this is a file trailer label; contains EOF for an end-of-file label, or EO2 for an end-of-volume label | | | | | | | | | | | | |
| Label number | 3 | Always 2 | | | | | | | | | | | | |
| Record format character | 4 | <table border="1"> <thead> <tr> <th>Character</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>D</td> <td>Variable-length (ASCII), with length fields specified in decimal</td> </tr> <tr> <td>F</td> <td>Fixed-length</td> </tr> <tr> <td>S</td> <td>Spanned</td> </tr> <tr> <td>U</td> <td>Undefined</td> </tr> <tr> <td>V</td> <td>Variable-length (EBCDIC), with length fields specified in binary</td> </tr> </tbody> </table> | Character | Meaning | D | Variable-length (ASCII), with length fields specified in decimal | F | Fixed-length | S | Spanned | U | Undefined | V | Variable-length (EBCDIC), with length fields specified in binary |
| Character | Meaning | | | | | | | | | | | | | |
| D | Variable-length (ASCII), with length fields specified in decimal | | | | | | | | | | | | | |
| F | Fixed-length | | | | | | | | | | | | | |
| S | Spanned | | | | | | | | | | | | | |
| U | Undefined | | | | | | | | | | | | | |
| V | Variable-length (EBCDIC), with length fields specified in binary | | | | | | | | | | | | | |
| Block length | 5-9 | Five EBCDIC characters specifying the maximum number of characters per block | | | | | | | | | | | | |
| Record length | 10-14 | Five EBCDIC characters specifying the record length for fixed-length records. For any other record format, this field contains 0's. | | | | | | | | | | | | |
| (Reserved) | 15-35 | Reserved for future system use | | | | | | | | | | | | |
| Printer control character | 36 | One EBCDIC character indicating which control character set was used to create the data set. A Special (ASA) control character present D Device independent control character present M IBM control character present U SPERRY UNIVAC control character present | | | | | | | | | | | | |
| (Reserved) | 37-79 | Reserved for future system use | | | | | | | | | | | | |

6.7. TAPE VOLUME AND FILE ORGANIZATION

As was stated earlier, magnetic tape files processed by TSAT must observe certain label conventions. These were described in 6.6. Magnetic tape files must also observe conventions as to volume and file organization. The following paragraphs and figures describe the organization of files and volumes with respect to standard labeled, nonstandard labeled, and unlabeled files used with OS/3 tape sequential access method (SAM). Except where noted otherwise, these conventions also apply to tape files used with TSAT.

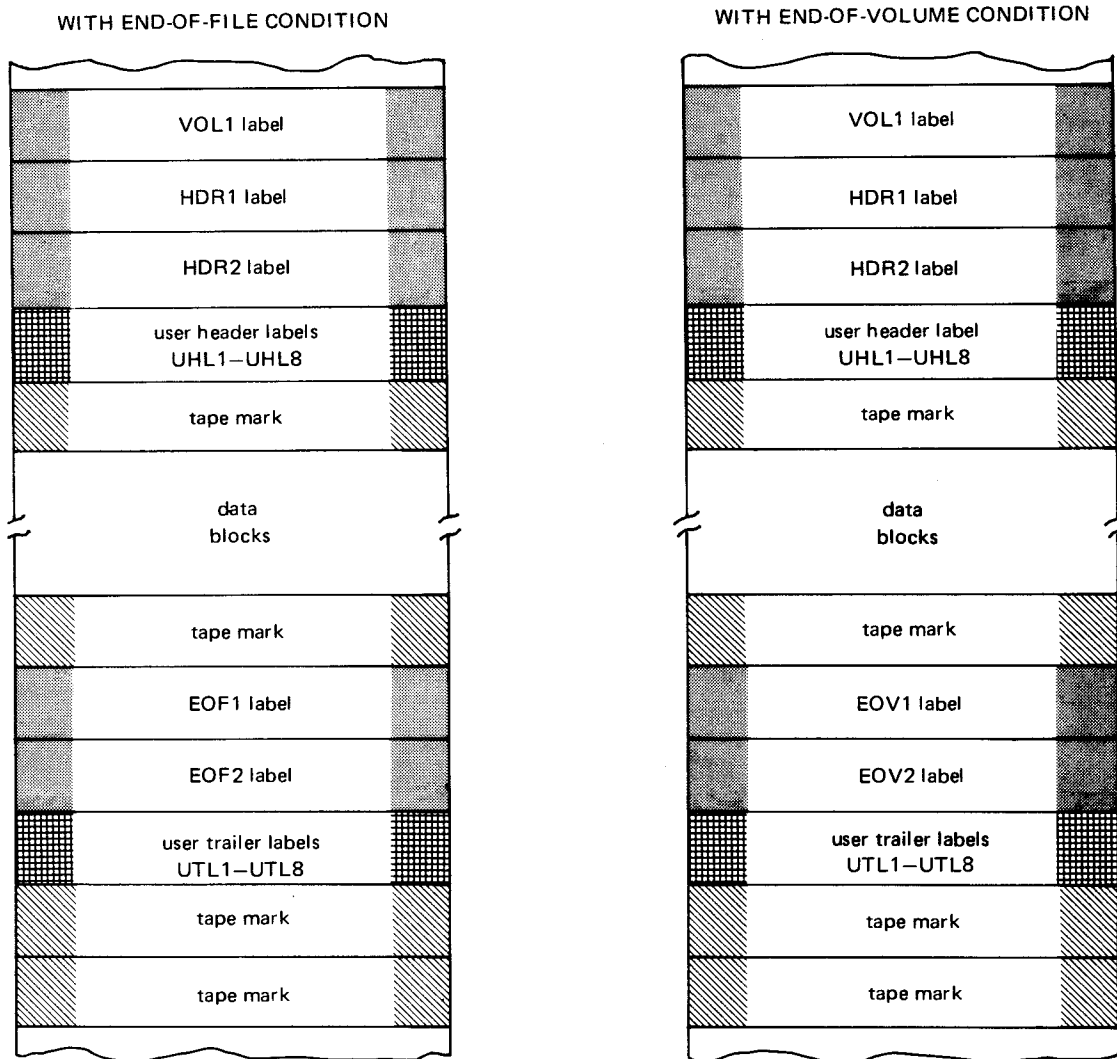
Remember that TSAT assumes only standard labeled files. TSAT bypasses user header labels, user trailer labels, and nonstandard labels. These labels are included in the following figures and descriptions only to show their relative location within the various volume organizations.

6.7.1. Standard Tape Volume Organization

A standard volume has standard labels, required tape marks, and is capable of being processed by the logical IOCS. Figures 6—11, 6—12, and 6—13 illustrate the reel organization for standard volumes with either an end-of-file (EOF) or end-of-volume (EOV) condition. The logical IOCS assumes that the labels appear in the order shown. A volume processed by TSAT will end in either an end-of-file or end-of-volume label group (EOF1 and EOF2 or EOV1 and EOV2) followed by two tape marks. The second tape mark signifies that no valid information follows.

User header (UHL) and user trailer (UTL) labels are optional. Tape SAM permits you to specify a special label handling routine to process these labels. If you do not specify such a routine, the optional labels are simply bypassed. However, with TSAT, you cannot specify your own label handling routine for optional labels. TSAT always bypasses these labels, and your program is not made aware of them.

On output operations no provision is made in TSAT for the creation of additional volume labels, file header labels, or file trailer labels. If these additional labels exist on input files, TSAT bypasses them.



LEGEND:





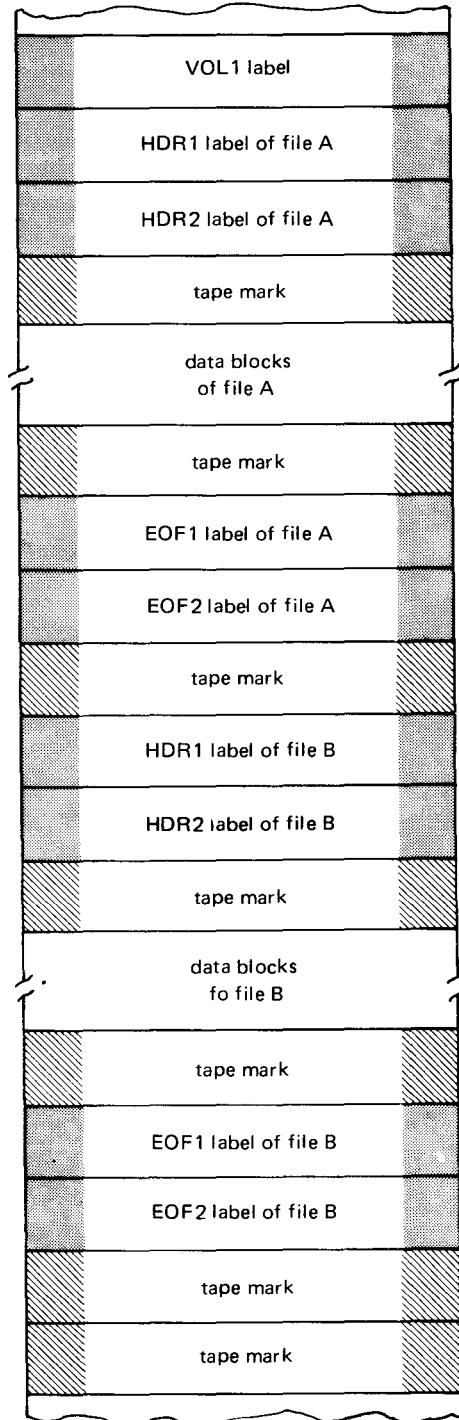
-  Contents supplied by user.
-  Required and generated by TSAT.
-  Generated by TSAT ; user supplies content for certain fields.
-  Generated by user at his option. Content is at user's option except for content of 4-byte label ID fields. User is limited to eight UHL and eight UTL. Bypassed by TSAT.

Figure 6-11. Reel Organization for EBCDIC Standard Labeled Tape Volumes Containing a Single File



LEGEND:



Content supplied by user.



Required and generated by TSAT.

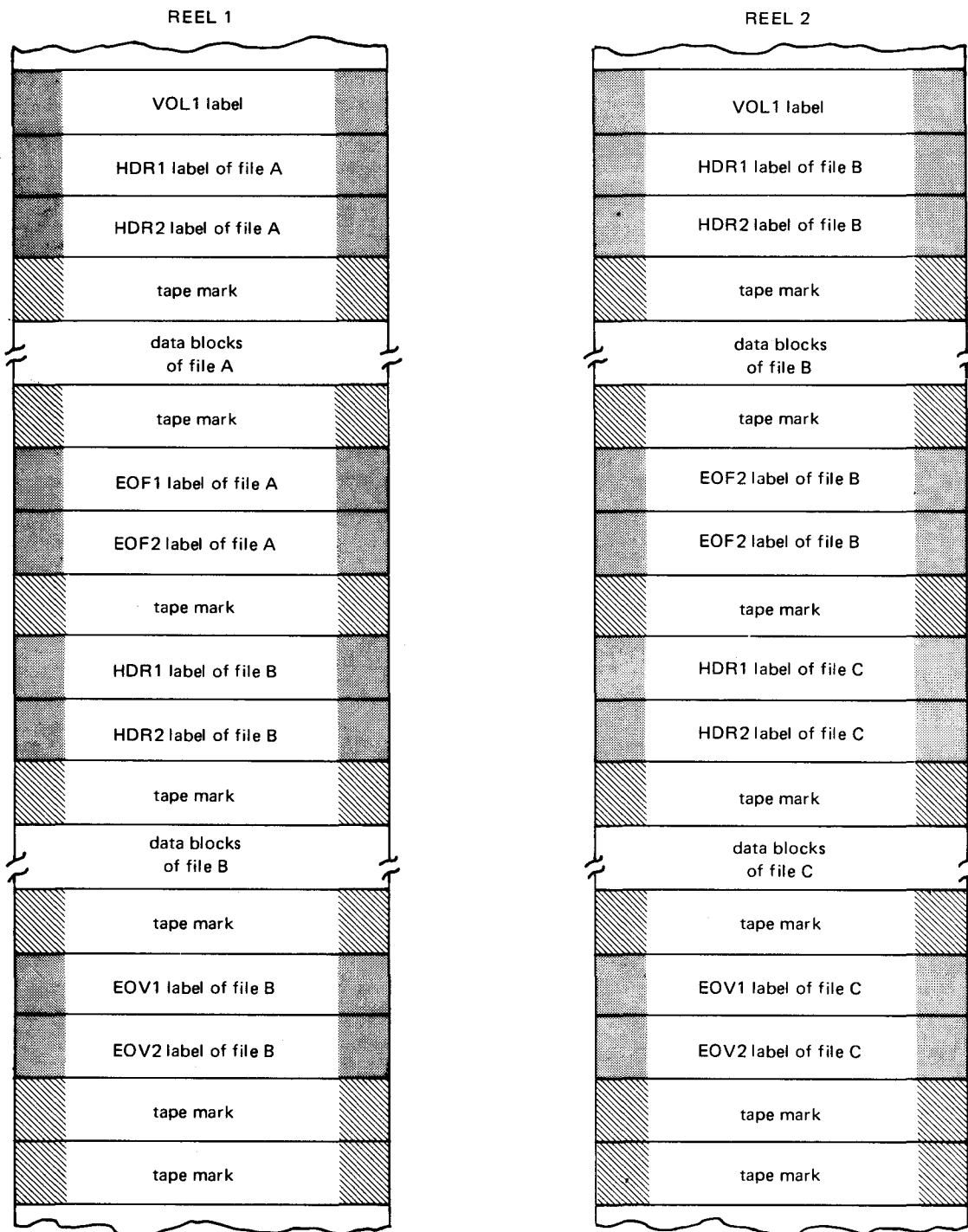


Generated by TSAT; user supplies content for certain fields.




NOTE:

Assume that file B completes on this volume.

Figure 6-12. Reel Organization for EBCDIC Standard Labeled Tape Volume: Multifile Volume With End-of-File Condition



LEGEND:

-  Content supplied by user.
-  Required and generated by TSAT.
-  Generated by TSAT; user supplies content for certain fields.

NOTE:

Assume that file C is not completed on reel 2, but carries over (like file B) onto another volume. If file C were completed on reel 2, its EOVI and EOVI labels shown here would be replaced with EOF1 and EOF2 labels.

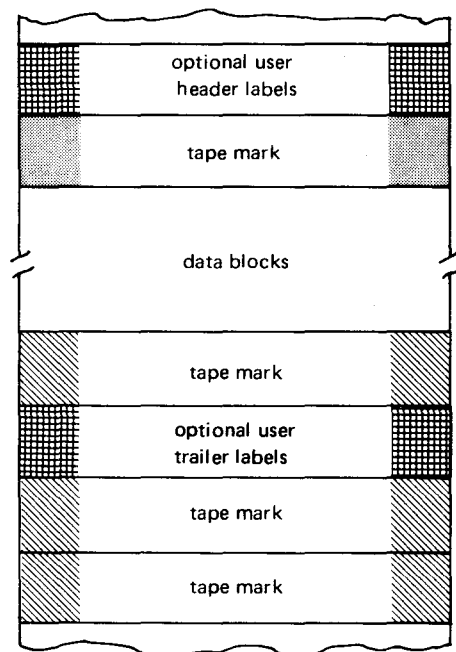
Figure 6-13. Reel Organization for EBCDIC Standard Labeled Tape Volumes: Multifile Volumes With End-of-Volume Condition

6.7.2. Nonstandard Tape Volume Organization

A nonstandard volume is any volume that has nonstandard labels and is capable of being processed by the logical IOCS. Figures 6—14 and 6—15 illustrate the reel organization for nonstandard volumes.

Nonstandard user header and trailer labels (UHL and UTL) are optional. These may be of any format, length, or number because they are handled by the user's label routine. When processing tapes using tape SAM, the address of the user's label handling routine to process nonstandard labels is usually specified, in which case the tape mark following the UHL may be omitted. It is required only if label checking is to be omitted or a read-backward operation is specified. If nonstandard labels appear on an input file but are not to be checked when the file is read, the user omits specifying the address of his label handling routine — but the tape mark must be present.

The tape mark following the data blocks is required and is written by logical IOCS, which also writes two required tape marks after the UTL, if they are present. If the optional UTL are not present, logical IOCS writes only one additional tape mark after the one following the data blocks. This second tape mark is always present when this file is the only file or the last file on the reel. It is overwritten by the next file to be written on a multifile volume.



LEGEND:





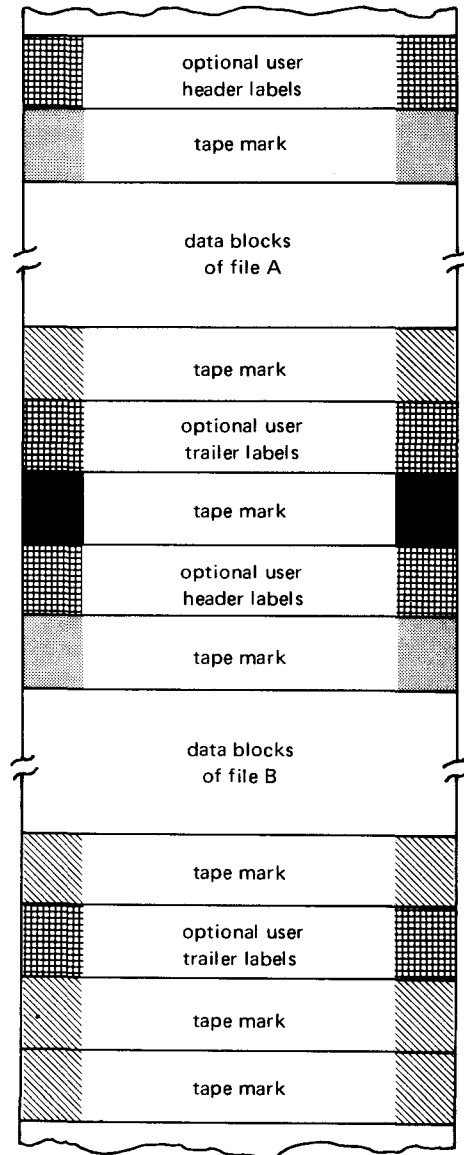
-  Contents supplied by user.
-  Required and generated by TSAT; only two tape marks follow data blocks if UTL are not present.
-  Generated by TSAT unless user specifies TPMARK=NO; required only if label checking is omitted or user specifies READ=BACK.
-  Presence, content, format, and number entirely at user's option. Bypassed by TSAT.

Figure 6—14. Reel Organization for EBCDIC Nonstandard Volume Containing a Single File



LEGEND:






-  Content supplied by user.
-  Required and generated by TSAT; only two tape marks follow data blocks of last file on volume if UTL are not present.
-  Generated by TSAT unless user specifies TPMARK=NO; required only if label checking is omitted or user specifies READ=BACK.
-  Presence, content, format, and number entirely at user's option. Bypassed by TSAT.
-  Always present; written by TSAT

Figure 6-15. Reel Organization for EBCDIC Nonstandard Multifile Volume

6.7.3. Unlabeled Tape Volume Organization

An unlabeled volume is any volume that has no labels and is capable of being processed by the logical IOCS. The user specifies FILABL=NO, or omits this parameter in the TCA macro instruction, to indicate an unlabeled volume or file. A tape mark is expected or written by logical IOCS preceding the data blocks unless the user has specified TPMARK=NO in the TCA macro instruction.

Figure 6—16 illustrates the reel organization for unlabeled volumes. The tape mark following the data blocks is required on both single-file and multifile volumes and is supplied by TSAT on output operations. A second tape mark is always written by TSAT following the last or only file on each volume and is overwritten by the next file to be written on a multifile volume.

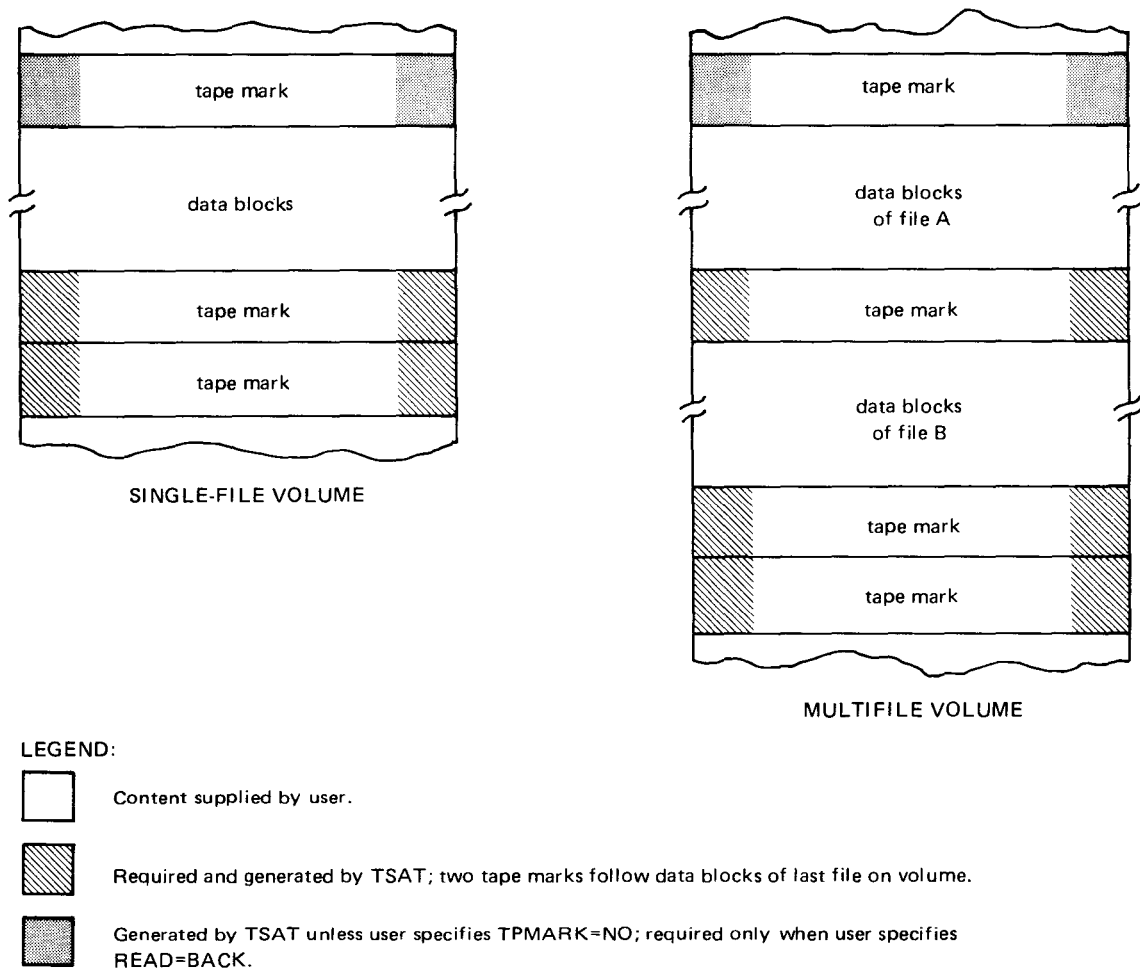


Figure 6—16. Reel Organization for Unlabeled EBCDIC Volumes

6.8. TAPE SAT FILE INTERFACE

Each file to be processed by TSAT must be predefined by two declarative macro instructions:

- **SAT**
Defines a TSAT magnetic tape file.
- **TCA**
Defines a tape control appendage.

The SAT macro instruction describes the physical characteristics of the file, and the TCA macro instruction describes the logical attributes of the file.

6.8.1. Define a Magnetic Tape File (SAT)

This is the DTF macro instruction for TSAT files. The assembler also accepts the name DTFPF; however, the name SAT is used here to avoid confusion between the DTF macro instruction for disk SAT files and tape SAT files.

Function:

The SAT macro instruction defines a magnetic tape file to be processed by SAT. It generates a DTF table in main storage containing the file name and operating and physical characteristics of your file that can be referenced by the system.

This is a declarative macro instruction and must not appear in a sequence of executable codes.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|--|
| filename | SAT | TCA=TCA-name [,CKPTREC=YES] [,ERROR=error-addr] [,FCB=YES] [,WAIT=YES] |

Label:

filename

Specifies the name used to identify the file. This is the same as the 8-character name in the LFD job control statement.

Keyword Parameter TCA:

TCA=TCA-name

Specifies the symbolic address of the TCA for the file. This name must be entered in the label field of the corresponding TCA macro instruction describing the tape control appendage.

Keyword Parameter CKPTREC:

CKPTREC=YES

Specifies that any checkpoint records occurring in an input tape file are to be bypassed by TSAT. In this case, your BLKSIZE specification in the TCA macro instruction must equal or exceed the length of a header or trailer label of the checkpoint set.

In OS/3 tape files, the first and last blocks of a checkpoint dump begin with the following:

```
//ΔCHKPTΔ//nnttCsss
```

where:

nn

Is the number, in binary, of image records plus control blocks, less 1, not including the header or trailer labels.

tt

Is the total number, in EBCDIC, of checkpoint records following the header label, including the trailer label; tt is 00 in a trailer label.

C

Is a constant, coded in EBCDIC as shown.

sss

Is the serial number of the checkpoint, in EBCDIC.

If omitted, any checkpoint records occurring are accepted as data by TSAT and your program must include the coding to recognize them.

Keyword Parameter ERROR:

ERROR=error-addr

Specifies the symbolic address of your error routine that receives control if an error occurs.

If omitted, the job is abnormally terminated if an error occurs.

Keyword Parameter FCB:

FCB=YES

Specifies that before issuing the OPEN macro instruction, you have placed the FCB for this file in the I/O area specified by the IOAREA1 keyword parameter of the TCA macro instruction associated with this file, instead of in the transient area where it is normally placed.

If omitted, the FCB, which controls file I/O, is placed into the transient area of main storage during file-open operations.

Keyword Parameter WAIT:

WAIT=YES

Specifies that TSAT is to issue the required WAITF macro instruction after each I/O function (GET, PUT). This initiates a waiting period to assure completion of the input or output operation and sets certain status bytes in the DTF table.

If omitted, you must issue a WAITF macro instruction after each I/O operation.

6.8.2. Define a Tape Control Appendage (TCA)

Function:

The TCA macro instruction defines the logical attributes of a magnetic tape file to be processed by TSAT. It generates a tape control appendage to the DTF table for the file.

This is a declarative macro instruction and must not appear in a sequence of executable code.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|--|
| TCA-name | TCA | IOAREA1=area-name ,BLKSIZE=n [,BKNO=YES] [,CLRW= { NORWD } { RWD }] [,EOFADDR=end-of-data-addr [,FILABL= { STD } { NSTD } { NO }] [,LBLK=n] [,OPRW=NORWD] [,READ= { FORWARD } { BACK }] [,REWIND= { UNLOAD } { NORWD }] [,TPMARK=NO] [,TYPEFLE=OUTPUT] |

Label:

TCA-name

Specifies the symbolic address of the TCA table generated by this macro instruction. This must be the same name specified in the TCA parameter of the SAT macro instruction for this file.

Keyword Parameter IOAREA1:

IOAREA1=area-name

Specifies the symbolic address of an input/output area in main storage where the blocks are to be processed. The size of this area is specified in the BLKSIZE keyword parameter.

When processing block numbered tapes (BKNO=YES), you must reserve a 4-byte storage area immediately preceding your input/output area for supervisor processing of the block number. The 4-byte block number area and the input/output area must be aligned on a full-word boundary. Do not include these four bytes as part of the IOAREA1 specification.

Keyword Parameter BLKSIZE:

BLKSIZE=n

Specifies the size in bytes of the area in main storage named by the IOAREA1 keyword parameter.

When processing block numbered tapes (BKNO=YES), you must reserve a 4-byte storage area immediately preceding your input/output area. Do not include these four bytes as part of the BLKSIZE specification.

If you are reading input tapes backward (READ=BACK), your BLKSIZE specification must accommodate the largest block on tape, otherwise the data at the beginning of the block may be lost. If the data is truncated on a backward read of a block numbered file, the block number field will be lost and incorrect positioning of the tape may result.

Keyword Parameter BKNO:

BKNO=YES

Specifies that you have reserved a 4-byte storage area, aligned on a full-word boundary, immediately preceding your input/output area. Do not include these four bytes as part of either the IOAREA1 specification or the BLKSIZE specification. Processing of block numbered tape files is described in 6.10.

Keyword Parameter CLRW:

CLRW=NORWD

Specifies that a tape is not to be rewound when a file is closed.

CLRW=RWD

Specifies that a tape is to be rewound without interlock when a file is closed.

If omitted, the tape is rewound with interlock when a file is closed, which causes the tape to be unloaded from the take-up reel.

Keyword Parameter EOFADDR:

EOFADDR=end-of-data-addr

Specifies the symbolic address of your end-of-data routine to which TSAT transfers control when the tape mark following the last block of input data is sensed. This keyword parameter is required for all input files. The optional spelling, EDDADDR, of this parameter is also acceptable.

Keyword Parameter FILABL:

FILABL=STD

Specifies that a tape contains standard labels.

FILABL=NSTD

Specifies that a tape contains nonstandard labels. These labels are not checked by TSAT. No provision is made in TSAT to create this type of label.

FILABL=NO

Specifies that labels are undefined or absent.

Keyword Parameter LBLK:

LBLK=n

Specifies the number of physical blocks (of the length specified in the BLKSIZE keyword parameter) comprising a logical block. The entry for n specifies the number of contiguous buffers supplied at the address specified by the IOAREA1 keyword parameter. Use this parameter when you want to act upon more than one physical block to construct one logical block.

If omitted, one physical block comprises one logical block (LBLK=1).

Keyword Parameter OPRW:

OPRW=NORWD

Specifies that a tape is not to be rewound before labels are checked during the processing of the OPEN macro instruction. When read-backward processing is specified, NORWD is assumed. This keyword parameter must not be used if the REWIND keyword parameter is specified. If both are used, they are mutually exclusive.

If omitted, the tapes are rewound at open time.

Keyword Parameter READ:

READ=FORWARD

Specifies that an input file is to be read forward.

READ=BACK

Specifies that an input file is to be read backward. If this is specified, you are limited to a single volume file. Also, your BLKSIZE specification must accommodate the largest block on tape.

If omitted, read forward is assumed.

Keyword Parameter REWIND:

REWIND=UNLOAD

Specifies that a tape is to be rewound to load point at open time, and rewound with interlock at close time or when an end-of-volume condition is encountered.

REWIND=NORWD

Specifies that a tape is not to be rewound at open time, and is to be positioned between the two file marks at close time.

If omitted, the OPRW or CLRW parameters are selected.

Keyword Parameter TPMARK:

TPMARK=NO

Specifies that, for output files with nonstandard labels or no labels, logical IOSC is not to write the tape mark that normally separates header labels from data. In this case, it is your responsibility to distinguish between header labels and data. In a multifile reel environment, this keyword parameter should not be used for files that are to be processed backward.

Keyword Parameter TYPEFLE:

TYPEFLE=OUTPUT

Specifies that this is an output file.

If omitted, an input file is assumed.

6.9. CONTROLLING YOUR TAPE FILE PROCESSING

There are six imperative macro instructions available for controlling your tape file processing using TSAT:

- OPEN
Opens a tape file.
- GET
Gets the next logical block.
- PUT
Outputs the next logical block.
- WAITF
Waits for block transfer.
- CNTRL
Controls tape unit functions.
- CLOSE
Closes a tape file.

6.9.1. Open a Tape File (OPEN)

Function:

After the file has been defined by the SAT and TCA declarative macro instructions, the OPEN macro instruction must be issued to initialize the file before any other access can be made. This macro instruction validates the DTF and TCA tables and performs any required tape positioning functions.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|---|
| [symbol] | OPEN | { filename-1 [, ..., filename-n] } (1) |

Positional Parameter 1:

filename-1

Specifies the symbolic address of the SAT macro instruction in the program corresponding to the file to be opened.

(1)

Indicates that register 1 has been preloaded with the address of the SAT macro instruction.

Positional Parameter n:

filename-n

Successive entries specify the symbolic addresses of the SAT macro instructions in the program corresponding to the additional files to be opened.

Use this form (for example, OPEN FILE1, FILE2) when more than one lockable file is to be accessed by a single task. This opens all the files named and applies the required read or write locks at the same time. In this way, you can avoid the possibility of two jobs locking each other out with each one waiting for the other to give up its file. The operator would then have to cancel one of the jobs to remove the stalemate and continue processing.

6.9.2. Get Next Logical Block (GET)

Function:

The GET macro instruction reads a logical block from tape into main storage and makes it accessible for processing. The address into which the data is read is specified in the associated TCA macro instruction by the keyword parameter IOAREA1.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|---|
| [symbol] | GET | { filename } (1) , { TCA-name } (0) |

Positional Parameter 1:

filename

Specifies the symbolic address of the SAT macro instruction in the program corresponding to the file being read.

(1)

Indicates that register 1 has been preloaded with the address of the SAT macro instruction.

Positional Parameter 2:

TCA-name

Specifies the symbolic address of the TCA macro instruction associated with the partition to be accessed.

(0)

Indicates that register 0 has been preloaded with the address of the TCA macro instruction.

6.9.3. Output Next Logical Block (PUT)

Function:

The PUT macro instruction writes a logical block from main storage to tape. The main storage address from which the data is written is specified in the associated TCA macro instruction by the keyword parameter IOAREA1.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|-------------------------------------|
| [symbol] | PUT | { filename } (1) , { TCA-name } (0) |

Positional Parameter 1:

filename

Specifies the symbolic address of the SAT macro instruction in the program corresponding to the file being written.

(1)

Indicates that register 1 has been preloaded with the address of the SAT macro instruction.

Positional Parameter 2:

TCA-name

Specifies the symbolic address of the TCA macro instruction associated with the partition to be written.

(0)

Indicates that register 0 has been preloaded with the address of the TCA macro instruction.

6.9.4. Wait For Block Transfer (WAITF)

Function:

The WAITF macro instruction ensures that a command initiated by a preceding GET or PUT macro instruction has been completed. When completed, the error status field contains the error status information pertaining to the I/O request. It is your responsibility to check these bits, which are in bytes 50 and 51 of the DTF table.

If the keyword parameter WAIT=YES was not specified in the SAT macro instruction, the WAITF macro instruction must be issued after a GET or PUT macro instruction and before another imperative macro instruction is issued for that file.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|---------------------|
| [symbol] | WAITF | { filename } (1) |

Positional Parameter 1:

filename

Specifies the symbolic address of the SAT macro instruction in the program corresponding to the file being accessed.

(1)

Indicates that register 1 has been preloaded with the address of the SAT macro instruction.

6.9.5. Control Tape Unit Functions (CNTRL)

Function:

This macro instruction initiates nondata operations on a tape unit. All tape control functions may be issued whether or not the file is open. Do not issue a WAITF macro instruction following a CNTRL macro instruction.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|---------------------------|
| [symbol] | CNTRL | { filename } ,code (1) |

Positional Parameter 1:

filename

Specifies the symbolic address of the corresponding SAT macro instruction in the program.

(1)

Indicates that register 1 has been preloaded with the address of the SAT macro instruction.

Positional Parameter 2:

code

Is a mnemonic 3-character code specifying the tape unit function to be performed:

- BSF Backspace to tape mark*
- BSR Backspace to interrecord gap*
- ERG Erase gap (writes blank tape)
- FSF Forward space to tape mark*
- FSR Forward space to interrecord gap*
- REW Rewind tape
- RUN Rewind tape with interlock (unloads tape)
- WTM Write tape mark

6.9.6. Close a Tape File (CLOSE)

Function:

The CLOSE macro instruction performs the required termination operations for a file; for example, construction of the EOF label group. Once the CLOSE macro instruction has been issued for a file, only the OPEN macro instruction may reference that file.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|---|
| [symbol] | CLOSE | { filename-1 [, ..., filename-n] (1) } |

*Applies only to input files.

Positional Parameter 1:

filename-1

Specifies the symbolic address of the SAT macro instruction in the program corresponding to the file to be closed.

(1)

Indicates that register 1 has been preloaded with the address of the SAT macro instruction.

Positional Parameter n:

filename-n

Successive entries specify the symbolic addresses of the SAT macro instructions in the program corresponding to the additional files to be closed.

6.10. BLOCK NUMBER PROCESSING

TSAT can process magnetic tapes with or without block numbers. The use of block numbers reduces the possibility of incorrect tape positioning and, therefore, incorrect tape processing. This is especially helpful for error recovery on read and write commands and for restarting at a checkpoint.

Processing of block numbered tapes for TSAT files will be executed by physical IOCS. The general requirements and processing are the same as detailed for physical IOCS in 4.4.1 to 4.4.4. Some of these are noted here for convenience.

- ↓
■ When the block numbering capability is being used, all blocks on tape except tape marks will include a 3-byte block number field as the first three bytes of the block. This 24-bit block number field is composed of a 4-bit tape mark counter and a 20-bit block number counter. Physical IOCS uses both of these counters when reading and writing block numbered tapes. The format of the tape block number field is shown in Figure 4-9.
- ↑
■ The first block on tape that is not a tape mark will contain a block count of 1 plus the number of tape marks preceding it.
- Block numbers are incremented sequentially by 1. All label, data, and checkpoint blocks are counted and numbered. Tape marks are counted, but no number is written.
- For both EBCDIC and ASCII tapes, the 3-byte block number field is added to a standard label immediately preceding the label identifier (VOL1, HDR1, etc.), making the label 83 bytes long. The 83-byte ASCII label is nonstandard for information interchange. Tape label formats for block numbered EBCDIC tapes are shown in Figures 6-17 through 6-21.
- Block number processing will be exactly the same for both EBCDIC and ASCII tape files.
- Block numbers will be volume dependent and file independent. If a volume contains more than one file, the block count is continued from the preceding file on the volume and the blocks are consecutively numbered to the end of the tape.

- Files on a volume and volumes in a multivolume file must be all numbered or all unnumbered, not mixed.
- The 7-track odd parity tapes operating in convert mode may be block numbered if the block size is a multiple of 3.

The PUB trailer for a block numbered tape file will contain an expected block number. This number will reflect the next block number anticipated in a forward read and will be adjusted accordingly for backward reads. When the tape is read in either direction, the block number read from tape is stored in the PUB trailer and compared with the expected block number. If there is no discrepancy (and no other errors) control is returned to the user program. If there is a discrepancy, physical IOCS attempts to find the correct block by moving the tape backward or forward the number of blocks implied by the discrepancy. If the correct block is found, control is returned to the user. If the correct block cannot be found, the tape is left positioned where it was on the last attempt and an error message is sent to the console.

6.10.1. Facilities Required for Block Number Processing

To process block numbered tape files, three conditions (called preliminary conditions) are required:

1. So that the generated supervisor can process both numbered and unnumbered tapes, you must operate with a supervisor configured to process block numbered tapes.
2. You must reserve a full-word aligned, 4-byte storage area immediately preceding your input/output area for supervisor processing of the block number. Do not include these four bytes as part of either the address or the length specifications (IOAREA and BLKSIZE keyword parameters of the TCA declarative macro instruction).
3. You must indicate to TSAT that you have reserved the 4-byte block number area by specifying BKNO=YES in the TCA macro instruction (6.8.2).

If these three preliminary conditions exist, you may then control block number processing through either job control (JCL) or automatic volume recognition (AVR). This permits you to leave the 4-byte storage area and the BKNO parameter in your program even though you may at times be processing unnumbered tapes.

6.10.2. Specifications for Block Number Processing

Several factors determine when and how block number processing is employed. If a tape is not at load point when the file is opened, the file will be handled according to the specifications existing when the tape was opened at load point. Therefore, you cannot have both numbered and unnumbered files on the same volume.

If a tape is at load point when it is opened, processing will proceed as described on the following pages.

The various methods of tape file processing can be divided into two categories: processing with tape initialization, and processing without tape initialization. These will be referred to simply as initialized or noninitialized processing.

6.10.2.1. Initialized Processing

Initialized processing includes:

- TPREP utility routine processing, described in the system service programs user guide, UP-8062 (current version);
- processing output files with standard labels (FILABL=STD specified in the TCA macro instruction) and PREP specified in the VOL job control statement; or
- processing input or output files with nonstandard labels (FILABL=NSTD) or no labels (FILABL=NO specified in the TCA macro instruction).

For initialized processing, you control the presence or absence and the processing of block numbers by the first parameter of the VOL job control statement as follows:

| You Specify | Preliminary Conditions | Result |
|-------------|------------------------|----------------------------|
| Nothing | All present | Block number processing |
| | Some missing | No block number processing |
| N | Ignored | No block number processing |

6.10.2.2. Noninitialized Processing

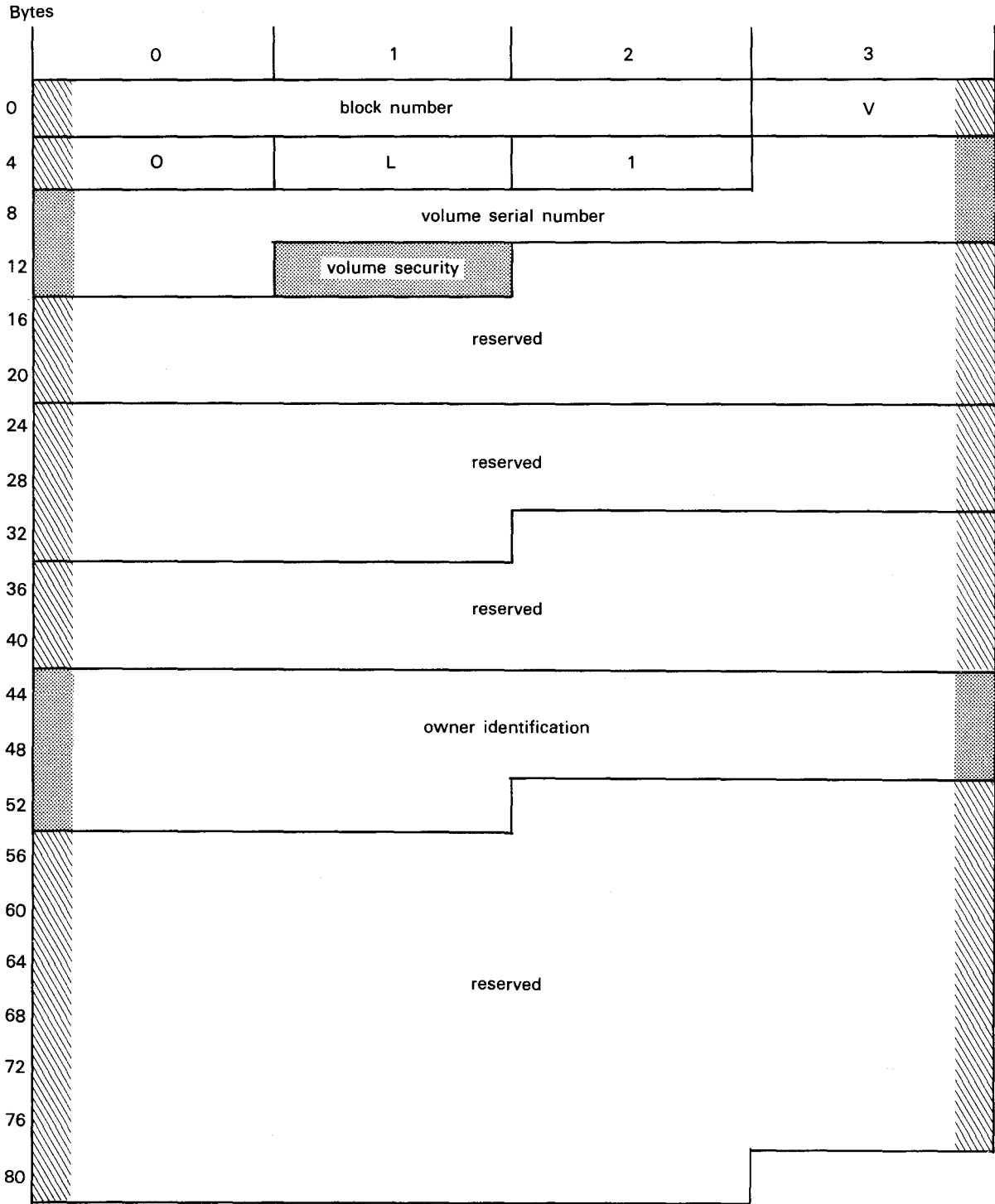
Noninitialized processing includes:

- processing output files with standard labels (FILABL=STD specified in the TCA macro instruction), but without PREP specified in the VOL job control statement; or
- processing input files with standard labels (FILABL=STD specified in the TCA macro instruction).

For noninitialized processing, TSAT ignores the first parameter of the VOL job control statement. Instead, the specification is obtained from the tape content (which was detected by AVR), as follows:

| Tape Content | Preliminary Conditions | Result |
|------------------|------------------------|----------------------------|
| Block numbers | All present | Block number processing |
| | Some missing | No block number processing |
| No block numbers | Ignored | No block number processing |

For processing of multivolume files, you must ensure that all volumes have (or do not have) block numbers. You cannot mix numbered and unnumbered volumes within a file.



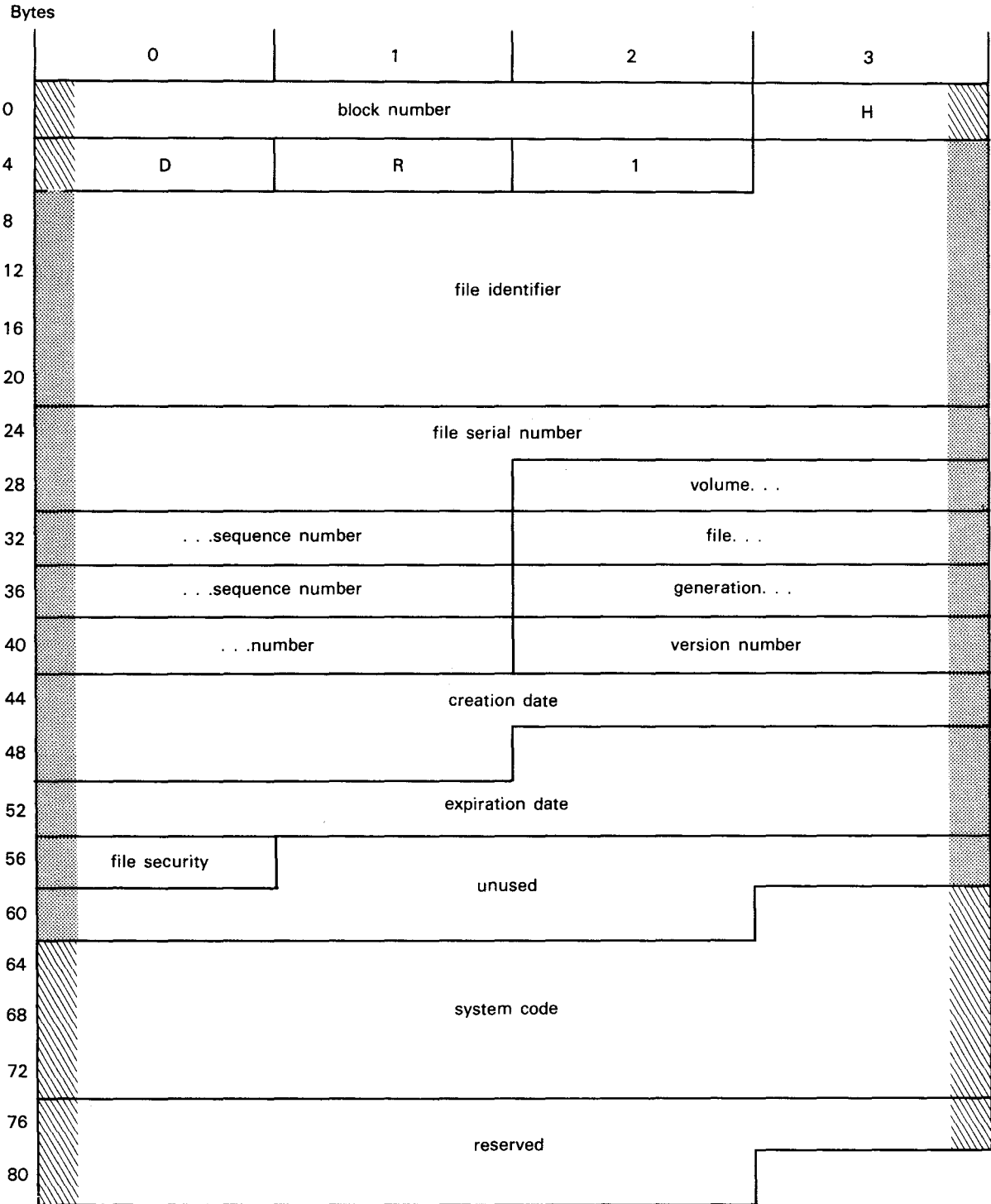
LEGEND:

- Generated by TSAT or reserved for system expansion.
- Written by TSAT from user-supplied data.

NOTE:

The first three bytes (bytes 0-2) of the tape file label contain a 24-bit block number field. The contents of the remainder of the VOL1 label are the same as described in Table 6-1, except that each field is offset three bytes.

Figure 6-17. Tape Volume 1 (VOL1) Label Format for an EBCDIC Volume With Block Numbers



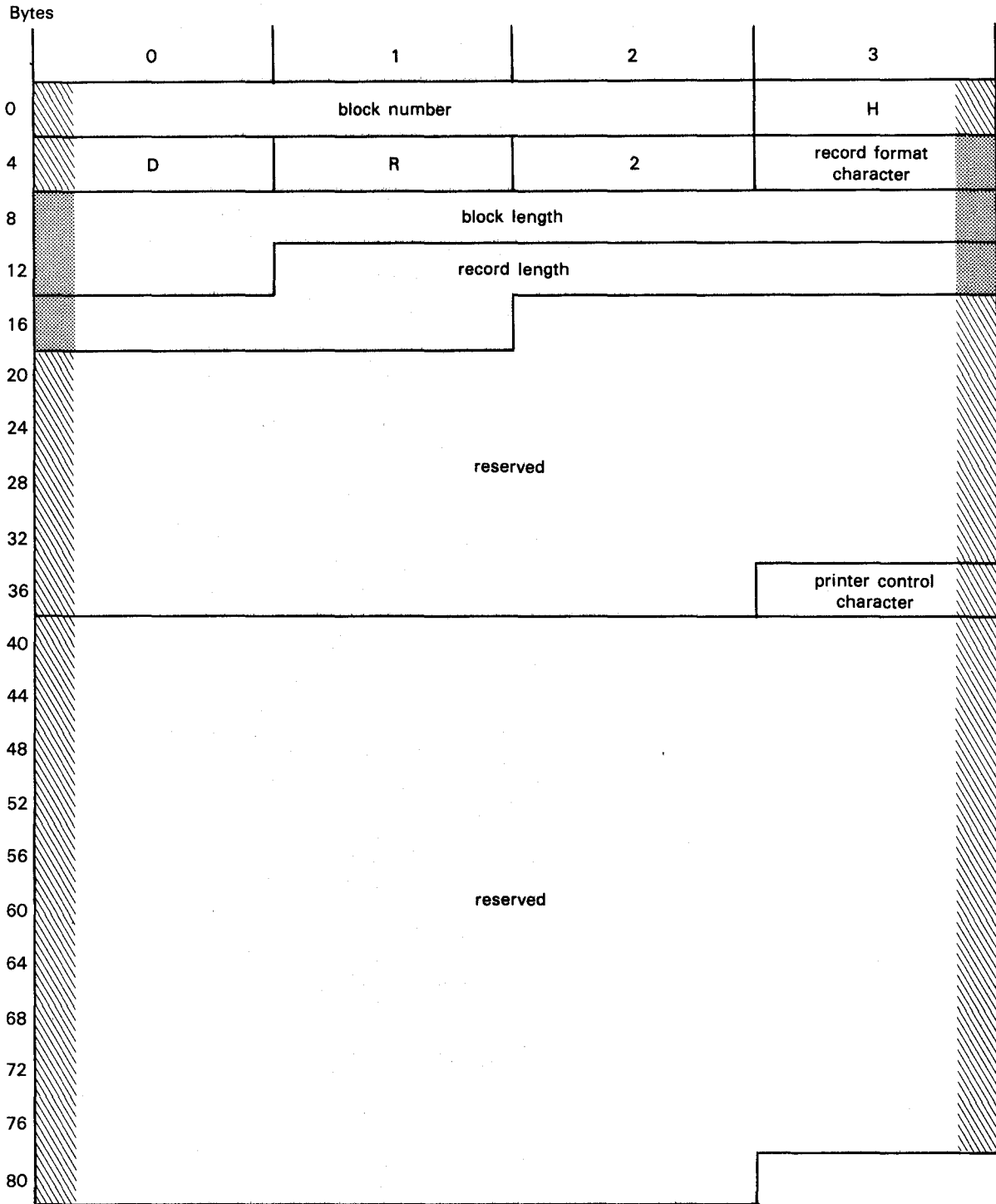
LEGEND:

- Generated by TSAT or reserved for system expansion.
- Written by TSAT from user-supplied data.

NOTE:

➔ The first three bytes (bytes 0—2) of the tape file label contain a 24-bit block number field. The contents of the remainder of the HDR1 label are the same as described in Table 6—2, except that each field is offset three bytes.

Figure 6—18. First File Header Label (HDR1) Format for an EBCDIC Tape Volume With Block Numbers



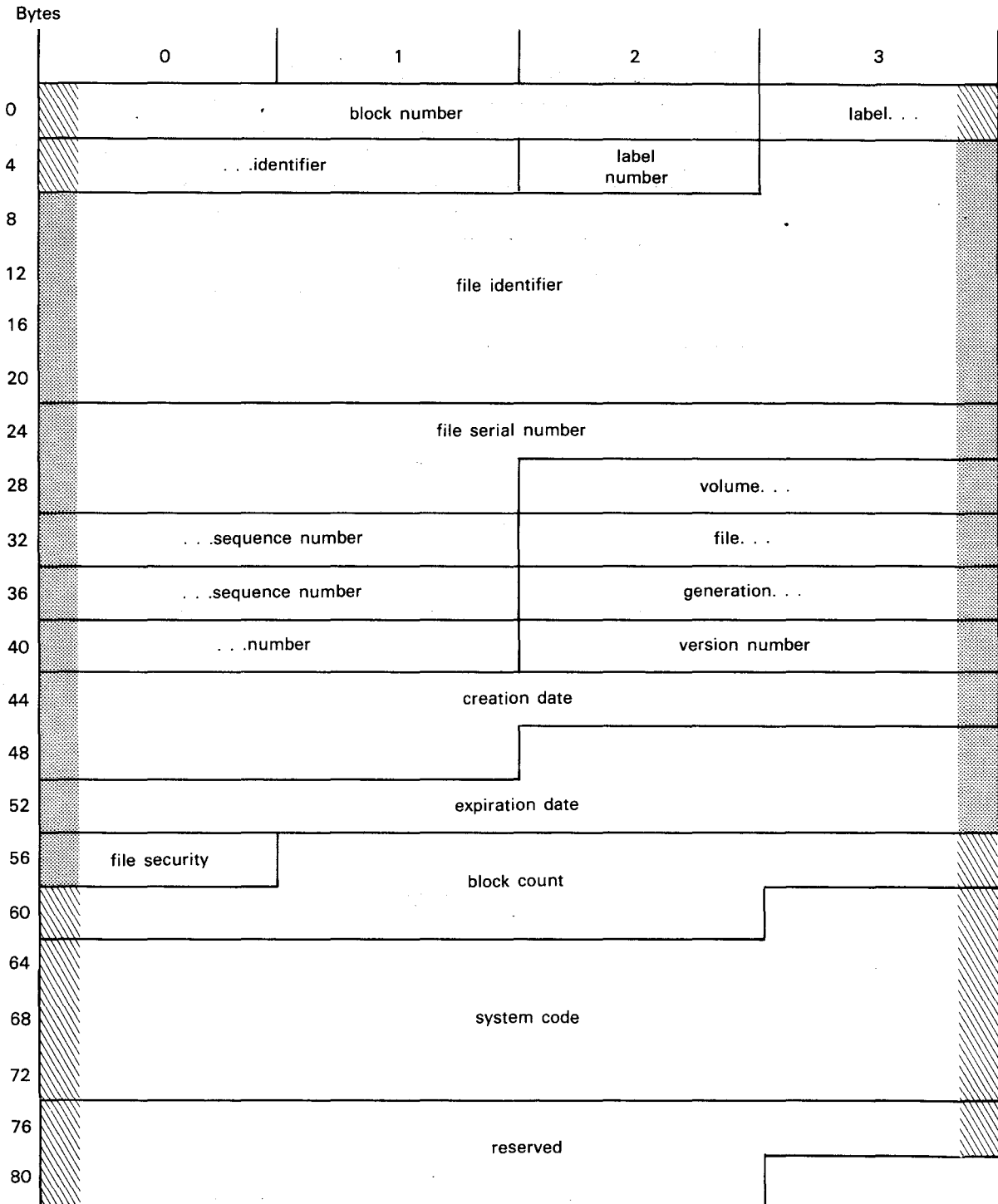
LEGEND:

- Generated by TSAT or reserved for system expansion.
- Written by TSAT from user-supplied data.

NOTE:

The first three bytes (bytes 0—2) of the tape file label contain a 24-bit block number field. The contents of the remainder of the HDR2 label are the same as described in Table 6—3, except that each field is offset three bytes.

Figure 6—19. Second File Header Label (HDR2) Format for an EBCDIC Tape Volume With Block Numbers



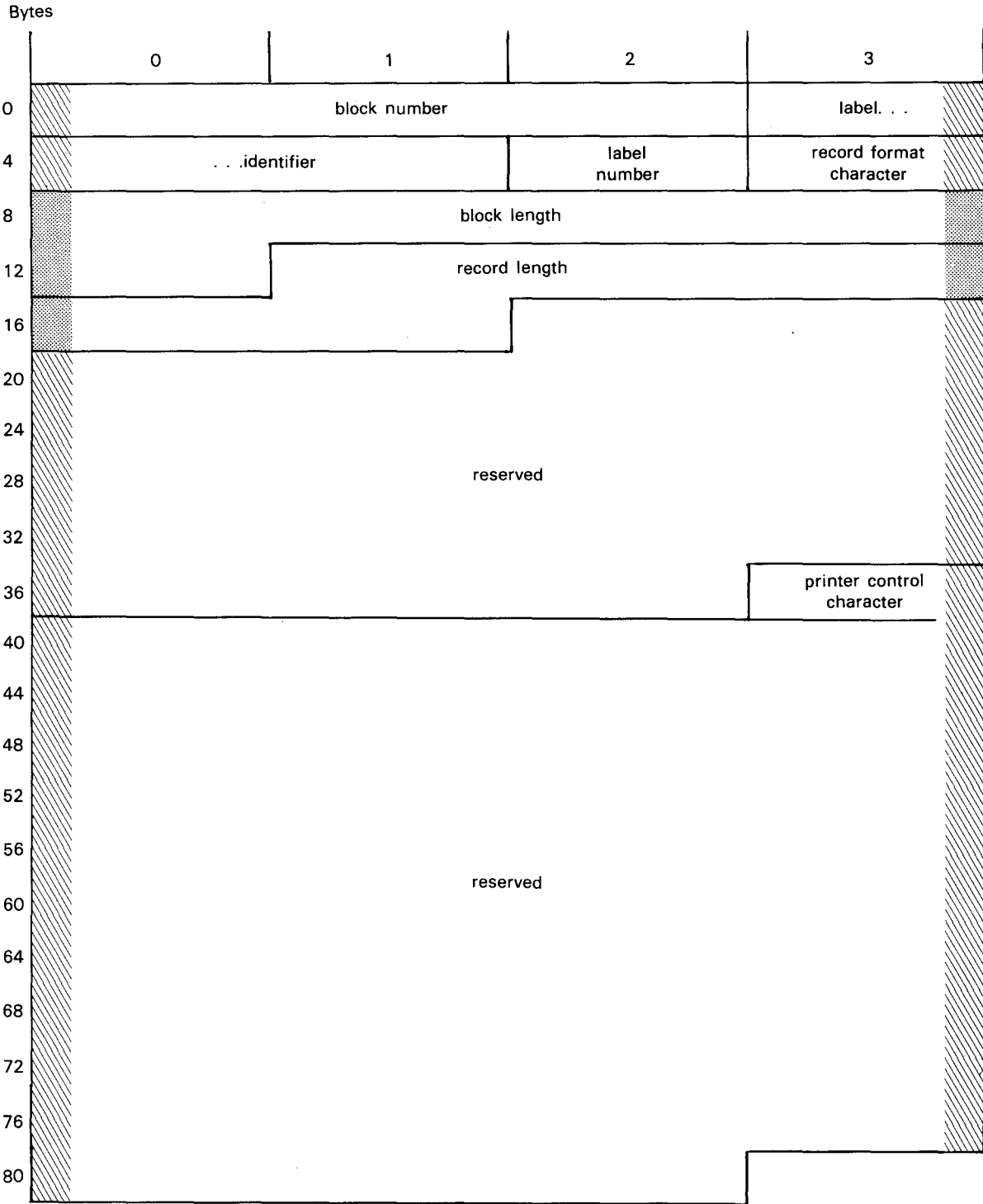
LEGEND:

- Generated by TSAT or reserved for system expansion.
- Written by TSAT from user-supplied data.



NOTE:

→ The first three bytes (bytes 0—2) of the tape file label contain a 24-bit block number field. The contents of the remainder of the EOF1 and EOVI labels are the same as described in Table 6—4 except that each field is offset three bytes.

Figure 6—20. Tape File LEOF1 and EOVI Label Formats for Block Numbered EBCDIC Files



LEGEND:

-  Generated by TSAT or reserved for system expansion.
-  Written by TSAT from user-supplied data.

NOTE:

The first three bytes (bytes 0—2) of the tape file label contain a 24-bit block number field. The contents of the remainder of the EOF2 and EO2 labels are the same as described in Table 6—5, except that each field is offset three bytes.

Figure 6—21. Tape File EOF2 and EO2 Label Formats for Block Numbered EBCDIC Files



PART 3. MULTITASKING



7. Multitasking

7.1. GENERAL

7.1.1. Multijobbing and Multitasking

The SPERRY UNIVAC 90/30 Data Processing System can concurrently process from one to seven jobs, each job consisting of one or more job steps (programs) which are executed serially. A job step will also have one or more tasks which may be executed concurrently. This capability allows you greater flexibility in attaining maximum use of the system's resources.

Multijobbing consists of scheduling multiple jobs (up to seven) for concurrent execution. The allocation of processor time to these jobs is based on a system switch list which contains information regarding task priorities, synchronization, and I/O utilization. While one job is awaiting the completion of an external event (such as completion of an I/O request), the operating system activates another job that is ready to ensure optimum utilization of the processor's capabilities. Since the majority of programs require support other than processing instructions, multijobbing provides an effective method for you to reduce processor idle time and increase system productivity (throughput).

Multitasking is the concurrent execution of multiple tasks. Because every job has at least one task to which control of the processor is dispatched, the term multitasking is sometimes applied (from the point of view of the task switcher within the supervisor) to the concurrent execution of several jobs each having one task. However, multitasking, as used here, refers to the concurrent execution of several tasks asynchronously within a given job step. Multitasking enables you to overlap processing with external occurrences within a program to obtain maximum throughput in the same manner as the system achieves optimum utilization using multijobbing.

7.1.1.1. Primary Task

Every job step submitted to OS/3 is established as a primary task. A task is the lowest viable entity that can compete for processor time. OS/3 permits up to 256 tasks per job. The switch list has the capacity to allow you to specify up to 60 levels of processing priority for tasks. The maximum number of task priority levels that the supervisor will recognize is established at system generation time. The technical limit is 60; however, a more practical limit of 3 to 15 is sufficient to achieve a high degree of processor utilization. When a task is interrupted to perform external processing (external to the instruction processor), it frees the processor and OS/3 searches the switch list for the highest priority task that is not waiting for an external event to be completed. This task could be in the same job or it could be from any other job currently being processed.

7.1.1.2. Subtask

OS/3 has another level of multitasking which may occur within a job step. The primary task is capable of initiating other tasks, called subtasks, within the job step. Primary tasks and subtasks are simply two categories of tasks; each is processed in the same manner. However, the primary task is automatically initiated into the multitasking environment by OS/3 at job step initiation, while subtasks must be created by the program in the job step. Subtasks can be given the same priority as the primary task or they can have a lower priority. Thus, a job step may consist of a primary task and several subtasks, all of which compete independently for processor time.

7.2. TASK MANAGEMENT

7.2.1. General

The supervisor is designed with multitasking capability which is utilized by the supervisor and extended to the user through macros. In a multitasking environment, several tasks may compete for control of the processor on a priority basis.

A task is defined as a point of control within an environment which is capable of utilizing the processor asynchronously with other tasks. It refers to a level of control only and not the physical code itself.

Every task, regardless of the code the task executes, will be identified to the supervisor by a task control block (TCB). The TCB contains or points to all control information associated with a task. This includes register/program status word save areas and other task-oriented information.

Each job step has a task (and thus a TCB) inherited at job step initialization from job control which is referred to as the primary task. Additional tasks may be attached as subtasks and cause additional TCBs to be created to identify the new tasks to the supervisor. The primary task is considered to represent the job step. As such, any termination, normal or abnormal, of this task will cause the job step to terminate.

Additional tasks (subtask, other than primary) are created by the ATTACH macro instruction which causes task management to create a TCB and initialize it with the attaching task's environment. Once a subtask has been created it is entered on the switch list to compete for the processor control on a priority basis. Each task competes independently for the processor. When the switcher gains control, it selects the highest priority nonwaited task and dispatches control. A task is nonwaited (active) if it can use the processor and waited (not active) if some event must take place before the task can use the processor.

A subtask terminates when a DETACH macro instruction is executed for that task or an error occurs which prevents the task from successfully completing its work. A DETACH in behalf of the primary task is interpreted as an end-of-job step (8.3.4). When a subtask terminates normally, a completion code is posted within the task's event control block (ECB) which can be examined by the attaching task to determine the result of the task processing. If a subtask terminates abnormally, an error code is posted for examination by the attaching task, and control is passed to the abnormal termination island code.

All tasks of a job have all the capabilities of the primary task; that is, a task can create additional tasks of its own and perform all communication functions with these tasks. The exception is that unsolicited operator messages can only be accepted at the job step level.

7.2.2. Task Creation

Task creation is performed by the ATTACH macro instruction, which causes entry into the attach function to create a subtask. The code to be executed by the task specified on the ATTACH macro instruction call must be in main storage, within the user region, when the ATTACH macro instruction is issued.

The number of tasks which may be created by a user is limited to the number designated to job control with a maximum of 255 subtasks. The space for creation of task control blocks is reserved by job control when the job region is established. The number of possible simultaneous tasks must be specified as a parameter on the JOB statement in the job control stream. The maximum number of ATTACH/DETACH sequences permissible per job step is also limited to 255.

Tasks may create other subtasks in a pyramidal fashion with a limit of four total or three subtask levels. This hierarchical structure is not intended to provide a means of task synchronization. This structure is composed of subtask families so that when a subtask terminates, the family it has created is also terminated.

When a task is created, the originating task must pass the address of an area in the user storage to be used as an event control block (ECB) for the newly created task. This ECB address is placed in the newly created task's TCB and may be considered as an extension to the TCB for the purpose of task synchronization by user. The separation of the ECB and TCB is a system requirement since a TCB cannot be addressed by the user programs.

7.2.3. Task Priority

When a primary task is created, job control assigns it a dispatching (switch list) priority as requested on the job control EXEC statement. Any subtask created by the primary task or other subtask can have a priority based on the primary task priority as specified on the ATTACH call. The attaching task may request the same or a lower priority for the new subtask.

7.2.4. Task Termination

A task executes a DETACH macro instruction to cause entry into the task termination function for processing. The DETACH function determines whether the DETACH was executed from abnormal termination island code to determine if termination was normal or abnormal. For normal termination, the ECB for that task is posted by the termination routines and all tasks in the subtask family of this task are terminated.

Task control posts completion codes for the terminating task, notifies any other task awaiting the completion of the terminating task, and unlinks the TCB from the system. The task termination routines recognize the TCB for a primary task and treat that as a job step termination (EOJ).

An abnormally terminating task is one that executed a CANCEL either intentionally or imposed by the system. Task control when processing an abnormally terminating task, posts the task's ECB, and activates abnormal termination island code under the primary task but in behalf of this task.

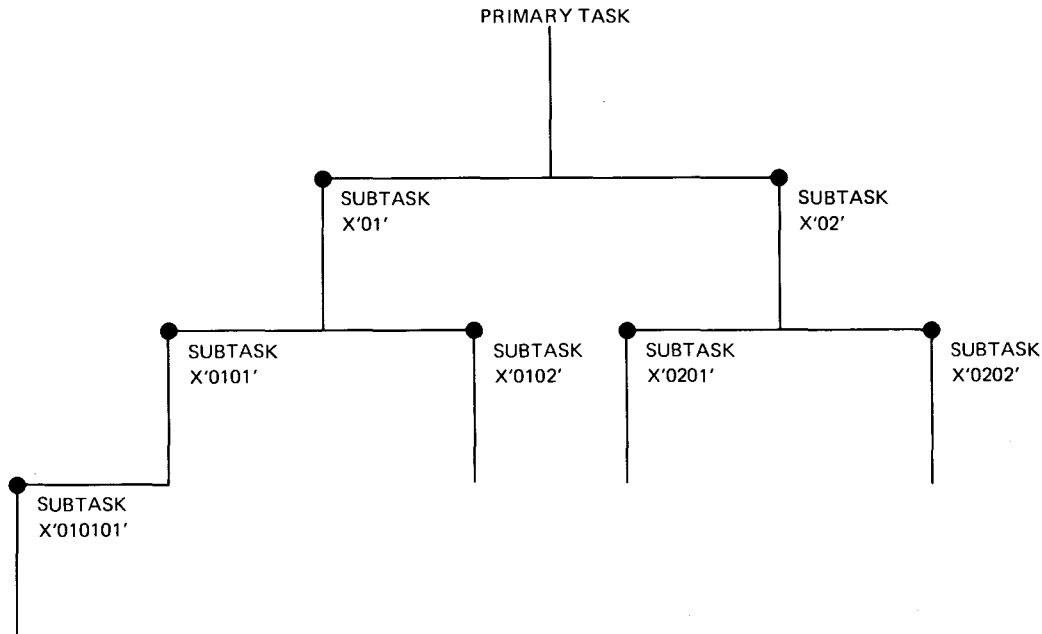
7.2.5. Queue Driven Task

The AWAKE function is provided for queue driven tasks to allow for better synchronization and less overhead. AWAKE can only be issued to a task which has been previously created by ATTACH. If the AWAKE function is addressed to a nonexistent task (no ATTACH), an abnormal termination is initiated. The AWAKE is utilized to activate an existing but idle task.

The queue driven task continues to process until it has exhausted all queue entries and then can execute the TYIELD macro instruction to mark itself nondispatchable until further queue entries have been made. Each time an AWAKE macro instruction is executed, the addressed task will be removed from the idle state. This is accomplished whether the task is idle or active and will permit a task to be dispatched.

7.2.6. Hierarchical Structure

Subtasks are attached as members of task families providing a hierarchical structure similar to a pyramid. This structure provides the family naming conventions which allow a task to terminate and have all its subtasks also terminated. The hierarchy is not imposed as a restriction to task synchronization or control. Therefore, tasking functions may reference across family lines. Additionally, this structure has no relationship to the dispatching priority. The hierarchical structure is illustrated in the following diagram:



Subtasks are named by concatenating the count of this task as a subtask of the attaching task to the attaching task's name. For example, if a task named X'0102' attaches its third subtask, the subtask will be named X'010203'. The primary task does not have a name since it is not a member of a subtask family. The family names can extend to three bytes which imposes the limit of three subtask levels below the primary task.

7.3. TASK MANAGEMENT MACRO INSTRUCTIONS

Task management macros provide the interface by which jobs can create and control a multitasking environment. Each job step by definition has at least one task, which is referred to as the primary task. The following macros allow for the creation, activation, deactivation and deletion of additional tasks within a job step.

The user must inform job control via job control statements of the maximum number of tasks that can be created for a job step. This allows job control to reserve the main storage required for TCB within the job's prologue. Likewise, you must provide storage for and control of the ECBs. These ECBs are 2-word (8 bytes) fields which task management utilizes to communicate with the user to allow for task synchronization and to identify the task. You can look at the information but should not write into these words which are unique to a given task. The primary task doesn't have an ECB and is identified by an ECB address of zero.

The following macro instructions are available for multitasking:

- ECB
Generates an event control block for task identification and status.
- ATTACH
Creates and activates an additional task.

- DETACH
Terminates a task normally.
- TYIELD
Deactivates a task.
- AWAKE
Reactivates an existing nonactive task.
- CHAP
Changes (reduces) the relative priority of a task.

7.3.1. Generate an Event Control Block (ECB)

Function:

The ECB macro instruction generates and initializes an event control block. The event control block is used by task management to identify a task and to indicate status to the other tasks within a job step. The current status of the associated task is reflected by bits within the ECB (Figure 7—1).

This is a declarative macro instruction and must not appear in a sequence of executable code.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|---------|
| [symbol] | ECB | |

There are no parameters for the ECB macro instruction.

The ECB is utilized to communicate between task management and the job step. The following programming considerations and conditions are set into the ECB.

1. The ATTACH macro specifies an ECB when the task is created. The specified ECB is linked to the TCB and is reserved for this task until this task is detached.
2. As with I/O, only one task can wait for a given command control block (CCB) or ECB. However, unlike I/O, which allows only the task that submitted the CCB to wait for it, task management allows only one of the other tasks to wait for the task which is identified by the ECB.

- A primary task does not have an ECB associated with it, therefore, the primary task cannot be awaited. This task can synchronize with I/O by utilizing the WAIT and WAITM macros and can synchronize with other tasks by the AWAKE and TYIELD macros.

Example:

| 1 | LABEL | Δ OPERATION Δ | OPERAND | Δ | COMMENTS |
|----|----------|---------------|----------------|---|-------------------------------|
| | PRIMTASK | START | | | |
| | |) | | | |
| 1. | | ATTACH | ECB1, START, 2 | | |
| | |) | | | |
| 2. | | WAIT | ECB1 | | |
| | | EQT | | | |
| 3. | ECB1 | ECB | | | |
| | | EQU | * | | SUBTASK EXECUTION STARTS HERE |
| | |) | | | |
| 5. | | DETACH | ECB1 | | SUBTASK EXECUTION ENDS HERE |

Explanation:

- Line 1 attaches a subtask whose ECB name is ECB1. The subtask will begin execution at the address of START. If the priority of PRIMTASK is two, the priority of the subtask being attached is four.
- At line 2, the primary task gives up control until the subtask is completed.
- Line 3 generates the ECB called ECB1 associated with the subtask. Note that this macro does not appear in a sequence of executable code.
- 4.&5.
Lines 4 and 5 represent the beginning and ending of the subtask execution.

| Byte | 0 | 1 | 2 | 3 |
|------|-------------------------------------|---------------------|---------------|--------|
| 0 | control byte | attaching task's ID | activity byte | unused |
| 4 | address of TCB waiting for this ECB | | | |

Byte

0 Control Byte

- Bit 0 1 = This is an ECB.
- 1—4 Not used
- 5 1 = This task completion is being awaited.
- 6—7 Not used

1 Attaching Task's ID

Task identification number of task with which this ECB is associated. This ID number is not related to subtask name. It is the number of the TCB counting from the job step TCB which is number 0.

2 Activity Byte

- Bit 0 0 = Task is active in that it has not executed either a TYIELD or DETACH macro.
- 1 = Task is idle in that it has executed a TYIELD or DETACH macro.
- 1—6 Not used
- 7 1 = Task has abnormally terminated and should be detached.

3 Unused

4—7 Address of TCB which is awaiting completion of the task with which this ECB is associated.

Figure 7—1. Event Control Block (ECB) Format

7.3.2. Create an Additional Task (ATTACH)

Function:

The ATTACH macro instruction creates and activates a task desiring control of the processor. It generates an additional task control block and enters the task onto the switch list.

Format:

| LABEL | Δ OPERATION Δ | OPERAND | |
|----------|---------------|---------------------|--|
| [symbol] | ATTACH | { ECB-name } (1) | , { entry-point-name } (0) [, { error-addr }] [, n] (r) |

Positional Parameter 1:

ECB-name

Specifies the symbolic address of the ECB used to identify and control this task.

(1)

Indicates that register 1 has been preloaded with the address of the event control block.

Positional Parameter 2:

entry-point-name

Specifies the symbolic address of the point in the program at which this task will receive control. The coding to be executed for the task must be in main storage when the ATTACH macro instruction is issued.

(0)

Indicates that register 0 has been preloaded with the address of the entry point.

Positional Parameter 3:

error-addr

Specifies the symbolic address of an error routine to receive control if an error occurs.

(r)

Indicates that the register designated (other than 0 or 1) has been preloaded with the address of the error routine.

If omitted, the task will be abnormally terminated if an error occurs.

Positional Parameter 4:

n

Specifies a 1-byte value to be added to the switch list priority of the originating task. This raises the dispatching priority value resulting in a lesser priority for the task. (The higher the priority number, the lower the priority.) The result is assigned as the switch list priority of the new task unless it exceeds the limit of this system, in which case the highest number (lowest priority) for this system is used.

The use of this parameter always results in a lesser priority. There is no way to attach a task with a priority higher than that of the primary task.

If omitted, the new task will be created at the same priority as the originating task.

Example:

| 1 | LABEL | Δ OPERATION Δ | OPERAND | Δ | COMMENTS |
|---|---------|---------------|-------------------------|---|---------------------------------------|
| | | 10 | 16 | | |
| | | ATTACH | ECB1, ENTRYPT, ERROR, 2 | | |
| | ECB1 | ECB | | | |
| | ENTRYPT | EQU | * | | SUBTASK EXECUTION BEGINS HERE |
| | ERROR | EQU | * | | CONTROL RETURNS HERE IN CASE OF ERROR |

Attach a task identified by the event control block named ECB1. The subtask will receive control at the instruction whose address is labeled ENTRYPT. If an error is encountered during the execution of the ATTACH macro instruction, control will be transferred to the error processing routine labeled ERROR. The dispatching priority of the newly created task will be two greater than that of the originating task.

7.3.3. Terminate a Task (DETACH)

Function:

The DETACH macro instruction terminates a task by delinking the TCB from the switch list and returning the TCB to the job's free TCB queue. If this macro instruction is executed by the primary task, it will be interpreted as an end of job step. All subtasks of the task being detached will also be detached.

This macro instruction also clears all I/O locks for the task.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|---|
| [symbol] | DETACH | {(ECB-name)} [(1)] [, {error-addr} (r)] |

Positional Parameter 1:

ECB-name

Specifies the symbolic address of the event control block of the task to be detached.

(1)

Indicates that register 1 has been preloaded with the address of the ECB.

If omitted, indicates that the task issuing the DETACH instruction is terminating. No task other than the primary can terminate the primary task.

Positional Parameter 2:

error-addr

Specifies the symbolic address of an error routine to be executed if an error occurs.

(r)

Indicates that the register designated (other than 0 or 1) has been preloaded with the address of the error routine.

If omitted, the executing task will be abnormally terminated if an error occurs.

Example:

| 1 | LABEL | ΔOPERATIONΔ | OPERAND | Δ |
|---|-------|-------------|-------------|---|
| | | 10 16 | | |
| | | DETACH | ECB1, ERROR | |
| | | | | |
| | | | | |
| | | | | |
| | ECB1 | ECB | | |
| | ERROR | EQU | * | |

Detach the task identified by the event control block labeled ECB1. If an error occurs during the DETACH macro instruction, transfer control to error processing routine labeled ERROR.

7.3.4. Yield Until Task Completion (TYIELD)

Function:

The TYIELD macro instruction relinquishes control of the processor and sets the TCB in a waiting state. The ECB is tested and if there is a task awaiting the yielding task, the waiting task is activated.

The TYIELD macro instruction is used in combination with the AWAKE macro instruction which reactivates a task made dormant by the TYIELD macro instruction.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|---------|
| [symbol] | TYIELD | |

There are no parameters for the TYIELD macro instruction.

7.3.5. Reactivate a Task (AWAKE)

Function:

The AWAKE macro instruction reactivates a task made dormant by a TYIELD macro instruction. It clears the TYIELD bit within the wait bytes of the TCB regardless of whether or not the task is idle, thereby activating the task to receive control of the processor from the switcher.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|-------------------------|
| [symbol] | AWAKE | [{ ECB-name } (1)] |

Positional Parameter 1:

ECB-name

Specifies the symbolic address of the event control block of the task to be reactivated.

(1)

Indicates that register 1 has been preloaded with the address of the ECB.

If omitted, or if this macro instruction is executed with a zero address in register 1, the primary task will be taken out of a TYIELD condition.

Examples:

| 1 | LABEL | △OPERATION△ 10 16 | OPERAND | △ |
|----|-------|---------------------------|-----------------|--------------------|
| 1. | | AWAKE | | AWAKE PRIMARY TASK |
| 2. | | SR AWAKE | RI, RI (1) | AWAKE PRIMARY TASK |
| 3. | | AWAKE | ECB1 | AWAKE SUBTASK |
| | | . | | |
| | | . | | |
| | | . | | |
| | ECB1 | ECB | | |
| 4. | | LA AWAKE | RI, ECB1 (1) | AWAKE SUBTASK |

Explanations:

Examples 1 and 2 will take the primary task out of a TYIELD condition.

Examples 3 and 4 indicate that the task identified by the ECB named ECB1 will be taken out of a TYIELD condition.

7.3.6. Change a Priority (CHAP)

Function:

The CHAP macro instruction reduces the dispatching priority of the task issuing the instruction. The increment entered as the operand is added to the current dispatching priority of the task (specified by the switch-priority parameter in the EXEC job control statement). This raises the dispatching priority value resulting in a lesser priority for the task. This macro instruction does not change the priority to a specific level; instead, it adjusts the priority relative to the level under which it is executed.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|--------------|
| [symbol] | CHAP | { n } (1) |

Positional Parameter 1:

- n
Specifies a 1-byte value to be added to the dispatching priority for the task.
- (1)
Indicates register 1 has been preloaded with the increment.

Example:

| 1 | LABEL | Δ OPERATION Δ | OPERAND | Δ |
|----|-------|---------------|---------|---|
| | | 10 | 16 | |
| 1. | | CHAP | 2 | |
| 2. | | LA | R1, 2 | |
| | | CHAP | (1) | |

Change the dispatching priority of the task by two. This will raise the dispatching priority value by two, which will result in a priority two less than the current priority. Both examples perform the same function.

7.4. TASK SYNCHRONIZATION

7.4.1. General

Task synchronization provides a task with a means of waiting for one or more other tasks. The waiting task is awaiting the completion of the specified task or tasks which is signaled by the deactivation of an awaited task or by the execution of the POST macro instruction.

Tasks are waited by setting a unique wait bit within that TCB. These wait bits signal the switcher that this task is nondispatchable and indicate the reason for the wait. Upon clearing the wait bits, the task becomes dispatchable and can be activated.

The ECB address, which is specified as a parameter to task management macros, points to an event control block which allows for task to task synchronization. The ECB format is compatible with the first two words of I/O CCBs as far as the WAIT and WAITM macro instructions are concerned. These macros are utilized to synchronize tasks in a manner similar to I/O synchronization.

When the performance of a task is dependent on any other task or tasks, the tasks involved may synchronize themselves via the ECB associated with a task from the ATTACH macro instruction. The ECB is posted with a completion code when a task terminates or executes the TYIELD macro instruction. The ECB may be specified on a WAIT and WAITM instruction in order to hold processing of a task until the awaited task terminates or by the POST macro instruction when some event is completed.

Three macro instructions are available for task synchronization:

- WAIT

Wait for a task request to complete.

- WAITM

Wait for one of several task requests to complete.

- POST

Activate a waiting task.

These macro instructions can also be used (with different parameters) to synchronize a task with its I/O. For task synchronization, the macro instruction references an event control block; and for I/O synchronization, the macro instruction references a command control block. I/O synchronization is described in 4.3.

7.4.2. Wait for Task Completion (WAIT)

Function:

The WAIT macro instruction temporarily suspends program execution until the specified task is completed or executes a POST macro instruction in behalf of the waiting task. If the related task is completed, control is returned to the point immediately following the WAIT macro instruction. If the awaited task is not complete, the issuing task is placed in a wait state and control is passed to another task.

The ECB indicates the status of the task. When a WAIT macro instruction is issued, the issuing task relinquishes control until the ECB is marked complete or until a POST macro instruction is executed by the awaited task in behalf of the waiting task.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|-------------------|
| [symbol] | WAIT | {ECB-name} (1) |

Positional Parameter 1:

ECB-name

Specifies the symbolic address of the event control block to be tested for completion.

(1)

Indicates that register 1 has been preloaded with the address of the event control block.

Example:

| 1 | LABEL | Δ OPERATION Δ | OPERAND | Δ |
|---|-------|---------------|---------|---|
| | | 10 | 16 | |
| | | WAIT | ECB1 | |
| | | WAIT | (1) | |

7.4.3. Multiple Task Wait (WAITM)

Function:

The WAITM macro instruction temporarily suspends program execution until any one of several tasks specified by the instruction is completed or executes a POST macro instruction in behalf of the waiting task. Upon completion of one of the tasks, control is returned to the program at the point immediately following the WAITM macro instruction, with register 1 containing the address of the event control block associated with the completed task.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|---|
| [symbol] | WAITM | {ECB-name-1, ECB-name-2[, ..., ECB-name-n]} list-name (1) |

Positional Parameter 1:

ECB-name-1, ECB-name-2, ..., ECB-name-n

Specifies the symbolic addresses of the event control blocks to be tested that are associated with the tasks to be awaited. At least two ECBs must be specified.

list-name

This is a single entry which specifies the symbolic address of a list containing full-word addresses of ECBs associated with the tasks to be awaited. The byte following the last full word must be nonzero to indicate end of list.

(1)

Indicates that register 1 has been preloaded with the address of the list of ECB addresses.

NOTE:

The WAITM macro instruction may also specify a combination of ECB and CCB addresses as parameters. See also the multiple I/O wait macro instruction described in 4.3.2.

When this macro instruction is executed, each referenced ECB is marked as being awaited. Upon completion of a marked ECB, the waiting task is activated and the remaining ECBs that are marked as being awaited are cleared.

The WAITM macro instruction always requires more than one event to be tested. If only one event is to be tested, use the WAIT macro instruction.

7.4.4. Activate the Waiting Task (POST)

Function:

The POST macro instruction activates the waiting task without requiring the awaited task to terminate. When the POST macro instruction is issued by a task, the task waiting on the event completion which was posted will be reactivated at the point immediately following the WAIT or WAITM macro instruction.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|---------|
| [symbol] | POST | |

The task being activated by the POST macro instruction is the one waiting for the task executing the POST macro instruction, therefore, there are no parameters for this macro instruction.

Examples:

| 1 | LABEL | ΔOPERATIONΔ | OPERAND | Δ |
|---|----------|-------------|----------|---|
| | | 10 | 16 | |
| | TASK1 | EQU | * | |
| | | AWAKE | TASK2ECB | |
| | | WAIT | TASK2ECB | |
| | | . | | |
| | | . | | |
| | | . | | |
| | TASK1ECB | ECB | | |
| | TASK2ECB | ECB | | |
| | TASK2 | EQU | * | |
| | | EXCP | INPUTCCB | |
| | | WAIT | INPUTCCB | |
| | | POST | | |
| | | . | | |
| | | . | | |
| | | . | | |
| | | TYIELD | | |

PART 4. SUPERVISOR SERVICES



8. Program Management

8.1. GENERAL

Although the supervisor of the OS/3 manages the system resources for efficient overall operation, management of problem program resources is performed by the individual programs. The supervisor assists the program in using its allocated main storage space and allotted processor time by providing services that, when invoked by certain macro instructions, support flexibility in program design and economy of program execution.

The services are:

- Program initiation and loading

- Dynamic allocation of main storage at job initiation.

- Dynamic loading of program modules.

- Program termination

- Orderly release of system resources assigned to a problem program.

8.1.1. Program Initiation and Loading

Before a program is actually brought into main storage for execution, job control has already built the prologue for the job. The supervisor is responsible for locating and loading the proper program or segment. The supervisor uses the job prologue area for communication between job control and itself. The supervisor accesses information in the prologue and also enters information in the prologue area.

The supervisor must then:

- Assign relocation register number and value
- Set proper storage protection keys in key storage
- Load program into main storage for execution

- Link user job step at proper execution priority
- Pass control to task switcher

8.2. PROGRAM LOADER

The program loader is responsible for locating and loading program modules or overlays output by the linkage editor in the form of phases. A load module phase may be thought of as a program segment that can perform one or more specific processing operations. The following macro instructions are available:

- **LOAD**
Load a phase and return control.
- **LOADR**
Load a phase, relocate address constants, and return control.
- **LOADI**
Locate a phase and return its phase header in a work area.
- **FETCH**
Load a phase and give it control.

The use of these macro instructions is described on the following pages.

In addition, the loader is capable of modifying data in any phase of a problem program whenever that phase is loaded. The job control ALTER statement is used to specify such changes to the loader.

8.2.1. Block Loader

The LOAD, LOADR, LOADI, and FETCH macro instructions handle both standard load modules, which are loaded by the regular program loader, and block modules, which are loaded by the block loader, an extension of the program loader. The program loader reads one sector at a time from disk, and then moves this data one record at a time to the user job region in main storage. The block loader reads an entire track of data at a time directly into the user job region in main storage. You can take advantage of the faster block loader by using the BLK control statement in the system librarian to convert a load module phase from the standard load module format to block format (described in the system service programs user guide, UP-8062 (current version)). This may be done at any time before the job is executed and there is no need to specify in the macro instructions loading the phase whether the load module phase is in the standard format or in block format.

8.2.2. Relocation

The loader can perform positive or negative relocation on 8-bit, 16-bit, 24-bit, or 32-bit fields as specified by the relocation list dictionary (RLD) information in the text/RLD records of the load module.

Because of the relocation register, user programs do not require location of address constants (A-cons) when the phase is loaded at the address at which it was linked. If an alternate load address is specified on LOAD or LOADR, however, the loader handles it as follows:

- **LOAD**

No relocation is performed. You must ensure that the phase being loaded is self relocating.

- **LOADR**

Relocation is performed on all A-cons specified by the linker which refer to addresses in that same phase. A-cons which point inside another phase are not relocated since the loader has no way of knowing where that phase was loaded.

The following examples illustrate when the loader performs relocation:

| <u>Macro</u> | <u>Call</u> | <u>User Program Relocation</u> |
|--------------|-------------|--------------------------------|
| LOAD | NAME | No |
| LOADR | NAME | No |
| FETCH | NAME | No |
| LOAD | NAME,ALTAD | No* |
| LOADR | NAME,ALTAD | Yes |
| FETCH | NAME,ENTPT | No |

*Phase being loaded should be self-relocating.

8.2.3. Library Search Order

The default order of search employed by the loader is:

1. Load library file (\$Y\$LOD)
2. Temporary job run library file (\$Y\$RUN)

If the temporary job run library file (\$Y\$RUN) is specified on the EXEC job control card, the order of search is:

1. Temporary job run library file (\$Y\$RUN)
2. Load library file (\$Y\$LOD)

If an alternate library is specified on the EXEC job control card, the order of search is:

1. Alternate load library
2. Load library file (\$Y\$LOD)
3. Temporary job run library file (\$Y\$RUN)

To minimize search time, the loader always begins searching a library at the last root phase loaded from that library for that job. This means that it is generally more efficient to link modules together than to create a series of smaller, separately linked load modules.

8.2.4. Read Pointer for Repetitive Loads

Another way to minimize search time is to reduce the need for a directory search. This can be done by using a read pointer for repetitive loads of a particular load module. When the disc address (DA) optional parameter is used with the LOAD, LOADR, or FETCH macro instruction, the 8-byte EBCDIC phase name in the user program (possibly within the macro-generated code) is overwritten with a read pointer during the first execution of the macro. This read pointer contains the relative disk address of the phase being loaded. The next execution of the same macro call uses this read pointer to find the phase, instead of performing a directory search.

With the DA option, only the first load of a module requires a directory search. All subsequent loads of the same module use the read pointer and do not have to repeat the directory search. In this case, the larger the directory, the more efficient the use of the read pointer.

When using the DA option, you must be certain that the module is not being updated by another job at the same time that it is being loaded by your job; otherwise, an error will result. Remember, the DA option may be used only with the LOAD, LOADR, and FETCH macro instructions, and should only be used for repetitive loads of the same module. It is not available for use with the LOADI macro instruction. If you do not wish to add the DA capability to an assembled program, there is no need to reassemble.

8.2.5. Loader Error Processing

When an error is detected by the loader, a binary error code is set up in register 0. If an error address was specified, the macro-generated code branches to that address. If no error address was specified (or if the call was a FETCH), the calling task is abnormally terminated.

The 4-byte error code set up in register 0 has the following format:

Byte 0 = EBCDIC A, R, or L specifying whether the error occurred while loading from alternate, run, or load library respectively.

Bytes 1, 2 = 0

Byte 3 = Binary error code. For descriptions, refer to the system messages programmer/operator reference, UP-8076 (current version).

8.2.6. Load a Program Phase (LOAD)

Function:

The LOAD macro instruction locates a program phase in a load library on disk, loads it into main storage, and transfers control to the calling program immediately following the LOAD macro instruction.

After execution of this macro instruction, register 0 contains the job-relative address at which the phase was loaded, and register 1 contains the entry-point address. This entry point address is determined at linkage edit time. If an alternate load address is provided (positional parameter 2), the load point address specified to the linkage editor is overridden and the phase is loaded at the specified address. This new override address is returned in register 0.

This macro instruction does not relocate address constants regardless of whether an alternate load address is specified (positional parameter 2).

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|---|
| [symbol] | LOAD | $\left\{ \begin{array}{c} \text{phase-name} \\ (1) \end{array} \right\} \left[, \left\{ \begin{array}{c} \text{load-addr} \\ (0) \end{array} \right\} \right] \left[, \left\{ \begin{array}{c} \text{error-addr} \\ (r) \end{array} \right\} \right] [,R][,DA]$ |

Positional Parameter 1:

phase-name

Specifies the name of the program phase to be loaded. This may be either the 1- to 6-character user-assigned alias phase name or the 8-character linker-assigned phase name in the format nnnnnpp where nnnnnn is the program name and pp is the phase number.

(1)

Indicates that register 1 has been preloaded with the address of the 8-character phase name.

Positional Parameter 2:

load-addr

Specifies the symbolic address at which the phase is to be loaded.

(0)

Specifies that register 0 has been preloaded with the load address.

If omitted, the program phase will be loaded at the address specified by the linkage editor.

Positional Parameter 3:

error-addr

Specifies the symbolic address of an error routine that is to be executed if a load error occurs.

(r)

Specifies that the designated register (other than 0 or 1) contains the address of the error routine.

If omitted, the calling task will be abnormally terminated if a load error occurs.

Positional Parameter 4:

R

Specifies that only the system load library is to be searched for the phase.

If omitted, a full search is to be performed (8.2.3.).

Positional Parameter 5:

DA

Specifies that the 8-byte phase name specified in positional parameter 1 will be overwritten with a read pointer during the first execution of this macro instruction. This read pointer is used to find the phase on the second and all subsequent executions of this macro instruction.

If omitted, a search is performed on the phase name specified in positional parameter 1 each time this macro instruction is executed, and the 8-byte phase name is not overwritten.

8.2.7. Load a Program Phase and Relocate (LOADR)

Function:

The LOADR macro instruction locates a program phase in a load library on disc, loads it into main storage, and transfers control to the calling program immediately following the LOADR macro instruction.

After execution of this macro instruction, register 0 contains the job-relative address at which the phase was loaded, and register 1 contains the job-relative entry-point address. This entry point address is determined at linkage edit time. If an alternate load address is provided (positional parameter 2), the load point address specified to the linkage editor is overridden and the phase is loaded at the specified address. This new override address is returned in register 0.

The format and operation of the macro instruction is identical to the LOAD macro instruction except that all address constants in the phase are relocated if an alternate load address is specified (positional parameter 2).

This macro instruction is used to load a phase at an address other than that at which it was linked.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|---|
| [symbol] | LOADR | $\left\{ \begin{array}{c} \text{phase-name} \\ (1) \end{array} \right\} \left[\left[\left\{ \begin{array}{c} \text{load-addr} \\ (0) \end{array} \right\} \right] \left[\left\{ \begin{array}{c} \text{error-addr} \\ (r) \end{array} \right\} \right] \right] [, R] [, DA]$ |

Positional Parameter 1:

phase-name

Specifies the name of the program phase to be loaded. This may be either the 1- to 6-character user-assigned alias phase name or the 8-character linker-assigned phase name in the format nnnnnpp where nnnnnn is the program name and pp is the phase number.

(1)

Indicates that register 1 has been preloaded with the address of the 8-character phase name.

Positional Parameter 2:

load-addr

Specifies the symbolic address at which the phase is to be loaded.

(0)

Specifies that register 0 has been preloaded with the load address.

If omitted, the program phase will be loaded at the address specified by the linkage editor.

Positional Parameter 3:

error-addr

Specifies the symbolic address of an error routine that is to be executed if a load error occurs.

(r)

Specifies that the designated register (other than 0 or 1) contains the address of the error routine.

If omitted, the calling task will be abnormally terminated if a load error occurs.

Positional Parameter 4:

R

Specifies that only the system load library is to be searched for the phase.

If omitted, a full search is to be performed (8.2.3).

Positional Parameter 5:

DA

Specifies that the 8-byte phase name specified in positional parameter 1 will be overwritten with a read pointer during the first execution of this macro instruction. This read pointer is used to find the phase on the second and all subsequent executions of this macro instruction.

This option is designed to reduce the search time for separately linked load modules which are loaded repeatedly. When using this option, you must ensure that there is no possibility of another job deleting or moving the load module you are trying to load. For example, if another job uses the librarian to pack the library, this may cause a load error in your job. If you can be sure this doesn't happen, you may be able to reduce considerably the load time for some modules, particularly in large libraries.

If omitted, a search is performed on the phase name specified in positional parameter 1 each time this macro instruction is executed, and the 8-byte phase name is not overwritten.

8.2.8. Locate a Program Phase Header (LOADI)

Function:

The LOADI macro instruction locates the header record of a program phase and stores it in a work area.

You may then examine the information contained in the program phase header to determine if it is desirable to load the program phase. If the phase is to be loaded, you must use one of the other load instructions to load the program phase.

The format of the phase header record is shown in 8.2.8.1.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|--|
| [symbol] | LOADI | $\left\{ \begin{array}{c} \text{phase-name} \\ (1) \end{array} \right\}, \left\{ \begin{array}{c} \text{work-area-addr} \\ (0) \end{array} \right\} \left[\left\{ \begin{array}{c} \text{work-area-length} \\ 13 \end{array} \right\} \right]$ $\left[\left\{ \begin{array}{c} \text{error-addr} \\ (r) \end{array} \right\} \right] [,R]$ |

Positional Parameter 1:

phase-name

Specifies the name of the program phase to be loaded. This may be either the 1- to 6-character user-assigned alias phase name or the 8-character linker-assigned phase name in the format nnnnnpp where nnnnnn is the program name and pp is the phase number.

(1)

Indicates that register 1 has been preloaded with the address of the 8-character phase name.

Positional Parameter 2:

work-area-addr

Specifies the symbolic address of the area in main storage where the phase header is to be placed.

(0)

Specifies that register 0 has been preloaded with the work area address.

Positional Parameter 3:

work-area-length

Specifies the number of bytes of the phase header that are to be placed in the work area.

If omitted, the value 13_{10} is assumed. This specifies that the portion of the phase header up to and including the phase load address and the phase length is to be placed in the work area.

Positional Parameter 4:

error-addr

Specifies the symbolic address of an error routine that is to be executed if a load error occurs.

(r)

Specifies that the designated register (other than 0 or 1) has been preloaded with the address of the error routine.

If omitted, the calling task will be abnormally terminated if a load error occurs.

Positional Parameter 5:

R

Specifies that only the system load library is to be searched for the phase.

If omitted, a full search is to be performed (8.2.3).

8.2.8.1. Program Phase Header

The format of the phase header is as follows:

| <u>Bytes</u> | <u>Contents</u> |
|--------------|--------------------------------------|
| 0,1 | Systems use |
| 2 | Phase number |
| 3,4 | System flags |
| 5-8 | Phase load address (linker assigned) |
| 9-12 | Phase length |
| 13-20 | Phase name (linker assigned) |
| 21-23 | Date (packed decimal — yymmdd) |
| 24-26 | Time (packed decimal — hhmmss) |
| 27-30 | Module length |
| 31-38 | Alias phase name |
| 39-68 | Comments |

8.2.9. Load a Program Phase and Branch (FETCH)

Function:

The FETCH macro instruction locates a program phase in a load library on disk, loads it into main storage, and transfers control to the address specified in the phase transfer record, unless an alternate address has been specified (in positional parameter 2).

After execution of this macro instruction, register 0 contains the job-relative address at which the phase was loaded, and register 1 contains the job-relative entry point address. This entry point address is determined at linkage edit time. If an alternate entry point address is provided (positional parameter 2), the entry point address specified to the linkage editor is overridden and the phase is given control at the specified address. This new entry point address is returned in register 1.

Format:

| LABEL | △OPERATION△ | OPERAND |
|----------|-------------|--|
| [symbol] | FETCH | $\left\{ \begin{array}{c} \text{phase-name} \\ (1) \end{array} \right\} \left[\cdot \left\{ \begin{array}{c} \text{entry-point-name} \\ (0) \end{array} \right\} \right] [,R][,DA]$ |

Positional Parameter 1:

phase-name

Specifies the name of the program phase to be loaded. This may be either the 1- to 6-character user-assigned alias phase name or the 8-character linker-assigned phase name in the format nnnnnpp where nnnnnn is the program name and pp is the phase number.

(1)

Indicates that register 1 has been preloaded with the address of the 8-character phase name.

Positional Parameter 2:

entry-point-name

Specifies the symbolic address of the point in the program at which control is to be passed after a successful load.

(0)

Indicates that register 0 has been preloaded with the entry point address.

If omitted, control will be passed to the address specified in the phase transfer record.

Positional Parameter 3:

R

Specifies that only the system load library is to be searched for the phase.

If omitted, a full search is to be performed (8.2.3).

Positional Parameter 4:

DA

Specifies that the 8-byte phase name specified in positional parameter 1 will be overwritten with a read pointer during the first execution of this macro instruction. This read pointer is used to find the phase on the second and all subsequent executions of this macro instruction.

If omitted, a search is performed on the phase name specified in positional parameter 1 each time this macro instruction is executed, and the 8-byte phase name is not overwritten.

8.3. PROGRAM TERMINATION

The program termination macro instructions cause the system facilities assigned to a job or to a task to be relinquished for assignment to other jobs or to other tasks. When terminating a task, the EOJ, CANCEL, and DETACH macro instructions will also clear all I/O locks for the task.

The following macro instructions are available:

- **EOJ**

Causes normal job step termination.

- **CANCEL**

Causes abnormal job termination and prints out the job main storage.

There are two other macro instructions used for job and task termination:

- **DETACH**

Causes normal termination of a task (7.3.3).

- **DUMP**

Causes normal job step termination in addition to printing out the job main storage (9.1.2).

8.3.1. Normal Termination

Normal program termination is requested by means of the EOJ or DUMP macro instructions, the DUMP operator command, or the self detaching of a primary task. This implies normal completion of the job step and continuation of the job. These functions will allow all task and I/O to idle down prior to terminating the program and passing control to the next phase of job control.

The termination of a job step which has open data files will cause an immediate cancellation of the job.

The DUMP macro instruction provides a printout of the contents of the job region. Subsequent to printing the region the DUMP transient routine overlays itself with the end-of-job step transient routine which provides normal job step termination.

The DUMP console command sets the job step to execute its own DUMP macro instruction.

8.3.2. Abnormal Termination

Abnormal job termination can be requested by you through the CANCEL macro instruction, by the operator through the CANCEL command, or as a result of a system detected error. The latter case includes: systems function errors with no error address specified, program exception errors without program check island code, and unrecoverable hardware errors.

The cancel function detaches all subtasks, delinks all outstanding I/O and waits for all outstanding system functions to be completed. It provides a printout of the contents of the job region if the DUMP option was specified on the OPTION statement, and either there is a printer assigned to this job or there is a printer available.

8.3.3. Printout

Both the CANCEL and DUMP macro instructions provide for a printout of the contents of the job main storage which will occur if a printer was assigned to the job using the DVC and LFD job control statements, or is available for assignment, and the DUMP, JOBDUMP, or SYSDUMP parameter was specified in the OPTION job control statement. Otherwise, both macro instructions will execute normally; however, no printout will occur.

8.3.4. End-of-Job Step (EOJ)

Function:

The EOJ macro instruction causes normal job step termination. It terminates a primary task or a subtask. If an EOJ macro instruction is issued from a primary task with active subtasks, all subtasks are terminated. If an EOJ macro instruction is issued from a subtask, only the subtask and any subtasks it created are terminated.

This macro instruction also clears all I/O locks for the task.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|---------|
| [symbol] | EOJ | |

There are no parameters for the EOJ macro instruction.

The EOJ macro instruction is used to cause normal job step termination. It is usually invoked by the job step task after all attached subtasks have been detached and all data files have been closed. Job control is then loaded in the problem program area to prepare the next scheduled job step, or to terminate the job if it is the last job step of the job.

The EOJ macro instruction may be used to force subtask termination for the job step. If a subtask encounters a fatal (abnormal termination) error condition before the EOJ function receives control, the job may be cancelled (depending on the existence and function of an abnormal termination island code routine). An EOJ macro instruction executed by a subtask is treated as a request for the DETACH macro instruction function.

Error Conditions:

The job will be cancelled if errors which prevent normal termination are encountered by the EOJ routine. A hexadecimal error code is provided for display in the diagnostic storage dump produced by the CANCEL function. The error codes and their meaning are shown in the system messages programmer/operator reference, UP-8076 (current version).

8.3.5. Cancel a Job (CANCEL)

Function:

The CANCEL macro instruction causes abnormal termination of a job. It terminates the current job step, prevents execution of any remaining job steps for that job, detaches all subtasks, delinks all outstanding I/Os, and waits for all outstanding system functions to complete.

This macro instruction also displays an abnormal termination message on the operator console indicating which job is being terminated and the error code defining the error. In addition, this macro instruction provides a diagnostic storage dump of the job region similar to that produced by the DUMP macro instruction. (See 9.1.2 for details of the dump printout.)

This macro instruction also clears all I/O locks for the task.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|--------------------------------------|
| [symbol] | CANCEL | { (error-code) } { (0) } { 0 } |

Positional Parameter 1:

error-code

Specifies a 1- to 3-digit hexadecimal error code to be displayed on the system console and included in the diagnostic storage dump.

(0)

Indicates that register 0 has been preloaded with the error code.

If omitted, the error code is set to binary zero.

The CANCEL macro instruction is used to cause abnormal job termination when error conditions are encountered which prevent further processing. The abnormal job termination function may be requested by you through the CANCEL macro instruction, by the operator through the CANCEL command, or as a result of a system detected error.

If an error occurs during the execution of a macro instruction, control will be passed to the error routine if an error address was specified or, if none was specified, to the abnormal termination island code if it is present. The use of island code permits you to take additional action prior to terminating the task or job step which is in error.

Error Conditions:

A number of conditions may exist when the cancel routine is entered, however, the error code displayed in the diagnostic storage dump will always represent the original cause of entry to the abnormal termination function.

A printout is produced if:

- the DUMP, JOBDUMP, or SYSDUMP option was specified via job control; and
- a printer was assigned to the job or is available.

8.4. TIMER SERVICES

During execution of a job, you may want to record the date and time that an event occurred, for example, the date a credit was posted to an accounts receivable record, the date and time a message was received from a remote communications terminal, or the date and time a job step was completed. You can do this by using the GETIME macro instruction.

At times you may want to request an interrupt to your program after a specified interval. For example, you may wish to allow 30 seconds for a response from a terminal, and if no response is received within that time, branch to another subroutine or to another task. You can do this by using the SETIME macro instruction.

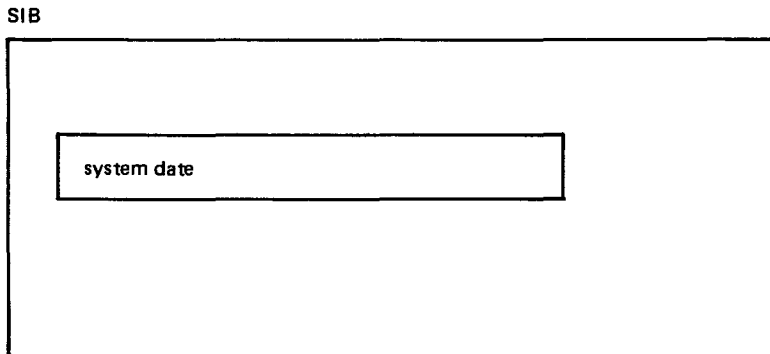
The date capability is always available. The GETIME macro instruction can be used to obtain the date even if the timer services module is not part of the resident supervisor.

Day clock and timer facilities are optional functions which must be requested during system generation. For full use of the GETIME macro instruction, and to use the SETIME macro instruction, the timer services module must be resident. For example, if the statement `TIMER=MAX` was specified at system generation, the timer services module is included in the generated operating system and stored in the resident supervisor portion of main storage. This allows you to use the GETIME macro instruction (for both date and time) and the SETIME macro instruction.

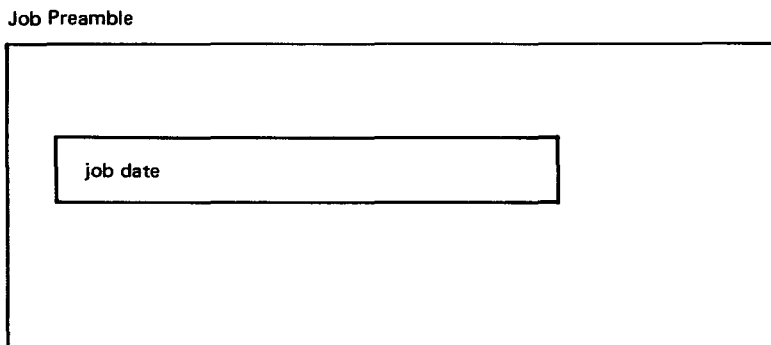
8.4.1. Date and Time Facilities

8.4.1.1. Current Date

The current date is placed in the systems information block (SIB) by the operator during initial program load. The date is automatically advanced each day at midnight unless the supervisor was configured at system generation time not to update. In that case, the operator must change the date through a console command. This date is referred to herein as the system date to distinguish it from the job date.



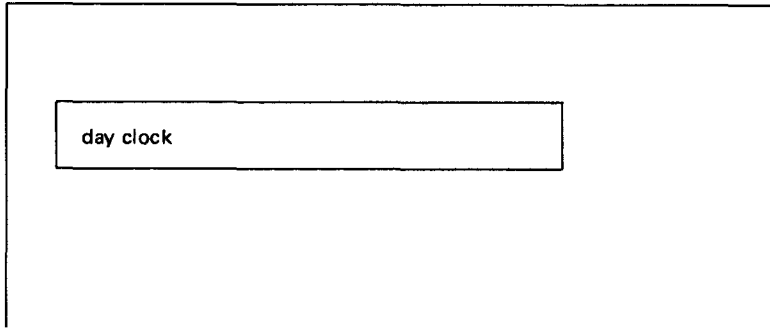
There is a date for each job, which is stored in the preamble for the job. This is the date you get when you use the GETIME macro instruction. Normally, the job date is the same as the system date. However, you can change it using the SET job control statement which changes the date for your own job and does not disturb the system date or the job dates for other jobs being processed. For example, if your application calls for statements to be produced on the fifteenth of each month but no processing was done that day because of a holiday or because of machine maintenance, you could change the job date in the preamble the next day from 16 to 15 so that the statements and other records produced will show the date the job was intended to be run.



8.4.1.2. Time of Day

If the timer services module is resident, the GETIME macro instruction gives you the time along with the date. The current time of day is maintained by a simulated day clock in the SIB. This day clock specifies the amount of time that elapsed since midnight. The clock can show a maximum of 99 hours and may be permitted to run past midnight if jobs were processing at that time. The time of day is automatically reset at midnight along with the date unless the supervisor was configured not to update. Otherwise, the operator must reset the clock each day. A common use of the clock is to record the time of day a job was run and to calculate the length of time required to run it. The job log you receive with your listing shows the start and stop times for your job steps. The run time could be used to charge an account number, or to invoice your department for the computer time required to run your job.

SIB



8.4.1.3. Get Current Date and Time (GETIME)

Function:

The GETIME macro instruction obtains the calendar date and the current time of day from the simulated day clock function of the supervisor. The date is returned in register 0, and the time is returned in register 1. If the timer services module is not part of the resident supervisor, the contents of register 1 are unpredictable.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|--------------------|
| [symbol] | GETIME | [{ M } { S }] |

Positional Parameter 1:

M

Specifies that the current time of day is to be expressed in milliseconds in binary representation.

S

Specifies that the current time of day is to be expressed in packed decimal format.

If omitted, the parameter S is assumed.

The current calendar date is returned in register 0 expressed in packed decimal in the form:

Oyymmdd+

where:

yy = year

mm = month

dd = day

The high-order half byte is always zero, and the low-order half byte is the sign, which is always positive.

The current time of day is returned in register 1. If you write this macro instruction with the S parameter or with no parameter, the time is expressed in packed decimal format in the form:

Ohhmss+

where:

hh = hours

mm = minutes

ss = seconds

The high-order half byte is always zero, and the low-order half byte is the sign, which is always positive.

The following entries:

| 1 | LABEL | △OPERATION△ | OPERAND | △ |
|---|-------|-------------|---------|---|
| | | 10 16 | | |
| | | GETIME | S | |

or

| | | | | |
|--|--|--------|--|--|
| | | GETIME | | |
|--|--|--------|--|--|

return the date and time in registers 0 and 1 in packed decimal format. You can then store the contents of these registers, and edit the fields for a printout of the date and exact time that an event occurred.

For example, let us assume you wish to print the date and exact time a job step is completed. The subroutine shown in Figure 8—1 gets the date and time from the job preamble and the SIB, unpacks the contents of registers 0 and 1 into an 18-byte buffer for editing and printing, then terminates the job step.

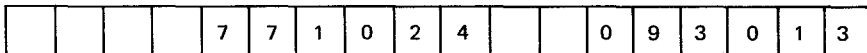
| 1 | LABEL | OPERATION | OPERAND | COMMENTS | 72 | 80 |
|-----|--------|-----------|-------------------------|--|----|----|
| 1. | GETIME | S | | Returns date in register 0 and time in register 1. | | |
| 2. | ST | | R0, WS1 | Stores date from register 0 to WS1. | | |
| 3. | ST | | R1, WS2 | Stores time from register 1 to WS2. | | |
| 4. | UNPK | | BUFFER+4(6), WS1(4) | Unpacks date into BUFFER. | | |
| 5. | UNPK | | BUFFER+12(6), WS2(4) | Unpacks time into BUFFER. | | |
| 6. | DT | | BUFFER+9, X'FO' | Changes contents of right hand byte of date from C1 to F1. | | |
| 7. | DT | | BUFFER+17, X'FO' | Changes contents of right hand byte of time from C3 to F3. | | |
| 8. | MVC | | BUFFER(2), BUFFER+4 | Inserts slashes, spaces, and periods in date and time. | | |
| 9. | MVI | | BUFFER+2, C'/' | | | |
| 10. | MVC | | BUFFER+3(2), BUFFER+6 | | | |
| 11. | MVI | | BUFFER+5, C'/' | | | |
| 12. | MVC | | BUFFER+6(2), BUFFER+8 | | | |
| 13. | MVI | | BUFFER+8, C' ' | | | |
| 14. | MVI | | BUFFER+9, C'.' | | | |
| 15. | MVC | | BUFFER+10(2), BUFFER+12 | | | |
| 16. | MVI | | BUFFER+12, C'.' | | | |
| 17. | MVC | | BUFFER+13(2), BUFFER+14 | | | |
| 18. | MVI | | BUFFER+15, C'.' | | | |
| 19. | PUT | | PRINT, BUFFER | Prints date and time from BUFFER. | | |
| 20. | EOJ | | | Terminates the job step. | | |

Figure 8-1. Example of GETIME Macro Instruction

Let us assume the GETIME macro instruction was executed October 24, 1977 at 13 seconds after 9:30 A.M. The job date from the preamble would be returned in register 0, and the time from the day clock in the SIB would be returned in register 1. The registers would contain:



Following execution of line 7, BUFFER contains:



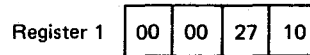
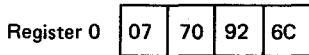
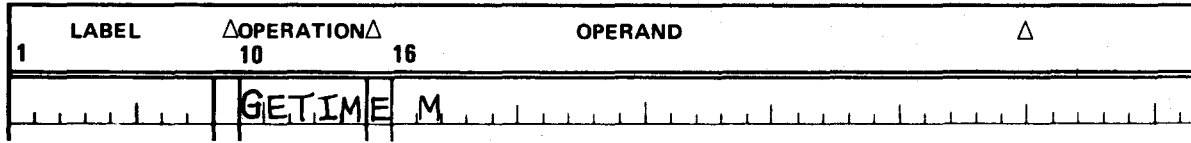
Following execution of line 18, BUFFER contains the date (year/mon/day) and the time (hours.min.sec):



The date and time are printed:

77/10/24 09.30.13

If you write this macro instruction using the M parameter, the date is expressed in packed decimal in register 0, but the time is expressed in milliseconds in binary representation in register 1. For example, if the following macro instruction were executed at 10 seconds after midnight, September 26, 1977, registers 0 and 1 would contain:



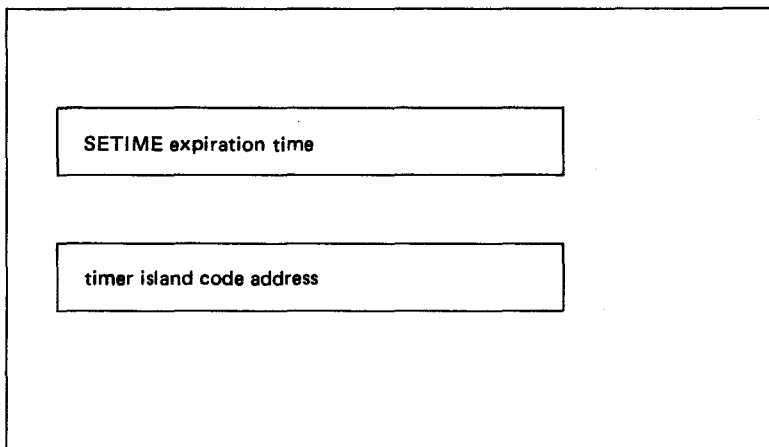
8.4.2. Timer Interrupt Facilities

The timer services module also enables you to request a scheduled timer interrupt in the requesting task. Using the SETIME macro instruction you may request an interrupt after any time period greater than 1 millisecond. You may:

- continue processing the task until the interrupt, then transfer control to the task's timer island code;
- suspend processing the task until the interrupt, then continue with the next instruction; or
- cancel a previous SETIME request.

The time interval requested in the SETIME macro instruction is added to the current time of day to calculate the time when the interrupt is scheduled to occur, and this SETIME expiration time is stored in the task control block (TCB).

TCB



If timer island code is to be executed, a STXIT macro instruction must have been previously issued to link the island code to this task. Timer island code is described in 8.6. If no timer island code is present, or if the interrupt request was cancelled, the interrupt is ignored. There may only be one set of timer island code per task.

If the task is to be suspended, the next available task in the switch list is executed. When the interrupt occurs, control is returned to the next instruction in the task immediately following the SETIME macro instruction.

8.4.2.1. Set Timer Interrupt (SETIME)

Function:

The SETIME macro instruction requests a scheduled timer interrupt in the requesting task and continues executing the requesting task. When the specified time interval elapses, the task's timer island code (as specified by a STXIT macro instruction) is executed.

Note that in this case the STXIT macro instruction must have been previously issued to set up timer island code for this task. There may be only one set of timer island code per task.

If written with the WAIT parameter, this macro instruction requests a timer interrupt and suspends execution of the requesting task until the timer interval elapses. At this time, the task resumes execution with the next instruction following the SETIME macro instruction.

This macro instruction cancels any previous SETIME request if entered with no parameters.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|---|
| [symbol] | SETIME | [{ time-interval } (1)] [,WAIT] [, { M } S } |

Positional Parameter 1:

time-interval

Specifies the interval of time that must expire before the interrupt is generated. This interval is expressed either in seconds or milliseconds depending on the entry in positional parameter 3. The maximum value that may be entered as positional parameter 1 is 4095₁₀. To specify a value greater than 4095, enter (1) as positional parameter 1 and preload register 1 with the required time interval value.

(1)

Indicates that register 1 has been preloaded with the time interval value.

If omitted, any previous SETIME request for this task is cancelled, preventing the scheduled interrupt.

Positional Parameter 2:

WAIT

Specifies that the problem program is to relinquish control until the specified time interval expires, at which time control is returned to the point immediately following the SETIME macro instruction.

If omitted, the requesting program retains program control. When the time interval expires, the timer island code is activated.

Positional Parameter 3:

M

Specifies that the time interval entered as positional parameter 1 is expressed in milliseconds.

S

Specifies that the time interval entered as positional parameter 1 is expressed in seconds.

If omitted, the parameter S is assumed.

8.4.2.2. Continue Processing Until Interrupt

If you omit the WAIT parameter, the task retains program control and continues processing at the instruction immediately following the SETIME macro instruction. When the time interval elapses, the timer island code for this task is executed. For example, the instruction:

| 1 | LABEL | △OPERATION△ 10 | 16 | OPERAND | △ |
|---|-------|-------------------|----|------------------|---|
| | | SETIME | 30 | ,,S | |
| | | | | next instruction | |
| | | . | | | |
| | | . | | | |
| | | . | | | |

or

| | | | | | |
|--|--|--------|----|------------------|--|
| | | SETIME | 30 | | |
| | | | | next instruction | |
| | | . | | | |
| | | . | | | |
| | | . | | | |

requests a timer interrupt in 30 seconds. The task continues processing until the 30-second time interval elapses; then the timer island code is executed.

If you want to specify an interval smaller than a second, the instruction:

| 1 | LABEL | Δ OPERATION Δ | OPERAND | Δ |
|---|-------|---------------|------------------|---|
| | | 10 | 16 | |
| | | SETIME | 200,1,M | |
| | | | next instruction | |
| | | . | | |
| | | . | | |
| | | . | | |

requests a timer interrupt in 200 milliseconds. The task continues processing until the 200-millisecond time interval elapses; then the timer island code is executed.

Figure 8-2 is an example of the use of the SETIME macro instruction to request an interrupt in 25 seconds so that a time limit of 25 seconds can be placed on the computation that follows.

| 1 | LABEL | Δ OPERATION Δ | OPERAND | Δ | COMMENTS | 72 |
|-----|-----------|---------------|----------------------|---|----------|----|
| | | 10 | 16 | | | |
| 1. | * TIMER | EXAMPLE | - | LIMIT COMPUTE LOOP TO 25 SECONDS | | |
| 2. | * | | | | | |
| 3. | * | | | ESTABLISH TIMER ISLAND CODE TO HANDLE INTERRUPT | | |
| 4. | | STXIT | IT, ILANDCOD, ICSAVE | | | |
| 5. | * | | | START TIMING INTERVAL | | |
| 6. | | SETIME | 25,1,S | TWENTY FIVE SECONDS | | |
| 7. | * | | | START OF COMPUTE LOOP | | |
| 8. | COMPUTE | EQU | * | | | |
| 9. | * | | | TEST TO SEE IF TIME LIMIT HAS BEEN EXCEEDED | | |
| 10. | | TM | FLAGBYTE, TIMEFLAG | TEST IF FLAG WAS SET BY ISLAND CODE | | |
| 11. | | BO | TOOLONG | BRANCH IF FLAG IS SET | | |
| 12. | . | | | } computation occurs here | | |
| 13. | . | | | | | |
| 14. | . | | | | | |
| 15. | | C | X, Y | TEST TO SEE IF COMPUTATION IS DONE | | |
| 16. | | BNE | COMPUTE | LOOP BACK IF NOT | | |
| 17. | * | | | NORMAL EXIT FROM COMPUTE LOOP | | |
| 18. | | STXIT | IT | DISABLES ISLAND CODE | | |
| . | . | | | } exit routine | | |
| . | . | | | | | |
| . | . | | | | | |
| 19. | * | | | ERROR IF COMPUTATION NOT DONE BEFORE TIME ELAPSES | | |
| 20. | TOOLONG | EQU | * | | | |
| 21. | | STXIT | IT | DISABLES ISLAND CODE | | |
| 22. | * | | | PRINT ERROR MESSAGES, ETC | | |
| 23. | . | | | } error print routine | | |
| 24. | . | | | | | |
| 25. | . | | | | | |
| 26. | * | | | TIMER ISLAND CODE - ACTIVATED WHEN TIME ELAPSES | | |
| 27. | I LANDCOD | EQU | * | | | |
| 28. | | OR | FLAGBYTE, TIMEFLAG | SET FLAG | | |
| 29. | | EXIT | IT | | | |
| 30. | * | | | WORK AREAS | | |
| 31. | ICSAVE | DS | 18F | REGISTER SAVE AREA REQUIRED | | |
| 32. | FLAGBYTE | DC | X'00' | INITIALLY ZERO | | |
| 33. | TIMEFLAG | EQU | X'01' | BIT #1 WHEN TIME ELAPSES | | |

Figure 8-2. Example of SETIME Macro Instruction

Line 4 links the timer island code (lines 27 to 29) which sets a flag when the time interval expires. Line 6 requests an interrupt in 25 seconds and the compute routine (lines 8 to 16) is entered. Line 18 is the normal exit which occurs if computation is completed before the time elapses. Lines 20 to 25 are the error routine which is executed if the time elapses before the computation is completed.

8.4.2.3. Wait for Interrupt

If you use the WAIT parameter, the task suspends processing and program control is transferred to the next available task. When the time interval elapses, program control is returned to the next instruction in the task immediately following the SETIME macro instruction. For example, the instruction:

| 1 | LABEL | ΔOPERATIONΔ | OPERAND | Δ |
|---|-------|-------------|----------------------------------|---|
| | | 10 | 16 | |
| | | SETIME | 30, WAIT | |
| | | | next instruction after interrupt | |
| | | . | | |
| | | . | | |
| | | . | | |

requests a timer interrupt in 30 seconds. The task is suspended until the 30-second time interval elapses, then processing continues with the next instruction. This instruction could be used following a message to the console operator or a question to a user at a remote terminal allowing a period of time (in this case, 30 seconds) to reply or to enter additional data.

8.4.2.4. Cancel a Previous Timer Interrupt Request

To cancel a previous timer interrupt request, simply use the SETIME macro instruction without parameters. For example:

| | | | |
|----|--|--------|------------------|
| 1. | | SETIME | 300, WAIT, M |
| 2. | | | next instruction |
| 3. | | . | |
| 4. | | . | |
| 5. | | . | |
| 6. | | SETIME | |

Line 1 requests a wait and timer interrupt in 300 milliseconds. Line 6 cancels the request. The next time these instructions are executed, the wait and interrupt requested by line 1 are ignored and program control goes immediately to line 2.

8.5. PROGRAM LINKAGE

A program may consist of several phases or routines produced by an assembler, compiler, or other language translator, and then combined by the linkage editor. Control can be passed from one routine to another within the program. This is referred to as direct linkage. Linkage can proceed through as many levels as necessary. During the execution of a job step, a routine (referred to as the calling program) passes control to another routine (the called program), which can in turn become the calling program passing control to a third routine (the called program), etc. This branch and linking process requires that the contents of certain registers be saved, then restored, so that control can be returned to the calling program.

The following macro instructions are used for direct linkage:

- **CALL/VCALL**
Calls a program module and gives it control.
- **ARGLST**
Generates an argument (parameter) list.
- **SAVE**
Saves the contents of specified registers.
- **RETURN**
Restores registers and returns control.

The CALL and VCALL macro instructions can also be used to pass parameters from the calling program to the called program.

8.5.1. Linkage Register Conventions

During the direct linkage process, certain registers are used for specific purposes to avoid conflicts in register use. These registers and their uses in the linkage procedure are:

- **Register 0** — Reserved for system use
- **Register 1** — Parameter or parameter list register
Register 1 is used for passing parameters between linked programs (each parameter is four bytes long and is aligned on a word boundary). This register is loaded with the parameter to be passed, or, in the case of a parameter list, the address of the first parameter in the list. The last parameter in a parameter list has its sign bit set to 1.
- **Register 2 through 12** — Free registers
These registers are never used or referenced by the direct linkage macro instructions.
- **Register 13** — Save area register
If a save area is provided for the called program by the calling program (for temporary register storage), the address of the save area, which must be aligned on a full-word boundary, is loaded in register 13 by the calling program.

- Register 14 — Return address register

This register is loaded by the calling program with the address to which control should be returned following the execution of the called program.

- Register 15 — Entry point register

This register is loaded by the calling program with the address of the entry point in the called program. This register can be used to provide initial addressability in the called program.

8.5.2. Linkage Procedure

The calling program establishes direct linkage with another program by means of the CALL or VCALL macro instruction. If registers are used in the called program (other than 0, 1, and 15), the SAVE macro instruction must be used to save their content. The RETURN macro is used to return control to the calling program.

The calling program is responsible for the following:

- Loading register 13 with the address of a 72-byte save area (if one is required by the called program). The save area must be aligned on a full-word boundary.
- Loading the parameter register, if necessary.
- Loading register 14 with the return address.
- Loading register 15 with the entry point in the called program.

The called program is responsible for the following:

- Saving the content of all registers used by it, with the exception of registers 0, 1, and 15 which are considered volatile. The contents of registers are saved in the area addressed by register 13.
- Following its execution, the called program must reload the saved registers and transfer program control to the return address loaded in register 14 by the called program.

You can have the CALL, VCALL, SAVE, and RETURN macro instructions perform the linkage functions for you. Or if you prefer, depending on how you code the parameters in the SAVE and RETURN macro instructions, you can perform some of these functions yourself.

If you use the SA parameters in the SAVE and RETURN macro instructions, the macro establishes a save area and loads the address of the save area into register 13. If you do not use the SA parameters, you must establish the save area in the calling program and load the address of the save area into register 13 before issuing the CALL or VCALL macro instruction.

If you use the COVER and COVADR parameters in the SAVE macro instruction, the macro loads the base register addresses. If you do not use the COVER and COVADR parameters, you must perform your own base register loading.

8.5.3. Register Save Area

A save area is established by one program (the calling program) for use by a second program (the called program). If the called program finds it necessary to use any of registers 2 through 14 thereby destroying their contents, the called program must store the original contents of these registers in the save area provided by the calling program, before using them. The called program itself can be a calling program, and must provide a save area for its called program (the third program in the chain). Any number of programs can be chained together in this manner. It is not necessary to have a save area in the last program of a chain.

Standard register save areas are used with the CALL, VCALL, SAVE, and RETURN macro instructions. Note that this register save area is different from the save area used with island code linkage for register and PSW storage (described in 8.6).

The format of the register save area is shown in Figure 8—3, and further explained in Table 8—1.

| Word | Byte | Content |
|------|------|------------------------------------|
| 1 | 0 | RESERVED FOR SYSTEM USE |
| 2 | 4 | SAVE AREA BACKWARD LINK ADDRESS |
| 3 | 8 | SAVE AREA FORWARD LINK ADDRESS |
| 4 | 12 | CALLING PROGRAM RETURN ADDRESS |
| 5 | 16 | CALLED PROGRAM ENTRY POINT ADDRESS |
| 6 | 20 | REGISTER 0 |
| 7 | 24 | REGISTER 1 |
| 8 | 28 | REGISTER 2 |
| 9 | 32 | REGISTER 3 |
| 10 | 36 | REGISTER 4 |
| 11 | 40 | REGISTER 5 |
| 12 | 44 | REGISTER 6 |
| 13 | 48 | REGISTER 7 |
| 14 | 52 | REGISTER 8 |
| 15 | 56 | REGISTER 9 |
| 16 | 60 | REGISTER 10 |
| 17 | 64 | REGISTER 11 |
| 18 | 68 | REGISTER 12 |

NOTE:

Each word in the save area is aligned on a full-word boundary.

Figure 8—3. Register Save Area Format

Table 8-1. Register Save Area

| Word | Content |
|--------|--|
| 1 | Reserved for system use. |
| 2 | If zero, indicates the first save area of a chain. Otherwise, this is the address of the save area used by the calling program which is located in the higher level program that called the calling program. For example, bytes 4-7 of SAVE B (a save area in program B for the use of program C) contains the address of SAVE A (a save area in program A for the use of program B). It is the responsibility of the calling program to store the backward link address in this field from register 13 before loading the current save area address in register 13. |
| 3 | If zero, indicates the last save area in a chain. Otherwise, this is the address of the save area in the most recently called program. It is the responsibility of this called program to store the save area address in this field before calling a lower level program. |
| 4 | The address in the calling program (as loaded in register 14) to which control is to be returned. This address must be stored in this field by the called program if that program intends to alter the contents of register 14. |
| 5 | The entry point address of the called program (as stored in register 15) to which control is to be transferred. This address must be moved to this field by the calling program. |
| 6 to 8 | A storage area provided to contain the contents of registers 0 through 12. The called program determines the number of registers, if any, to be saved. |

8.5.4. Call a Program (CALL/VCALL)

The CALL and VCALL macro instructions pass control from a program to a specified entry point in another program. They are written in the calling program to establish linkage with a called program. CALL is used to establish a direct linkage with a program already in main storage. It loads an A-type address constant, and branches. VCALL is used to establish a V-CON type linkage with a program not necessarily in main storage. It loads a V-type address constant, and branches. No SVCs are generated by either macro instruction.

The CALL or VCALL entry point need not have a manually coded EXTRN. All other labels used on these calls, which appear outside the assembly, must have manually coded EXTRNs.

You can use positional parameter 2 of the CALL or VCALL macro instruction to pass parameters from the calling program to the called program. In this case, you can enter the parameters themselves, enclosed in parentheses; the macro expansion will generate a parameter list in the required format. Or, you can enter the address of a parameter list defined elsewhere in your program in the format required by the macro.

Another convenient method is to use the ARGLST macro instruction to generate this list for you. You then enter the symbolic address of the macro call as positional parameter 2 of the CALL or VCALL macro instruction.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|-----------------------|--|
| [symbol] | { CALL } { VCALL } | { entry-point } (15) [{ (param-1,...,param-n) } list-address (1)] |

Positional Parameter 1:

entry-point

Specifies the symbolic address of the entry point in the called program to which program control is to be given.

(15)

Indicates that register 15 has been preloaded with the address of the called program.

Positional Parameter 2:

(param-1,...,param-n)

Specifies one or more parameters to be passed to the called program. These parameters are written enclosed in parentheses, and are included in the CALL or VCALL macro expansion in the same sequence as entered on the call line. Each parameter is considered as one full word and is aligned on a full-word boundary. The three low-order bytes of each generated word contain the address of a parameter. To mark the end of the parameter list, the sign bit of the last parameter in the list is set to 1. The address loaded in register 1, prior to control being transferred to the called program, is the address of the first parameter in the list.

The parameter entries can represent actual values. However, for compatibility with higher-level languages, this parameter is usually used to pass address constants to the called program.

list-address

Specifies the symbolic address of a user-defined parameter list. You can define the list in the required format, or you can use the ARGLST macro instruction to generate the list for you.

(1)

Indicates that register 1 has been preloaded with the address of the parameter list.

If omitted, no parameters are assumed.

Examples:



| 1 | LABEL | Δ OPERATION Δ | OPERAND | Δ |
|---|--------|---------------|-----------------------------|---|
| | | 10 | 16 | |
| | CIALLI | | TEST, (ADDR1, ADDR2, ADDR3) | |
| | CIALLI | | TEST, TESTADR | |
| | CIALLI | | SINE, (1) | |
| | CIALLI | | (1.5), (1) | |

8.5.5. Generate an Argument List (ARGLST)

The ARGLST macro instruction generates an argument list (list of parameters) in the format required by the CALL/VCALL macro instruction.

This is a declarative macro instruction and must not appear in a sequence of executable code.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|--------|---------------|-----------------------|
| symbol | ARGLST | param-1, ..., param-n |

Positional Parameter 1:

symbol

Specifies the symbolic address of the generated parameter list. This name can be used in the CALL/VCALL macro instruction to refer to the parameter list.

param-1, ..., param-n

Specifies one or more parameters to be included in the parameter list generated by this macro.

Example:

| | | |
|---------|--------|---------------------|
| TESTADR | ARGLST | ADDR1, ADDR2, ADDR3 |
|---------|--------|---------------------|



8.5.6. Save Register Contents (SAVE)

The SAVE macro instruction is written at the entry point of the called program. It saves the contents of the calling program registers, loads one or more base registers, establishes addressability, and sets the linking pointers of the save areas. All code is generated inline with no inner subroutine calls or SVCs.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|---|
| [symbol] | SAVE | $[(r1,r2)] [,T] \left[,COVER= \left\{ \begin{matrix} r \\ (r1,r2,\dots,rn) \end{matrix} \right\} \right]$ $\left[,COVADR= \left\{ \begin{matrix} \text{base-addr} \\ * \end{matrix} \right\} \right] [,SA=savearea-name]$ |

Positional Parameter 1:

(r1,r2)

Specifies that the registers designated in r1 through r2 are to be saved in the calling program save area. The registers are always stored in their respective fields of the save area. For example, if register 2 is specified, it is stored in word 8. All combinations of valid r1 and r2 register addresses are acceptable. If r1 > r2, the register addresses wrap around from 15 to 0. If register 13 is included within this range, it is ignored. However, if the SA keyword parameter is coded, the contents of register 13 are stored in the save area specified.

If omitted, no registers are saved by this parameter.

Positional parameter 2:

T

Specifies that if the return and entry point registers (14 and 15) are not saved by positional parameter 1, these registers are to be stored in the calling program save area in words 4 and 5.

If omitted, registers 14 and 15 are not saved by this parameter.

Keyword Parameter COVER:

The COVER and COVADR keyword parameters are used to establish addressability. The values specified by COVADR are loaded in the registers specified by COVER.

COVER=r

Specifies the register designated as base register for the called program.

COVER=(r1,r2,...,rn)

Specifies the registers to be designated as base registers. A total of nine registers can be designated.

If omitted, register 15 is assumed to be the base register.

Keyword Parameter COVADR:

COVADR=base-addr

Specifies the base address for the called program. If only one register is specified by the COVER keyword parameter, this base address is loaded in that register. If several registers are specified by the COVER keyword parameter, they are successively loaded with 4096 increments of COVADR. A USING statement is generated indicating the base address and all cover registers, regardless of whether this parameter is specified or omitted.

If omitted, the base address is assumed to be the address of this SAVE macro instruction, that is, the contents of the location counter at the time this macro instruction is assembled.

Keyword Parameter SA:

SA=savearea-name

Specifies the symbolic address of a 72-byte register save area. This address is loaded into register 13 after register 13 (which is assumed to contain the address of a previous save area if there is one) is stored in word 2 of the save area. This process provides linkage to a higher level save area if there is one.

If omitted, register 13 is unaltered.

Examples:

| 1 | LABEL | Δ OPERATION Δ | | OPERAND | Δ |
|---|-------|---------------|----|------------------------------|---|
| | | 10 | 16 | | |
| | SUB1 | SAVE | | (14,12),COVER=12 | |
| | | SAVE | | (14,12),SA=SAVEAREA,COVER=10 | |
| | | SAVE | | (14,12),COVADR=SUB2,SA=AREA | |

8.5.7. Restore Registers and Return (RETURN)

The RETURN macro instruction is written at the exit point of the called program. It restores the contents of the calling program registers, branches back to the calling program, and reserves storage for the current save area. All code is generated inline with no inner subroutine calls or SVCs.

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|---|
| [symbol] | RETURN | [(r1,r2) [T] [,SA= { savearea-name }]] |

Positional Parameter 1:

(r1,r2)

Specifies that the registers designated in r1 through r2 are to be restored from the calling program save area. The address of the save area is assumed to be in register 13. All combinations of valid r1 and r2 register addresses are acceptable. If $r1 > r2$, the register addresses wrap around from 15 to 0. If register 13 is included within this range, it is ignored. However, if the SA parameter is coded, register 13 is reloaded from word 2 of the save area before the registers are restored.

If omitted, no registers are restored by this parameter.

Positional Parameter 2:

T

Specifies that if the return and entry point registers (14 and 15) are not restored by positional parameter 1, these registers are to be restored from the calling program save area (words 4 and 5).

If omitted, registers 14 and 15 are not saved by this parameter.

Keyword Parameter SA:

The SA keyword parameter creates a 72-byte save area, or else it indicates that you have created the save area elsewhere in the routine. It reloads register 13 (from word 2 of this program's save area) with the pointer to the calling program's save area. It generates a branch via register 14 as the last executable instruction.

SA=savearea-name

Specifies the symbolic address of a 72-byte register save area to be created by this macro instruction.

SA=*

Specifies that you have defined a save area elsewhere in the routine.

If omitted, a save area is not created by this macro instruction, and register 13 is unaltered.

Examples:

| 1 | LABEL | ΔOPERATIONΔ | | OPERAND | Δ |
|---|-------|-------------|----|---------------------|---|
| | | 10 | 16 | | |
| | | RETURN | | (14,12) | |
| | | RETURN | | (14,12),T,SA=* | |
| | | RETURN | | (14,12),SA=SAVEAREA | |

8.6. ISLAND CODE LINKAGE

As you know, there are six levels of interrupts in OS/3. Two of these interrupts are handled by system routines; however, there are four interrupts that you must handle yourself. These interrupts are:

1. Program Check — An operation in your program causes a program check interrupt, such as an addressing error, arithmetic overflow, or operation exception.
2. Interval Timer — A time interval, which you specified using the SETIME macro instruction (WAIT parameter omitted), elapses.
3. Abnormal Termination — An error occurs that makes continuation of your program impossible.
4. Operator Communication — The operator entered an unsolicited message at the system console.

To handle these interrupts, you must write closed routines, called *island code*, and link these routines to tasks in your program. When one of these interrupts occurs, the supervisor stores the contents of the program status word (PSW) and general registers, and then transfers control to your island code routine. If you elect to resume processing the interrupted task, the supervisor uses this stored information to return control to the task at the point of interrupt.

The purpose of the program check, interval timer, and operator communication island code routines is to handle program contingencies or to notify your program that the interrupt has occurred. In the case of abnormal termination, the function of your island code routine is to terminate either a task or a job step rather than the entire job (normal procedure for abnormal termination if there is no abnormal termination island code routine).

The supervisor provides two macro instructions that automatically generate the linkages between your island code routine and your program. The macro instructions are:

- STXIT

Attach and detach your island code routine.

- EXIT

Exit from your island code routine.

You must use the STXIT macro instruction in your program to attach your island code routines to your tasks. You use the EXIT macro instruction in your program check, interval timer, and operator communication island code routines to return control to the interrupted task. Do not use the EXIT macro instruction in the abnormal termination island code routine. Instead, use:

- a DETACH macro instruction to detach the task;
- a DUMP or EOJ macro instruction to terminate the job step; or
- a CANCEL macro instruction to terminate the job.

8.6.1. Attaching Island Code to a Task (STXIT)

You use the STXIT macro instruction to attach island code routines to a task. An important point to remember is that STXIT only sets up the linkage, it does not call in the island code routine. Control passes to the island code routine only when the interrupt for which it was written occurs.

There are two formats for the STXIT macro instruction. One is for program check, abnormal termination, and interval timer island code routines; and one is for operator communication.

8.6.1.1. Attaching Program Check, Abnormal Termination, and Interval Timer Island Code

Function:

This form of the STXIT macro instruction establishes or terminates linkage between your task and the user island code routine specified by the parameters. If only parameter 1 is supplied, the previous linkage with the island code specified is terminated.

If a program check or an abnormal termination condition occurs for which no linkage is provided, the task is terminated. If the task is a primary task, the entire job is terminated; if it is a subtask, only the subtask is terminated.

If a timer interrupt occurs for which no linkage is provided, the interrupt is ignored.

Format:

The format for the STXIT macro instruction when it is used for program check, abnormal termination, or interval timer island code linkage is:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|--|
| [symbol] | STXIT | $\left\{ \begin{array}{c} \text{PC} \\ \text{AB} \\ \text{IT} \end{array} \right\} \left[\left\{ \begin{array}{c} \text{entry-point} \\ (1) \end{array} \right\} , \left\{ \begin{array}{c} \text{save-area} \\ (0) \end{array} \right\} \right]$ |

Positional Parameter 1:

PC

Establishes linkage with the program check island code routine.

AB

Establishes linkage with the abnormal termination island code routine.

IT

Establishes linkage with the interval timer island code routine.

If only positional parameter 1 is specified, the previous linkage with the particular user island code routine is terminated; otherwise, a linkage is established.

Positional Parameter 2:**entry-point**

Specifies the symbolic address of the entry point of the user island code routine that processes the interrupt.

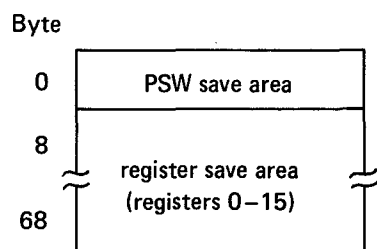
(1)

Indicates that register 1 has been preloaded with the address of the entry point.

If positional parameters 2 and 3 are omitted, the previous linkage with the island code specified in positional parameter 1 is terminated.

Positional Parameter 3:**save-area**

Specifies the symbolic address of an 18-word save area for PSW and general register storage. This save area must be aligned on a full-word boundary. The format for the save area is:



(0)

Indicates that register 0 has been preloaded with the address of the save area.

If positional parameters 2 and 3 are omitted, the previous linkage with the island code specified in positional parameter 1 is terminated.

As you can see from the format, parameters 2 and 3 are indicated as being optional. They are shown this way only because these parameters are omitted when you use the STXIT macro instruction to detach an island code routine (8.6.2). Remember, when attaching an island code routine, you must specify parameters 2 and 3; when you detach an island code routine, you must omit them. Examples of the STXIT macro instruction for program check, abnormal termination, and interval timer, are shown in 8.6.5, 8.6.6, and 8.6.7.

8.6.1.2. Attaching Operator Communication Island Code**Function:**

This form of the STXIT macro instruction establishes or terminates linkage between your task and the operator communication island code specified by the parameters. If only parameter 1 is supplied, the previous linkage with the operator communication island code is terminated.

If an unsolicited console message interrupt occurs for which no linkage is provided, the interrupt is ignored.

Format:

The format for the STXIT macro instruction when it is used for unsolicited operator communication linkage is:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|--|
| [symbol] | STXIT | OC [, { entry-point, save-area, msg-area, length }] (1) |

Positional Parameter 1:

OC

Establishes linkage with the operator communication island code routine.

If only positional parameter 1 is specified, the previous linkage with the operator communication island code routine is terminated; otherwise, a linkage is established.

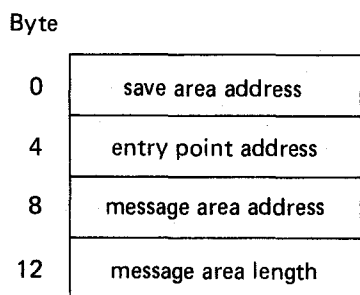
Positional Parameter 2:

entry-point

Specifies the symbolic address of the entry point of the operator communication user island code routine that processes the interrupt.

(1)

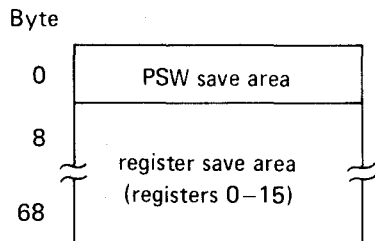
Indicates that register 1 has been preloaded with the address of a 4-word table containing parameters 2, 3, 4, and 5 in the following format:



Positional Parameter 3:

save-area

Specifies the symbolic address of an 18-word save area for PSW and general register storage. This save area must be aligned on a full-word boundary. The format for the save area is:



Positional Parameter 4:

msg-area

Specifies the symbolic address of an input area reserved for unsolicited messages from the operator.

Positional Parameter 5:

length

Specifies the length (in bytes) of the message area. The size of the area can be from 1 to 60 bytes; any message exceeding the specified length is truncated, while any message smaller is left-justified and space-filled.

8.6.2. Detaching Island Code From a Task (STXIT)

Function:

This form of the STXIT macro instruction terminates linkage between your task and the user island code routine specified by the parameter.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|---|
| [symbol] | STXIT | $\left. \begin{array}{c} \text{PC} \\ \text{AB} \\ \text{IT} \\ \text{OC} \end{array} \right\}$ |

Positional Parameter 1:

PC

Terminates linkage with the program check island code routine.

AB

Terminates linkage with the abnormal termination island code routine.

IT

Terminates linkage with the interval timer island code routine.

OC

Terminates linkage with the operator communication island code routine.

The specific island code routine remains in the program, but it is not entered the next time that type of interrupt occurs. Later in the program, if you want to attach the island code routine again, use the STXIT macro instruction with the same parameters or with other appropriate parameters. You may want to link another set of island code to the same task, in which case you would detach the old routine and attach the new. Remember, except for program check and interval timer island code in a multitasking environment, there can only be one current island code routine of one type in a job step, that is, one island code routine of one type currently linked to the task.

8.6.3. Island Code Entrance

As we have described earlier, you attach your island code routine with the STXIT macro instruction, specifying the type of island code routine, the routine's entry point, and a save area. When the event occurs for which your routine was written, the instruction being executed at that time completes, and the PSW and the general register contents are stored in the save area. Control is then transferred to your island code routine. If the last instruction in the routine is an EXIT macro instruction, the supervisor uses the stored PSW and general register contents to return control to the interrupted task at the instruction following the point of interrupt.

Program check, abnormal termination, and interval timer island code routines receive control under the task control block (TCB) of the task, or subtask, causing the interrupt. Operator communication island code routines receive control under the TCB of the primary task. When your island code routine is activated, the contents of the PSW and the register save area of the TCB are moved to the island code routine's save area. Your island code routine should not change entries in this save area. If it does, the state of the system and your program are different after the interrupt is processed than it was before the interrupt occurred. You may not be able to resume program execution or you may get erroneous results. Floating point registers are undisturbed from the time of interrupt; however, any changes made during island code routine execution are returned to the interrupted task.

Your island code routines are given control by the task switcher even though the associated task is in a wait state. This override of normal waits (e.g., wait for I/O synchronization, or for interval timer) is referred to as *island code override* and remains in effect during island code execution. For example, your task can issue a wait for an I/O operation and immediately enter island code regardless of whether the wait has completed. When you exit from the island code routine, the island code override is removed; but if the I/O wait is still set, your program cannot return to the interrupted task until the I/O operation has completed. Also, if a wait within your island code routine is followed by an EXIT macro instruction, control is transferred immediately to the interrupted task even though this wait is still set. If another interrupt occurs, control is not transferred from your task to your island code routine until the wait within the island code routine has elapsed.

8.6.4. Island Code Exit (EXIT)

At the close of your island code routine, you can:

- use the EXIT macro instruction to return control to the interrupted task; or
- use the DETACH, EOJ, DUMP, or CANCEL macro instruction to terminate the task.

8.6.4.1. Exiting From Program Check, Interval Timer, and Operator Communication Island Code

The normal procedure for program check, interval timer, and operator communication island code is to return control to the interrupted task. You do this by coding the EXIT macro instruction as the last executable instruction of the island code routine.

Function:

The EXIT macro instruction terminates a user island code routine, restores the contents of the registers and the PSW, and returns program control to the point immediately following the interrupt. This macro instruction must be the last executable instruction within the island code routine (except for abnormal termination).

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|--------------------|
| [symbol] | EXIT | { PC IT OC } |

Positional Parameter 1:**PC**

Specifies that exit is from the program check island code routine.

IT

Specifies that exit is from the interval timer island code routine.

OC

Specifies that exit is from the operator communications island code routine.

8.6.4.2. Exiting from Abnormal Termination Island Code

You do not have the option to return to the interrupted task from abnormal termination island code. However, you do have a choice of four macro instructions. You may use the DETACH, EOJ, DUMP, or CANCEL macro instruction. The use of these macro instructions to terminate abnormal termination island code is described in 8.6.6.

8.6.5. Program Check

Your program check island code routine receives control as the result of a hardware program check interrupt. The island code routine gains control at the entry point specified in the STXIT macro instruction in your program that linked the island code to the task. At this time, the least significant eight bits of register 0 contains an error code, and register 1 contains the address of the event control block (ECB) of the task causing the interrupt. A value of zero in register 1 indicates a primary task, otherwise it is the address of the ECB of a subtask. All other registers are as they were when the task was interrupted.

The program check error code returned in register 0 does not necessarily indicate an error condition since occurrences such as arithmetic overflow can cause the interrupt. These codes, which range from hexadecimal 01 to 0F, are listed and described in the system messages programmer/operator reference, UP-8076 (current version).

Program check island code enables you to take some corrective action so that a program check interrupt does not cause abnormal termination of the job step. You can take whatever action is necessary to correct the situation, then return to the interrupted task by executing the EXIT macro instruction.

If a program check interrupt is caused by a task for which there is no program check island code routine, or the island code routine was detached using a STXIT macro instruction with only the first parameter, the task enters abnormal termination island code with an error code of hexadecimal 20. If there is no abnormal termination island code to handle the situation, the task is abnormally terminated. When the task is a primary task, the entire job is terminated; when it is a subtask, only the subtask is terminated.

Now let us look at how you would use the STXIT macro instruction with symbolic addresses. Figure 8-4 illustrates this.

| 1 | LABEL | ΔOPERATIONΔ | OPERAND | Δ |
|-----|--------|-------------|-----------------|-----------------------|
| | | 10 | 16 | |
| 1. | | LR | R3, R7 | |
| 2. | | . | | |
| 3. | | . | | |
| 4. | | . | | |
| 5. | | MH | R3, TRAJ | |
| 6. | PCOVFL | STXIT | PC, AROVFL, PCI | |
| 7. | | ST | R3, MAXTRAJ | |
| 8. | | . | | |
| 9. | | . | | |
| 10. | | . | | |
| 11. | | L | R6, MAXWGT | |
| . | | . | | |
| . | | . | | |
| . | | . | | |
| 57. | AROVFL | C | R1, TESTA | } island code routine |
| 58. | | . | | |
| 59. | | . | | |
| 60. | | . | | |
| 61. | | EXIT | PC | |
| 62. | PCI | DS | 18F | |

Figure 8-4. Example of Program Check Island Code Linkage Using Symbolic Addresses

In this example, we've coded a program check STXIT in line 6. The entry point address is AROVFL and the save area address is PC1. The STXIT macro instruction is not part of the island code routine, nor does it call the island code routine. It only attaches the island code routine to the task. The island code routine is coded in lines 57 through 61. You should place the island code routine in the nonexecutable portion of your program. Nothing, however, prevents you from coding it inline in your program. If you do this, you must unconditionally branch around the island code routine. The reason for this is that you want the island code routine executed only when a program check interrupt occurs, not every time it is encountered in the main line of your program. Line 62 reserves the main storage save area needed by the island code routine.

From the format, you can see that you can also code STXIT using register addresses instead of symbolic addresses. Let's take a look at the same program using the alternate method of coding STXIT, as shown in Figure 8-5.

| 1 | LABEL | △OPERATION△ | | OPERAND | △ |
|-----|--------|-------------|----|--------------|-----------------------|
| | | 10 | 16 | | |
| 1. | | LR | | R3, R7 | |
| 2. | | . | | | |
| 3. | | . | | | |
| 4. | | . | | | |
| 5. | | MH | | R3, TRAJ | |
| 6. | | LA | | R1, AROVFL | |
| 7. | | LIA | | R0, PCI | |
| 8. | PCOVFL | STXIT | | PC, (1), (0) | |
| 9. | | ST | | R3, MAXTRAJ | |
| 10. | | . | | | |
| 11. | | . | | | |
| 12. | | . | | | |
| 13. | | L | | R6, MAXWGT | |
| . | | . | | | |
| . | | . | | | |
| . | | . | | | |
| 59. | AROVFL | C | | R1, TESTA | } island code routine |
| 60. | | . | | | |
| 61. | | . | | | |
| 62. | | . | | | |
| 63. | | EXIT | | PC | |
| 64. | PC1 | DS | | 18F | |

Figure 8-5. Example of Program Check Island Code Linkage Using Register Addresses

Except for three lines of coding, the programs are identical. In order to use register addresses in the STXIT macro instruction, you must preload them. Register 1 must be preloaded with the entry point address (line 6) and register 0 with the save area address (line 7). When you code the STXIT macro instruction in line 8, you simply write the register numbers as shown in the format.

When the STXIT macro instruction is encountered, the supervisor takes the addresses in registers 0 and 1 and stores them in a control table. These entries in the control table are referenced when the interrupt occurs and the island code routine is needed. Once the addresses in the registers are stored, these registers are freed. It is advisable to code the *load address* instructions immediately preceding the STXIT macro instruction because these registers are frequently used by the system and their contents are dynamic. Other than the exceptions just noted, the points brought out in the previous example about island code routine placement and reserving main storage still apply.

8.6.6. Abnormal Termination

Abnormal termination island code is similar to program check island code in that an interrupt can occur at any time during the execution of the task; however, the action to be taken differs radically. Your program check island code routine must return control to the interrupted task; your abnormal termination island code routine cannot.

Abnormal termination island code receives control when a task enters cancel processing. The cancel can be either intentional (execution of a CANCEL macro instruction) or unintentional, as with a system imposed cancellation due to a software detected error. This permits you to intervene to prevent the abnormal termination of a job. For example, the operating system can abnormally terminate a job because of a physical IOCS error. Instead, you may prefer to terminate the job step in error, but process the next job step of the job. Or, in the case of a subtask causing the abnormal termination interrupt, you may want to detach only the subtask in error, and continue processing the remaining active subtasks or the primary task.

Your abnormal termination island code gains control at the entry point specified in the STXIT macro instruction that linked the island code routine to the job step. At this time, the least significant 12 bits of register 0 contain an error code, and register 1 contains the address of the ECB of the task causing the cancellation. A value of 0 in register 1 indicates a primary task, otherwise, it is the address of the ECB of a subtask.

The error codes that may cause cancellation are listed and described in the system messages programmer/operator reference, UP-8076 (current version). Because you cannot return to the interrupted task, you cannot use the EXIT macro instruction to exit from abnormal termination island code. Instead you may use any of the following macro instructions to terminate the task:

- DETACH

Terminate the task or subtask normally.

- EOJ

Terminate the job step normally.

■ DUMP

Print out the job region contents and terminate the job step normally.

■ CANCEL

Print out the job region contents and terminate the job abnormally.

The EOJ macro instruction is described in 8.3.4, CANCEL in 8.3.5, DUMP in 9.1.2, and DETACH in 7.3.3.

If an abnormal termination interrupt is caused by a task for which there is no abnormal termination island code routine, or the island code routine was detached, the task is abnormally terminated. If the task is a primary task, the entire job is terminated; if it is a subtask, only the subtask is terminated.

If a program check interrupt is caused by a task for which there is no program check island code routine, or the island code routine was detached, the task enters the abnormal termination island code routine with an error code of hexadecimal 20. If there is no abnormal termination island code routine or the island code routine was detached, the job is abnormally terminated.

Figure 8-6 is an example of how you use the STXIT macro instruction to attach abnormal termination island code routine to your task. Note that in this case we have chosen to use the DUMP macro instruction to exit from the island code routine.

| 1 | LABEL | ΔOPERATIONΔ | OPERAND | Δ |
|-----|---------|-------------|---------------------|-----------------------|
| | | 10 | 16 | |
| 1. | | LR | R4, R6 | |
| 2. | | . | | |
| 3. | | . | | |
| 4. | | . | | |
| 5. | | STXIT | AB, ABENTRY, ABSAVE | |
| 6. | | . | | |
| 7. | | . | | |
| 8. | | . | | |
| 9. | RDREC | GET | DATAFIL, PARNAM | |
| . | | . | | |
| . | | . | | |
| . | | . | | |
| 40. | ABENTRY | C | R0, TESTCODE | } island code routine |
| 41. | | . | | |
| 42. | | . | | |
| 43. | | . | | |
| 44. | | DUMP | 369 | |
| 45. | ABSAVE | DS | 18F | |

Figure 8-6. Example of Abnormal Termination Island Code Linkage Using Symbolic Addresses

In this example, we've coded an abnormal termination STXIT macro instruction in line 5. The entry point address is ABENTRY and the save area address is ABSAVE. As we mentioned earlier, the STXIT macro instruction is not part of the island code routine. The island code routine, which is coded in lines 40 through 44, is executed only when an abnormal termination interrupt occurs. Line 45 reserves the main storage save area needed by the island code routine.

8.6.7. Interval Timer

Your interval timer island code routine receives control as the result of a timer interrupt requested by a SETIME macro instruction in your program written without the WAIT parameter.

When the time interval specified in the SETIME macro instruction elapses, your interval timer island code routine gains control at the entry point specified in the STXIT macro instruction that linked the island code routine to the task. At this time, register 0 is cleared to 0, and register 1 contains the address of the ECB of the task for which the time interval elapsed. A value of 0 in register 1 indicates a primary task, otherwise, it is the address of the ECB of a subtask. All other registers are as they were when the task was interrupted.

To exit from your interval timer island code routine, use the EXIT macro instruction to return to the interrupted task.

Remember, there must be a SETIME macro instruction without the WAIT parameter to request an interval timer interrupt, and a STXIT macro instruction in the same task to attach the task to the island code routine that handles the interrupt. If an interval timer interrupt occurs in a task for which there is no interval timer island code routine, or the interval timer island code routine was detached, the interrupt is ignored.

Figure 8-7 is an example of the use of the SETIME, STXIT, and EXIT macro instructions with an interval timer island code routine.

| 1 | LABEL | Δ OPERATION Δ | OPERAND | Δ | COMMENTS | 72 |
|----|---------|---------------|----------------------|---|---|----|
| 1 | * TIMER | EXAMPLE | - | | LIMIT COMPUTE LOOP TO 25 SECONDS | |
| 2 | * | | | | | |
| 3 | * | | | | ESTABLISH TIMER ISLAND CODE TO HANDLE INTERRUPT | |
| 4 | | STXIT | IT, ILANDCOD, ICSAVE | | | |
| 5 | * | | | | START TIMING INTERVAL | |
| 6 | | SETIME | 25,, S | | TWENTY FIVE SECONDS | |
| 7 | * | | | | START OF COMPUTE LOOP | |
| 8 | COMPUTE | EQU | * | | | |
| 9 | * | | | | TEST TO SEE IF TIME LIMIT HAS BEEN EXCEEDED | |
| 10 | | TIM | FLAGBYTE, TIMEFLAG | | TEST IF FLAG WAS SET BY ISLAND CODE | |
| 11 | | BO | TOOLONG | | BRANCH IF FLAG IS SET | |
| 12 | | . | | | } computation occurs here | |
| 13 | | . | | | | |
| 14 | | . | | | | |
| 15 | | C | X, Y | | TEST TO SEE IF COMPUTATION IS DONE | |
| 16 | | BNE | COMPUTE | | LOOP BACK IF NOT | |
| 17 | * | | | | NORMAL EXIT FROM COMPUTE LOOP | |
| 18 | | STXIT | IT | | DETACH ISLAND CODE | |
| . | . | . | | | } exit routine | |
| . | . | . | | | | |
| . | . | . | | | | |

Figure 8-7. Example of Interval Timer Island Code Linkage Using Symbolic Addresses (Part 1 of 2)

| 1 | LABEL | Δ OPERATION Δ | OPERAND | Δ | COMMENTS | 72 |
|-----|-----------|---------------|--------------------|---|---|----|
| | 10 | 16 | | | | |
| 19. | * | | | | ERROR IF COMPUTATION NOT DONE BEFORE TIME ELAPSES | |
| 20. | TOOLONG | EQU | * | | | |
| 21. | | STXIT | IT | | DETACH ISLAND CODE | |
| 22. | * | | | | PRINT ERROR MESSAGES, ETC | |
| 23. | | . | | | } error print routine | |
| 24. | | . | | | | |
| 25. | | . | | | | |
| 26. | * | | | | TIMER ISLAND CODE - ACTIVATED WHEN TIME ELAPSES | |
| 27. | ISLANDCOD | EQU | * | | | |
| 28. | | DI | FLAGBYTE, TIMEFLAG | | SET FLAG | |
| 29. | | EXIT | IT | | | |
| 30. | * | | | | WORK AREAS | |
| 31. | ICSAVE | DS | 18F | | REGISTER SAVE AREA REQUIRED | |
| 32. | FLAGBYTE | DC | X'00' | | INITIALLY ZERO | |
| 33. | TIMEFLAG | EQU | X'01' | | BIT 1 WHEN TIME ELAPSES | |

Figure 8-7. Example of Interval Timer Island Code Linkage Using Symbolic Addresses (Part 2 of 2)

In this example, the SETIME macro instruction (line 6) requests a timer interrupt in 25 seconds so that a time limit of 25 seconds can be placed on the computation (lines 8 to 16) that follows. The STXIT macro instruction (line 4) attaches the interval timer island code routine (lines 27 to 29) to the task. The routine sets a flag when the time interval expires. The STXIT macro instruction is used again (lines 18 and 21) to detach the island code routine. The EXIT macro instruction (line 29) returns control from the island code routine to the interrupted task. Line 18 is the normal exit from the compute loop, which occurs if computation is completed before the timer elapses. Lines 20 and 25 are the error routine which is executed if the time elapses before the computation is completed. Line 31 defines the save area needed when the interrupt occurs.

8.6.8. Operator Communication

Your operator communication island code routine receives control when the operator enters an unsolicited message at the system console. He does this by typing the job number and a zero, followed by the message text. For additional details of the operating procedure at the system console, refer to the appropriate operations handbook for your system.

You can use the WTLD and OPR macro instructions to communicate with the operator. In these cases, your program displays a message on the system console and requests a reply. However, the use of operator communication island code routines permits the operator to enter a message for the attention of your program at any time during the execution of a job step without being prompted by your program. He could enter one of several predefined messages to acknowledge an event or a condition external to your program, for example, an infrequent request for statistics at the end of a particular job step.

The island code routine gains control at the entry point specified in the STXIT macro instruction that linked the island code routine to the job step. At this time, register 0 contains the length of the message entered by the operator, while the contents of register 1 are unpredictable. (Register 1 would not contain an ECB address because operator communication island code routines always execute under the primary task TCB.)

To exit from operator communication island code, use the EXIT macro instruction to return to the interrupted task.

If the operator attempts to enter an unsolicited message for a job step for which there is no operator communication island code routine, or the island code routine has been detached, the message is rejected.

Figure 8-8 is an example of the use of the STXIT and EXIT macro instructions for operator communication island code routine using symbolic addresses. The general operation is similar to that described for program check (8.6.5). However, you will note that, in addition to the entry point and save area, the STXIT macro instruction also specifies a message area and message length.

Following the format, the STXIT macro instruction in line 21 specifies that it is attaching an operator communication island code routine (OC). The island code routine's entry point is SYSCON, the save area address is OC1, and the message from the system console is stored in a reserved 60-byte area whose address is OPRMSG.

| 1 | LABEL | Δ OPERATION Δ | OPERAND | Δ |
|-----|--------|---------------|-----------------------------|-----------------------|
| | | 10 | 16 | |
| 1. | | LR | R3, R7 | |
| . | | . | | |
| . | | . | | |
| . | | . | | |
| 21. | | STXIT | OC, SYSCON, OC1, OPRMSG, 60 | |
| . | | . | | |
| . | | . | | |
| . | | . | | |
| 75. | SYSCON | LR | R5, OPRMSG+3 | } island code routine |
| . | | . | | |
| . | | . | | |
| . | | . | | |
| 89. | | EXIT | OC | |
| 90. | OC1 | DS | 18F | |
| 91. | OPRMSG | DS | 15F | |

Figure 8-8. Example of Operator Communication Island Code Linkage Using Symbolic Addresses

Figure 8—9 is an example of how your program would look if you elect to use register addresses instead of symbolic addresses. The *load address* instruction in line 21 places the address of a 16-byte area, called OCSETUP, into register 1. The format of this area is:

| | |
|----|----------------------|
| 0 | save area address |
| 4 | entry point address |
| 8 | message area address |
| 12 | message area length |

What we have done is taken the last four parameters of the STXIT OC format and converted them to a short table. This short table is referenced as the (1) parameter in line 22. Line 93 reserves the main storage area for this table. Note that in the table the *save area address* is the first field and the *entry point address* is the second field. Do not confuse this with how these parameters are listed in the STXIT macro instruction if you are using symbolic addresses. Remember also that at the time the operator communication interrupt occurs, this 16-byte field (OCSETUP) must contain the appropriate addresses and message area length.

| 1 | LABEL | △OPERATION△ | | OPERAND | △ |
|-----|---------|-------------|----|--------------|-----------------------|
| | | 10 | 16 | | |
| 1. | | LR | | R3, R7 | |
| . | | . | | | |
| . | | . | | | |
| . | | . | | | |
| 21. | | LA | | R1, OCSETUP | |
| 22. | | STXIT | | OC, (1) | |
| . | | . | | | |
| . | | . | | | |
| . | | . | | | |
| 75. | SYSCON | LR | | R5, OPRMSG+3 | } island code routine |
| . | | . | | | |
| . | | . | | | |
| . | | . | | | |
| 90. | | EXIT | | OC | |
| 91. | OC1 | DS | | 18F | |
| 92. | OPRMSG | DS | | 15F | |
| 93. | OCSETUP | DS | | 16F | |

Figure 8—9. Example of Operator Communication Island Code Linkage Using Register Addresses

8.6.9. Use of Island Code With Multitasking

8.6.9.1. Program Check and Interval Timer With Multitasking

In a multitasking environment, you can specify discrete program check and interval timer island code routines using a separate STXIT macro instruction in each task to link the task to its island code routine.

Figure 8—10 depicts a job step with three tasks (the primary task and two subtasks). The STXIT macro instruction in each task specifies a separate island code routine, and a separate save area. Note that upon exit, control returns to the interrupted task at the point of interrupt.

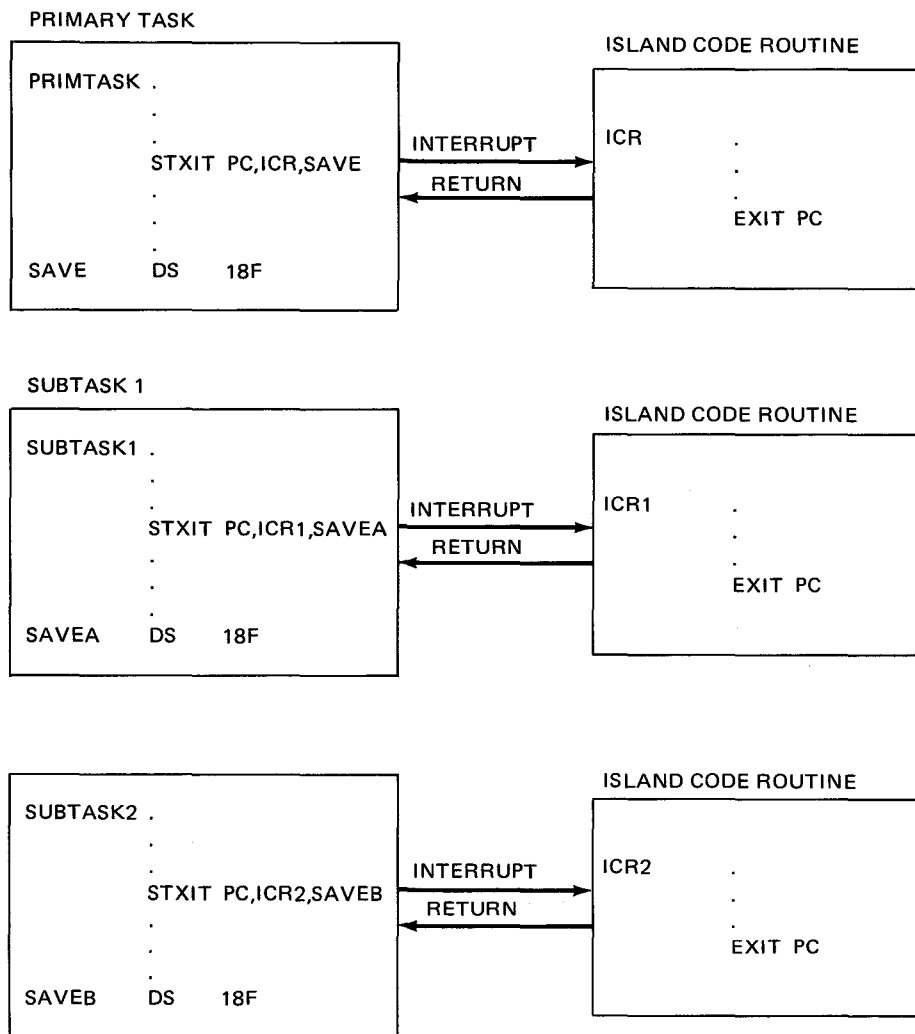


Figure 8—10. Example of Discrete Program Check Island Code for Each Task in a Job Step

You can also use a common program check island code routine or a common interval timer island code routine for all the tasks within a job step. In this case, you use a separate STXIT macro instruction for each task to link the task to the common island code routine, specifying the same entry point but with a different save area for each task. When you use a common island code routine, the island code routine must be reentrant, that is, it can't make any changes to itself or to its common parts.

Remember, this common island code routine can be entered from any of its associated tasks and program control returns to the interrupted task via the EXIT macro instruction. Also, make sure you don't disturb the affected save area because the task environment must be restored so that control can be returned to the interrupted task.

Figure 8-11 shows how all the tasks in a job step (in this case, a primary task and two subtasks) could use a common island code routine. The STXIT macro instruction in each task specifies the same entry point; however, each STXIT specifies a separate save area. When a program check interrupt occurs in any of the tasks, control is transferred to the one island code routine. Upon exit, control returns to the interrupted task at the point of interrupt.

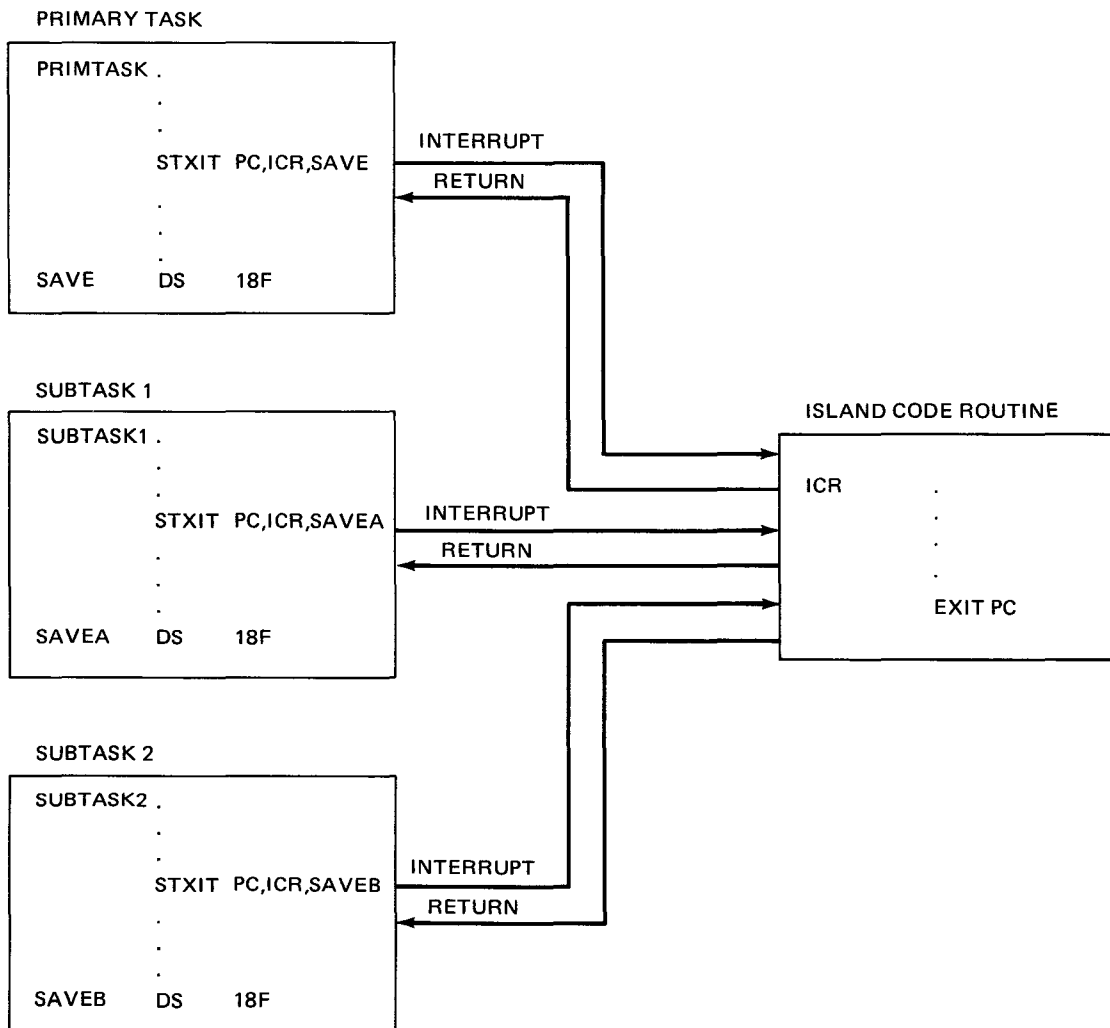


Figure 8-11. Example of Common Program Check Island Code for All Tasks in a Job Step

8.6.9.2. Abnormal Termination With Multitasking

There can be only one abnormal termination island code routine current in a job step at any one time. You can use a STXIT macro instruction in any task to attach the island code routine. The abnormal termination island code routine is associated with the job preamble; however, it may be entered from any task in the job step.

If several tasks in a job step are each causing an abnormal termination interrupt, these cancellation requests are queued for entry into the one abnormal termination island code routine for the job step.

You can have several abnormal termination island code routines in a job step (for example, one for each overlay), but only the routine linked by the current STXIT macro instruction is effective when an interrupt occurs. In this case, each succeeding STXIT macro instruction supersedes the previous one, and you do not have to issue a STXIT macro instruction to detach each previous island code routine.

8.6.9.3. Operator Communication With Multitasking

There can be only one operator communication island code routine current in a job step at one time. You can use a STXIT macro instruction in any task to attach the island code routine. The operator communication island code routine is associated with the primary TCB; however, it may be entered at any time regardless of which task is processing at the time the operator enters the job number and a zero at the system console to cause an operator communication interrupt.

Multiple activations of an operator communication island code routine are not possible. If the island code routine is executing, it must exit before it can be reentered. If the island code routine is handling an operator communication interrupt when the operator attempts to enter another unsolicited message for the same job step, the later unsolicited message is rejected.

As is the case with an abnormal termination island code routine, you can have several operator communication island code routines in a job step, but only the code linked by the current STXIT macro instruction is effective when an interrupt occurs.

8.7. SYSTEM INFORMATION CONTROL

Each problem program is assigned a variable-length storage area within the program region which is known as the job prologue and contains the job preamble and task control blocks. You can retrieve or read information from the job prologue only through the supervisor. In addition, you can establish, change, or cancel information only within the 12-byte communication region of the job preamble. You cannot alter any other part of the contents of these critical storage areas. The communication region is most commonly used to pass information from one job step to the next; 12 bytes of data can be stored by one job step and retrieved by subsequent job steps associated with the same job. The user program switch indicator (UPSI) can be retrieved using the GETCOM macro instruction or set using the PUTCOM macro instruction. The UPSI is the last byte in the 12-byte communication region in the job preamble and is tested by a subsequent SKIP job control statement. The job control user guide, UP-8065 (current version) contains a description of the UPSI bit values, how to set and change the bits, and how to use the UPSI to branch around JCL statements.

The following macro instructions are provided to assist you in accessing these restricted storage areas:

■ GETCOM

Retrieves the contents of the 12-byte communication region from within the job preamble.

■ PUTCOM

Writes a 12-byte character string into the communication region within the job preamble.

■ GETINF

Retrieves information from the SIB, PUBs, TCBs, or preamble.

8.7.1. Get Data From Communication Region (GETCOM)

Function:

The GETCOM macro instruction retrieves the contents of the 12-byte communication region from within the job preamble and stores it in an area specified in positional parameter 1:

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|--------------------|
| [symbol] | GETCOM | { to-addr } (1) |

Positional Parameter 1:

to-addr

Specifies the symbolic address of a 12-byte area in main storage to which the contents of the communication region is to be moved.

(1)

Indicates that register 1 has been preloaded with the address of the area in main storage.

8.7.2. Put Data Into Communication Region (PUTCOM)

Function:

The PUTCOM macro instruction moves the contents of a 12-byte area in main storage specified in positional parameter 1 to the communication region within the job preamble.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|----------------------|
| [symbol] | PUTCOM | { from-addr } (1) |

Positional Parameter 1:

from-addr

Specifies the symbolic address of a 12-byte area in main storage containing the data characters to be moved into the communication region within the job preamble.

(1)

Indicates that register 1 has been preloaded with the address of the area in main storage.

8.7.3. Get Data From System Control Tables (GETINF)

Function:

The GETINF macro instruction retrieves data from the task control block (TCB), systems information block (SIB), physical unit block (PUB), or the job preamble and stores it in a work area in main storage specified in positional parameter 2.

NOTE:

If you use the GETINF macro instruction in your program, you must reassemble your program upon every major release of the system software.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|---|
| [symbol] | GETINF | { TCB } { SIB } { PRE } { PUB } , { work-area } (1) , number-of-bytes, displacement |

Positional Parameter 1:**TCB**

Specifies that the data requested is from the job task control block.

SIB

Specifies that the data requested is from the systems information block.

PRE

Specifies that the data requested is from the job preamble.

PUB

Specifies that the data requested is from the physical unit block. In this case, register 1 must be preloaded with the address of the PUB or with the identifying number of the PUB. The PUB identifying number is its position within the PUBs specified at SYSGEN. That is, the first PUB is 0, the second PUB is 1, and so on.

Positional Parameter 2:**work-area**

Specifies the symbolic address of the work area in the problem program to which the data is to be moved. This area must be large enough to contain the data requested.

(1)

If positional parameter 1 is TCB, SIB, or PRE, indicates that register 1 has been preloaded with the address of the work area.

If positional parameter 1 is PUB, indicates that register 1 has been preloaded with the address of the PUB or with the identifying number of the PUB.

Positional Parameter 3:**number-of-bytes**

Specifies the number of bytes of data requested.

Positional Parameter 4:**displacement**

Specifies the displacement, that is, the number of bytes from the beginning of the table to the beginning of the data requested.

8.8. CONTROL STREAM READER

The control stream reader allows you to access data that was entered into the system with the job control stream. This provides a convenient method to handle small quantities of input that would normally have been handled as a card or diskette file. Because the data is embedded within the job control stream, there is no need to define a card file, nor is a device assignment set required for the card reader.

This embedded data might consist of transactions or changes to be processed against a master file, source code or control statements to be processed by a utility routine; or it might consist of PARAM job control statements to introduce parameters that can be used during program execution. Refer to the job control user guide, UP-8065 (current version) for a description of statements within embedded data.

When job control reads the job stream, it stores the embedded data in compressed form in the job's run library file. During the execution of the job step, this file is read into main storage and may be accessed by GETCS macro instructions (8.8.3) in your program. The requested records are expanded to their original form and stored in an input area you specify.

You can retrieve one or more records at a time from your embedded data file, and you can retrieve records from more than one set of embedded data belonging to the same job step. Images are read sequentially from the start of the entire data file. However, you can alter the sequence or reread data by using the SETCS macro instruction (8.8.5).

Each returned record is an exact image of the source statement which may be from 1 to 128 bytes. Thus, you can read 80- or 96-byte images from punched cards or 128-byte images from diskette.

NOTE:

Although PARAM and other job control statements may be handled as part of an embedded data set, they must still observe the job control statement conventions. Remember that job control statement information cannot extend past character position 71, and that position 72 is used to indicate continuation of a statement.

8.8.1. Embedded Data

Embedded data is delimited by a pair of /\$ (start-of-data) and /* (end-of-data) statements. They must follow the EXEC statement in the control stream or, if used, any PARAM statements. Note that PARAM statements are considered to be a part of the data set that follows. For example:

```
      // EXEC
      // PARAM
      // PARAM
Data  /$
Set  .
1    .
    .
    /*

Data // PARAM
Set  /$
2    .
    .
    .
    /*

Data /$
Set  .
3    .
    .
    /*
```

8.8.2. Reading Embedded Data

If you are reading one record at a time from this embedded data file, the first GETCS macro instruction executed retrieves the first PARAM statement of data set 1, the second retrieves the second PARAM statement, the third retrieves the /\$ statement, the fourth retrieves the first data card, etc.

Following the successful execution of a GETCS macro instruction, program control is returned to the issuing program at the point immediately following the GETCS macro instruction. Register 0 and 1 will contain:

- RO — The binary count of records retrieved.
 - O_{16} if a /* that terminates a data set is the first image in the input area.
 - $00FFFFFF_{16}$ when all embedded data images have been read.
- R1 — The reread pointer (8.8.4). When passed to the SETCS macro instruction (8.8.5), it allows you to reread embedded data at this pointer.

If two or more records are requested by a single GETCS macro instruction, the first occurrence of a real /* image terminates the control stream reader function. The /* is not returned until the next GETCS macro instruction call, at which time register 0 is set to zero and register 1 contains the reread pointer.

Since control streams may themselves be embedded data, the GETCS macro instruction indicates the end of a data set by signalling which end-of-data (/*) image actually terminates the data set. This is referred to as a real /* image as opposed to an embedded /* image. An embedded /* image is treated like any other image.

8.8.3. Get File From Control Stream (GETCS)

Function:

The GETCS macro instruction retrieves embedded data images and control statements that were entered in the system through the job control stream. You can retrieve one or more data images at a time from your embedded data file. The images may be from 1 to 128 bytes in length and may be obtained from more than one set of embedded data belonging to the same job step. Each retrieved record is an exact image of the source statement.

Images are read sequentially from the start of the entire data file. You can alter the sequence or reread data by using the SETCS macro instruction.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|--|
| [symbol] | GETCS | $\left\{ \begin{array}{c} \text{input-area} \\ (1) \end{array} \right\} \left[\begin{array}{c} \left\{ \begin{array}{c} \text{number-of-records} \\ (0) \\ \underline{1} \end{array} \right\} \\ \left[\begin{array}{c} \left\{ \begin{array}{c} \text{error-addr} \\ (r) \end{array} \right\} \\ \left[\begin{array}{c} \left\{ \begin{array}{c} n \\ \underline{80} \end{array} \right\} \end{array} \right] \end{array} \right] \end{array} \right]$ |

Positional Parameter 1:

input-area

→ Specifies the symbolic address of an input area in main storage that is to receive the records or records. When more than one record is requested at a time, as each record is retrieved from the control stream, it is stored in contiguous byte locations beginning with this address. This area must be large enough to contain the retrieved records. The record image size is specified in positional parameter 4.

(1)

Indicates that register 1 has been preloaded with the address of the main storage input area.

Positional Parameter 2:

number-of-records

Specifies the number of records requested.

(0)

Indicates that register 0 has been preloaded with the number of records requested.

If omitted, one record is assumed.

Positional Parameter 3:

error-addr

Specifies the symbolic address of an error routine to be executed if an error occurs.

(r)

Indicates that register r (other than 0 or 1) has been preloaded with the address of the error routine.

If omitted, the calling task is abnormally terminated if an error occurs.

Positional Parameter 4:

↓
n

Specifies the size of the data images to be retrieved. To retrieve the entire record, make sure this value equals the data stream record size.

If images smaller than n were originally written, the returned image will be left-justified and the remainder of the input area filled to the right with spaces. If images larger than n were originally written, only the number of bytes specified in this parameter will be returned and the remaining bytes in the data stream record will be ignored.

↑
If omitted, 80-byte images are retrieved. If smaller images were originally written, the returned image will be left-justified and space-filled to the right. If larger images were originally written, only the first 80 bytes will be returned.

8.8.4. Rereading Embedded Data

Following execution of a GETCS macro instruction, register 1 contains a full-word reread pointer consisting of the data set number, record displacement, and block number for the first record of the data just retrieved. If you intend to reread data, store this pointer in main storage and use the address of the pointer as parameter 1 of a SETCS macro instruction. A succeeding GETCS macro instruction will read the same data into your embedded data input area.

The pointer is advanced for every GETCS issued. If one image is requested, the pointer will point to the location in the data file of the record just returned. If more than one image is requested (parameter 2 of the GETCS macro instruction), the pointer will point to the location in the data file of the first record of the group of records just returned. For example, if an execution of a GETCS macro instruction has just returned five images, the reread pointer would point to the first image in the data file, not the fifth.

8.8.5. Reset Control Stream Reader (SETCS)

Function:

The SETCS macro instruction alters the sequence in which a subsequent GETCS macro instruction retrieves embedded data images from the job control stream. To do this, you may back up the GETCS pointer, skip backward or forward to the start of any embedded data set, or resume sequential reading of the data file at the beginning of the next data set.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|---|
| [symbol] | SETCS | $\left. \begin{array}{l} \text{pointer} \\ \text{data-set-no} \\ (1) \\ \text{NEXT} \end{array} \right\} \left[, \left\{ \begin{array}{l} R \\ S \end{array} \right\} \right] \left[, \left\{ \begin{array}{l} \text{error-addr} \\ (r) \end{array} \right\} \right]$ |

Positional Parameter 1:

pointer

Specifies the symbolic address of a full word embedded data file pointer provided by a previous GETCS macro instruction.

Upon successful completion of a GETCS macro instruction, control is returned to the program at the point immediately following the GETCS macro instruction, and register 1 contains a pointer to the last set of data images read from the embedded data file in the run library. When passed to the SETCS macro instruction, it allows embedded data to be reread starting at the pointer. Note that the pointer points to the *first* data image.

data-set-no

The number of the embedded data set from which subsequent GETCS macro instructions are to retrieve data images. Data sets are numbered sequentially starting with 1.

(1)

Indicates that register 1 has been preloaded with either the 4-byte GETCS pointer itself or with a data set number.

NEXT

Indicates that subsequent GETCS macro instructions are to retrieve data images starting at the beginning of the next data set.

Positional Parameter 2:

R

Specifies that the entry in positional parameter 1 is the address of the reread pointer provided by a previous GETCS macro instruction.

S

Specifies that the entry in positional parameter 1 is a data set number.

If omitted, the parameter S is assumed.

Positional Parameter 3:

error-addr

Specifies the symbolic address of an error routine to be executed if an error occurs.

(r)

Indicates that register r (other than 0 or 1) has been preloaded with the address of the error routine.

If omitted, the calling task is abnormally terminated if an error occurs.

8.8.6. Minimizing Disk Accesses

The job control stream embedded data reader operates as a transient within the supervisor and is called by the GETCS macro instruction. The transient is not replaced in main storage unless absolutely necessary. It can recognize whether or not the same user is recalling it. If so, there is no need to reread the embedded data file from disk unless the final record of the data file block was returned for the previous call. This reduces disk accesses while reading the embedded data.

Note that the use of a D option as positional parameter 4 is no longer more efficient than a GETCS without the D option. The D option is still supported even though it is pointless to use it. If your program contains GETCS macro instructions with the D option, it need not be changed.

9. Diagnostic and Debugging Aids

9.1. STORAGE DISPLAYS

Most programs don't run properly on the first try. Sometimes, there may only be minor coding errors, but other times, there may be logic errors. Coding errors are relatively easy to find, but logic errors tend to be elusive. This is why Sperry Univac has provided a method of obtaining printouts of main storage areas. These printouts are commonly called *dumps*.

Dumps are most helpful when you, not the operating system, control when they occur. This control is available through four macro instructions: CANCEL (8.3.5), SNAP, SNAPF, and DUMP. For any of these macros, however, a dump is not provided if:

- a printer is not assigned to the job; or
- an OPTION job control statement with the DUMP, JOBDUMP, or SYSDUMP parameter is not present in the job to override the default NODUMP condition of job control. ←

9.1.1. Snapshot Dumps (SNAP/SNAPF)

A snapshot dump is, by definition, a *selective dynamic dump* performed at various times in a run. The SNAP macro instruction produces this type of dump. It gives you a picture of the job's 16 general registers as well as selected areas of main storage.

There are really two macro instructions for obtaining snapshot dumps: SNAP and SNAPF. Each macro instruction performs the same function, except that the SNAPF macro instruction is used in the spooling environment. To simplify this discussion, whenever we mention the SNAP macro, we also mean the SNAPF macro instruction.

By using job control, you can initiate or suppress snapshot dumps at run time. You don't have to recompile a program in order to dump or not dump, since the SNAP macro instruction is only effective when combined with an OPTION job control statement (using the DUMP, JOBDUMP, or SYSDUMP parameter) in the job step in which you want the dump to occur. If the program is run without this OPTION job control statement, the SNAP macro instruction is bypassed. ←

A hexadecimal printout of the general registers and the job's main storage area is always given when the SNAP macro instruction executes. Program-relative addresses are listed on the left and absolute addresses are listed on the right. After the SNAP macro instruction executes, normal processing of the program continues. Control is returned to the instruction immediately following the SNAP macro instruction.

The SNAPF macro instruction works the same way as the SNAP macro instruction, but it allows you to direct the snapshot dump to a specified allocated printer or to a spool file via a virtual printer. In a spooling environment, you can use the SNAPF macro to direct snapshot dumps to a spool file other than the job log file. This enables you to obtain the printed output prior to job termination by using the spool breakpoint feature or closing the file.

The formats of the SNAP and SNAPF macro instructions are:

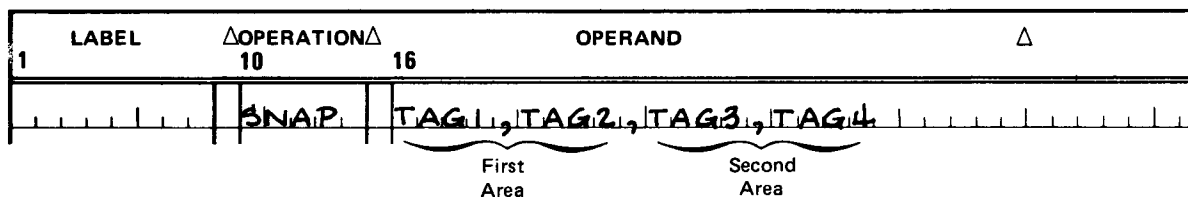
| LABEL | Δ OPERATION Δ | OPERAND |
|----------|-----------------------|---|
| [symbol] | { SNAP } { SNAPF } | [{ start-addr-1, end-addr-1 [, ..., start-addr-n, end-addr-n] } (1)] |

Either symbolic addresses or general register 1 can be used to indicate the areas to be dumped. To use symbolic addresses, you code the starting address (*start-addr* parameter) and the ending address (*end-addr* parameter) for each area you want dumped, up to a maximum of 50 separate areas per SNAP macro instruction.

If you use the SNAPF macro instruction, register 0 must be preloaded with the address of either an allocated printer or a virtual printer physical unit block (PUB), as obtained from execution of either a data management OPEN or a read file control block (RDFCB) macro instruction.

Remember, when using symbolic addresses, an even number of parameters must be used (start and end).

For example, if you coded the SNAP macro instruction like this:



and placed it in your program like this:

```

LUC* OBJECT CODE  ADDR1 ADDR2  LINE  SOURCE STATEMENT
000000          1 PROC      START 0
000000 0560      2 BEGIN   BALR 6,0
000002          3          USING *16
000002 47F0 6010      4 BRANCH  " **16
000006 C1C2C3C404040404  5          UC  CL8*ABCD*
000006 C5C6C7C8      6          UC  CL4*EFGH*
000012          7 SNAP   TAG1,TAG2,TAG3,TAG4
000012          8          US  0H
000012 0700          9          CNOP 0,4
000014 4510 6026     10         HAL 1,*(4*4)*4
000018 000002A      11         DC  A(TAG1)
00001C 0000030      12         DC  A(TAG2)
000020 0000040      13         DC  A(TAG3)
000024 60          14         UC  X'80'
000025 000004E      15         DC  AL3(TAG4)
000028 0A10          16         SVC 29 SNAP SVC
00002A 0203 6008 600C 0000A 0000E 17 TAG1  HVC BRANCH*B14),BRANCH*12
00002A 0203 6008 600C 0000A 0000E 18 TAG2  OPEN OUT
000030          19         CNOP 0,4
000030          20 TAG2  EQU  *
000030 4510 6036     21         HAL 1,*(4*2)
000034 80          22         UC  X'80'
000035 000058      23         DC  AL3(OUT)
000038 0A26          24         SVC 38 ISSUE SVC
00003A C207 6102 6004 00104 00006 25         HVC BUF(8),BRANCH*4
000040          26 TAG3  PUT  OUT
000040          27 TAG1  DC  0Y(0) SET ALIGNMENT
000040 5810 6116     28         L  1,=A(OUT) LOAD R15, FILENAME ADDRESS
000044 9220 1031     29         HVI 49(1),X'20' SET FUNCTION CODE
000048 58F0 1034     30         L  15,52(1) LOAD ADDR OF COMMON I/O
00004C 05EF          31         BALR 14,15 LINK TO COMMON
000040          32 TAG4  CLOSE OUT
00004E          33 TAG4  DC  0Y(0)
00004E 5810 6116     34         L  1,=A(OUT) LOAD R15, FILENAME ADDRESS
000052 0A27          35         SVC 39 ISSUE SVC
000054          36         EQU  *
000054          37         US  0H
000054 DATA      38         SVC 26
    
```

First Area to Dump
Second Area to Dump

you would get the following dump when you executed the program (provided you used an OPTION DUMP job control statement):

```

General Registers { SNAP BY SNAPSYMB AT 00040A
REGS 0-7 00000000 800004C8 00000000 00000000 00000000 00000000 00000000
REGS 8-F 00000000 00000000 00000000 00000000 00000000 00000000 00000000

TAG1-TAG2 { SNAP 006CDA TO 006CED
000408 0A100203 6008600C 45 006C08

TAG3-TAG4 { SNAP 006CF0 TO 006CFE
0004F0 5810611A 92201031 58F01034 05EF5A 006CF0
    
```

Notice that the SNAP macro instruction is placed *before* the instruction areas to be dumped.

If you code a large number of addresses to be dumped, the processor time to access the addresses for the dump will increase. But, it takes less time if you access these addresses from a general register. You preload register 1 with the address of a predefined list of one or more address pairs (full word) specifying the areas to be dumped. The leftmost bit of the last ending address must be set to 1 (X'80') to indicate the end of the list to the routine that interprets the SNAP macro instruction.

Borrowing from the example we just used, we'll alter it to set TAG1 through TAG4 in a predefined list, load the symbolic address of this list into general register 1, and instruct the SNAP macro instruction that this register contains the address by coding:

| 1 | LABEL | Δ OPERATION Δ | OPERAND | Δ |
|---|-------|---------------|---------|---|
| | | 10 | 16 | |
| | | SNAP | (1) | |

and inserting it in your program.

```

LOC* OBJECT CODE  ADDR1 ADDR2  LINE  SOURCE STATEMENT
000000          1 PROG      START 0
000000 0560      2 BEGIN    HALR 6,U
000002          3          USING *,6
000002 47F0 6010      4 BRANCH  B      **16
000006 C1C2C3C440404040  00012  5          UC      CL6*ABCD*
00000F C5C6C7C8      6          DC      CL4*EFGH*
000012 4110 6102      00104  7          LA      1,LIST
000016          8          SNAP (1)
000016 0A1D      A 9+      DS      OH
000016 0A1D      A 10+     SVC     29 SNAP SVL
000018 0203 6008 600C 0000A 0000E  11 TAG1   MVC     BRANCH*(4),BRANCH+12
000018 0203 6008 600C 0000A 0000E  12 TAG2   OPEN   OUT
00001E 0700      A 13+     CNOP   0,4
000020          A 14+TAG2 EQU   *
000020 4510 6026      00028  A 15+     HALR  1,*(1*2)
000024 80          A 16+     DC     X'80'
000025 000048      A 17+     UC     AL3(OUT)
000028 0A26      A 18+     SVC     38 ISSUE SVC
00002A C207 60F2 6004 000F4 00006  19          MVC     BUF(8),BRANCH+4
000030          20 TAG3   PUT     OUT
000030          A 21+TAG3 UC     0Y(0) SET ALIGNMENT
000030 5810 6116      00118  A 22+     L      1,=(OUT) LOAD K15, FILENAME ADDRESS
000034 9220 1031      00031  A 23+     MVI   49(1),X'20' SET FUNCTION CODE
000038 58F0 1034      00034  A 24+     L      15,52(1) LOAD ADDR OF COMMON I/O
00003C 05EF      A 25+     HALR  14,15 LINK TO COMMON
00003E          26 TAG4   CLOSE  OUT
00003E          A 27+TAG4 UC     0Y(0)
00003E 5810 6116      00118  A 28+     L      1,=(OUT) LOAD K15, FILENAME ADDRESS
000042 0A27      A 29+     SVC     39 ISSUE SVC
000044          30          EOJ
000044          A 31+     DS      OH
          :
          :
000044          85          DS      F
000048          86 SAVF   DS     CL72
000048          87          DS      F
000048          88 BUF     DS     ZL16
000048          89 LIST   DC     A(TAG1)
000048          90          UC     A(TAG2)
000048          91          UC     A(TAG3)
000048          92          UC     X'80'
000048          93          DC     AL3(TAG4)
000048          94          END   BEGIN
000048          95          =A(OUT)

```

NOTES:

- ① Designates the predefined list of areas to be dumped. The entire list is referenced as LIST. TAG1 through TAG3 are defined as full-word address constants (DC A). The X'80' sets the leftmost bit of TAG4 to 1 (80 = 1000 0000). The remaining three bits of TAG4 are specified by L3.
- ② Loads the address of LIST into general register 1.
- ③ Is the SNAP macro instruction, indicating that general register 1 contains the address of the list of the areas to be dumped.

The dump obtained is identical, in desired content, to the dump that was obtained using symbolic addresses. The execution time for the program using register 1 was reduced. The only differences in the output listings are minor, and do not affect the use of the dump as a debugging aid. They are:

- The program-relative and absolute addresses differ for each method used.

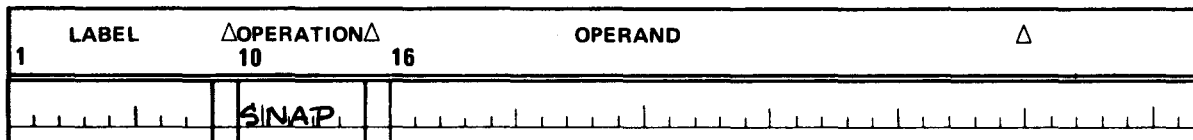
- The listing produced by either method aligns on a double-word boundary. Because different inline expansion codes are generated by the different uses of the SNAP macro instruction, there is a difference in the addresses of the areas to be dumped. So, the listings may be slightly different (as they are in our two examples). You will always get the exact area you want, but you can also receive the generated code of the instruction before or after the area to be dumped, depending on where the double-word alignment begins.

```

General Registers { SNAP HY SNAPREG1 AT 0004CB
                   REGS 0-7 00000000 00000584 00000000 00000000 00000000 00000000 00000484 00000000
                   REGS 8-F 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
TAG1 - TAG2 { SNAP 006CC8 TO 006CDB
              0004CB 02036006 600C0700 45 006CC8
TAG3 - TAG4 { SNAP 006CE0 TO 006CEE
              0004E0 58106116 92201031 58F01034 05EF54 006CE0
    
```

Another benefit of using general register 1, rather than symbolic addresses, is when there is a large string of addresses. If you wanted to remove one of the addresses, and it was not at the end of the string, you would have to change the entire line that contains the address. By using a predefined list, you only have to remove the DC instructions defining the symbolic addresses.

If you code the SNAP macro instruction without any parameters,



only the contents of the 16 general registers are printed.

NOTE:

The contents of general register 1 are destroyed by the SNAP or SNAPF macro instruction. If you want to record the true contents of the register, store it in a field within the area of main storage to be dumped. Also, if you do not specify full-word addresses, the nearest half-word location to the left of the specified address is used.

9.1.2. Normal Termination Dumps (DUMP)

A normal termination dump of main storage differs from a snapshot dump in that it prints out the entire contents of the job region or all main storage, not just selected areas. The DUMP macro instruction causes this, and it is inserted in place of and acts as an EOJ macro instruction (8.3). This means your job step runs to normal completion. Therefore, a DUMP macro instruction terminates a job step without cancelling it, unless, of course, something is wrong with your program.

Just as with the SNAP macro instruction, you can initiate or suppress the dump at run time through the OPTION job control statement. However, with the DUMP macro instruction, there are three types of dump available: SYSDUMP, JOBDUMP, and DUMP. Only one feature is functional per job step, and their hierarchy is in the order just stated. In other words, if both SYSDUMP and JOBDUMP are specified, only SYSDUMP is effective.

The specific meaning of each type of dump is explained in the system service programs user guide, UP-8062 (current version). But briefly, they can be summarized as follows.

The DUMP feature gives you:

- The job's last executed program status word (PSW) and an identification code indicating the source of the dump;
- the job's 16 general registers;
- the job's prologue area with the preamble and task control blocks (TCB); and
- the job's program region.

The SYSDUMP feature provides a method of determining why the system terminated abnormally, which entails:

- a translation of the state of the entire operating system into charts and texts; and
- a hexadecimal dump of all of main storage.

The JOBDUMP feature is basically the same as the DUMP feature, except that the dump listing is also translated from hexadecimal to a more easily readable, English-language version of the dump. Additionally, whenever you want to use the JOBDUMP feature, you must place the following device assignment set in your job control stream:

| | | | | | |
|--------------|----|----|----|----|----|
| 1 | 10 | 20 | 30 | 40 | 50 |
| // DVC 20 | | | | | |
| // LFD PRNTR | | | | | |

If this device assignment set is missing, the dump given is of the module (program) called JOBDUMP, not of your module.

For DUMP and SYSDUMP, a printer must be assigned to the job, but the LFD job control statement does not have to have a file name of PRNTR; the file name is what you have specified on your DTF macro instruction for the job's print file.

If an OPTION job control statement is not present in the control stream, the DUMP macro instruction acts as an EOJ macro instruction (8.3.4). The OPTION job control statement must appear in the job step in which you want the dump to occur.

For example, if you assemble, link edit, and execute your load module, and you want the dump to occur when you execute your load module, you place the OPTION job control statement in the job step that executes your load module, not in the one that assembles or link edits.

The format of the DUMP macro instruction is:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|---|
| [symbol] | DUMP | [{ identification-code } (0) 0] |

The *identification-code* parameter is a 1- to 4-byte hexadecimal code you assign within the program to indicate the source of the dump. If you use all four bytes, it can consist of four alphabetic characters, eight numeric characters, or, since each byte can hold one alphabetic character or two numeric characters, any combination that equals four bytes. Examples of this are:

- 12345678
- A123456
- AB1234
- ABC12
- ABCD

One of the reasons for using an identification code is to uniquely identify the load module producing the dump. This serves as an identifier, which can be used for easy reference when several different dumps are involved.

If we used an identification code of ABCD in the DUMP macro instruction, like this:

| 1 | LABEL | Δ OPERATION Δ | OPERAND | Δ |
|---|-------|---------------|---------|---|
| | | 10 | 16 | |
| | | DUMP | ABCD | |

and used an OPTION JOBDUMP job control statement, the identification code would show up here (note also the program status word):

```

-----
| TASK CONTROL BLOCK # 1 |
-----

```

```

TASK CONTROL BLOCK AT ADDRESS 006500
TASK KEY = 1
NEXT TCB ADDRESS = 006500
BACKWARD LINK ADDRESS = 006500
SUB TASK VERTICAL ID = 000000
SUB TASK COUNT = 0
WAIT FOR TRANSIENT
OUTSTANDING I/O REQUESTS = 0
TASK SWITCH PRIORITY = 20
PREAMBLE ADDRESS = 006400
TRANSIENT ID/SVC CODE = 1B
ECH ADDRESS = 000000
TIMER VALUE = 0:00:00

```

... TASK PSW ...

```

PSW → PROGRAM STATUS WORD = C016001B 700004FA
PROGRAM KEY = 1, WHICH IS JOB DUMP
SYSTEM MODE
CHARACTER MODE IS EHCDC
REGISTER SET IS PROBLEM
PROCESSOR STATE IS PROBLEM
OPERATION MODE IS NATIVE
MONITOR MODE IS OFF
INTERRUPT CODE = 1B
CONDITION CODE = 3
INSTRUCTION ADDRESS = 0004FA
NONZERO INSTRUCTION LENGTH (7 BYTES)
OPERATION: SVC DUMP INSTRUCTION: DABH

```

```

Identification Code → REG 0 0000ABCD
REG 1 0000500
REG 2 0000000
REG 3 0000000
REG 4 0000000
REG 5 0000000
REG 6 40004B2
REG 7 0000000
REG 8 0000000
REG 9 0000000
REG A 0000000
REG B 0000000
REG C 0000000
REG D 0000000
REG E 40004EC
REG F 40004E6

```

here:

```

-----
| T C B |
-----

```

```

PSW →
Identification Code →
000000-10006500 00000200 20000500 00000000 00000000 00006400 18000000 00000000 .....-006500
000020-C016001B 700004FA 0000ABCD 00000500 00000000 00000000 00000000 00000000 .....-006520
000040-400004B2 00000000 00000000 00000000 00000000 00000000 00000000 00000000 .....-006540
000060-400004EC 400004E6 00000000 00000000 00000000 00000000 00000000 00000000 .....-006560
000080-00000000 00000000 00000000 00000000 00000000 00000000 00C1E000 00000100 .....-006580
0000A0-000004CC 00000000 00000000 00000000 00000000 00000000 00000000 00000000 .....-0065A0
0000C0-00000000 00000000 ..... -0065C0

```

and here:

```

-----
| PROBLEM REGISTERS |
-----

```

```

Identification Code → REG 0 0000ABCD
REG 1 0000500
REG 2 0000000
REG 3 0000000
REG 4 0000000
REG 5 0000000
REG 6 40004B2
REG 7 0000000
REG 8 0000000
REG 9 0000000
REG A 0000000
REG B 0000000
REG C 0000000
REG D 0000000
REG E 40004EC
REG F 40004E6

```

```

-----
| JOB USED CORE |
-----

```


You can preload register 0 with the identification code in the same manner as you load the list of symbolic addresses for the SNAP or SNAPF macro instruction. By using register 0, you save on execution time and conserve main storage space.

If you don't specify an identification code, either on the DUMP macro instruction or by preloading it in register 0, an identification code of binary zeros is supplied by the operating system.

NOTE:

A main storage dump and normal termination can also be requested by the console operator entering the DUMP command at the system console. The results are the same as for a DUMP macro instruction included within your program.

9.1.3. Abnormal Termination

A main storage dump can also be obtained by using the CANCEL macro instruction (8.3.5). However, in this case, the issuing program is terminated (and any subsequent programs in the job). This macro instruction terminates the issuing program when error conditions are encountered that prevent further processing.

A main storage dump and abnormal termination can also occur when the operating system performs abnormally. This is known as a system failure dump.

The functions of the CANCEL macro instruction can also be obtained by the console operator entering the CANCEL command at the system console.

9.2. CHECKPOINT AND RESTART CAPABILITY

Hardware and software malfunctions, can cause your job to terminate before its normal completion. Another reason for termination could be that the operator cancelled your job because a high-priority job required all the facilities of the computer. If the job is small, you can rerun it without any really great loss. But, what if it is a long or complex job, where rerunning the job could increase both processing time and cost, thereby reducing productivity? OS/3 has provided the checkpoint facility, which allows you to periodically record the operational status of your job.

The capability to generate checkpoint records is a function of the supervisor, and the capability to use these checkpoint records to restart a job is a function of job control (through the RST job control statement).

You might want to create a checkpoint record at some specific occurrence, such as the end of a magnetic tape reel in a multivolume input file, or after processing a specific number of records. Some people prefer to generate the checkpoint record at fixed time intervals, say, every 15 minutes (by using the SETIME macro instruction to set a timer interrupt).

Macro instructions are available to define, open, and close files for checkpoint records. These macro instructions are used in various combinations, depending on whether your check-point record file and data files are on magnetic tape or disc, and whether you are using data management or the physical input/output control system (PIOCS) to process the data files.

The checkpoint and restart facilities are effective if you are sequentially updating magnetic tape or disc files (using the sequential access method — SAM) where data is not being overwritten. However, checkpoint records are not valid if you are processing disc data files and are updating using either the indexed sequential access method (ISAM) or the direct access method (DAM).

When a checkpoint is taken, a series of records are written to a checkpoint file on either magnetic tape or disc. These records contain the data needed to restart the job, which includes:

- the checkpoint header;
- the job preamble;
- the primary TCB (and any subtask TCBs);
- the remainder of the prologue;
- a list of the files open when the checkpoint was taken; and
- your program.

Each checkpoint is assigned a serial number, which is contained in a checkpoint header record along with the checkpoint file name, job name, and job step number. This information is displayed on the system console and written to the system log. When you want to restart from a checkpoint, you enter this information as parameters on the RST job control statement.

When you restart the job, it is reestablished to a condition functionally identical to the condition at the time the checkpoint was reached. In this way, you do not have to rerun the entire job; just the part that was not completed.

However, if the cause of the failure is in your program, the same error will reoccur.

When you use the restart facility, the job is returned to the status it held when the checkpoint occurred. Tape files are repositioned to the point at which they were, and control is returned to the program at the address specified by the checkpoint.

It is not practical to try to reposition data cards in the card reader when restarting from a checkpoint. However, if you want to use the checkpoint facility with card files, you can enter the cards as embedded data in the job control stream and use the GETCS macro instruction to access the data.

NOTE:

The LFD job control statement in the device assignment set for the checkpoint file must not contain the INIT parameter. This parameter causes the file to be written from the beginning of the file. In other words, the checkpoint records already existing on the file will be overwritten.

In order to restart a job, you must reenter the original control stream with an RST job control statement, which must appear as the first job control statement of your job control stream. Of course, all the files needed to complete the job must be available, along with the file that contains the checkpoint records. For information on how to use the RST job control statement, see the job control user guide, UP-8065 (current version).

9.2.1. How to Generate Checkpoint Records (CHKPT)

A series of checkpoint records are generated each time a CHKPT macro instruction is executed in a program. These records must be written to either a magnetic tape file (defined by a DTFMT macro instruction), or a disk file (defined by a DDCPF macro instruction). The use of these macro instructions and those needed to open and close the file are discussed later in this section (along with how the CHKPT macro instruction is used in connection with these other macro instructions). The format of the parameters of the CHKPT macro instruction is:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|---|
| [symbol] | CHKPT | filename [,restart-addr] [,list-name] [,error-addr] |

All the parameters of this macro instruction are positional parameters.

The *filename* parameter specifies the symbolic address of the macro instruction that defines the checkpoint record file. This macro instruction is either a DTFMT macro instruction for a magnetic tape, or a DDCPF macro instructions for a disk. The value specified for this *filename* parameter is also the value you use for the *filename* parameter of the LFD job control statement in the device assignment set that defines the checkpoint file in the job control stream.

The *restart-addr* parameter is used to supply the symbolic address of an instruction in your program that is to receive control when restarting the program from the series of records taken by the execution of *this* CHKPT macro instruction.

You can have more than one CHKPT macro instruction in a program. Assume, for instance, that your job could be broken down realistically into two separate processing functions: first, it has to read 10 tape volumes as input; and second, it then updates a master disk file with the data that was on these 10 tape volumes. You could open the checkpoint file, take checkpoint records, and then close the checkpoint file in the first step (tape-in), and then you could open, take checkpoint records, and close another checkpoint file when updating. In this way, you do not go through the series of code for tape-in if the restart is to affect only the update portion of the program. You also will not need tape drives that can be used by other jobs (thus you can omit the device assignment set for the tape-in).

If you omit the *restart-addr* parameter, the instruction immediately following the CHKPT macro instruction receives control.

The next parameter, *list-name*, is used only when working with PIOCS files. It specifies the symbolic address of the DCFLT macro instruction that generates a list of files in your program that are accessed via PIOCS.

If an error occurs during the execution of the CHKPT macro instruction, the job, by default, is terminated abnormally. However, you can place an error routine in your program to override this abnormal termination and continue processing without the checkpoint. The *error-addr* parameter is used to specify the symbolic address of this error routine. In this way, if an error does occur, the error routine receives control; no abnormal termination occurs. After the execution of a CHKPT macro, the checkpoint routine checks register 0, which contains the checkpoint status, and which is, in effect, an error code. If the error code is equal to 0, it means the checkpoint completed successfully, and processing of the job continues. If the code is other than 0, the error routine (or abnormal termination, if an *error-addr* parameter is not used) receives control. The possible checkpoint error conditions and error codes that may occur are listed in Table 9-1. Also listed are the possible restart error conditions and codes that may occur when trying to restart the job.

Table 9-1. Checkpoint/Restart Error Codes

| Error Code (in Hexadecimal) | Description |
|--------------------------------|--|
| Checkpoint Error Codes | |
| A0 | Checkpoint file is not opened. |
| A1 | Unrecoverable I/O error while writing a checkpoint record |
| A2 | Checkpoint record cannot fit in checkpoint file. |
| | NOTE: If the checkpoint record cannot fit, an attempt is made to write it at the start of the checkpoint file. If it still does not fit, this error code is returned. |
| A3 | Illegal parameter specified on checkpoint macro |
| Restart Error Codes | |
| A4 | Unrecoverable I/O error while reading checkpoint file |
| A5 | At restart, processor could not locate designated checkpoint. |
| A6 | At restart, processor could not position data tape files; unrecoverable I/O error. |
| A7 | At restart, processor determined that supervisor was not compatible with the supervisor at the time of the checkpoint. |
| A8 | At restart, processor determined that hardware incompatibilities existed between the system at checkpoint time and the system at restart time. |

Once again, if you do not provide a checkpoint error routine in your program and do not supply its symbolic address with the *error-addr* parameter of the CHKPT macro instruction, the job terminates abnormally (if an error occurs), and the following message is displayed at the system console:

JC03 JOB jobname TERMINATED ABNORMALLY.ERR CODE number

where *number* corresponds to the error code listed in Table 9—1. (These error codes are also listed in the system messages programmer/operator reference manual, UP-8076 (current version).

9.2.2. Using Magnetic Tape as the Checkpoint File

If a magnetic tape is to receive the checkpoint records, you have to use the DTFMT data management macro instruction to define the file. Two requirements of this macro instruction when defining a checkpoint file are:

- It must indicate that standard labels are being used (FILABL=STD keyword parameter).
- The block size (BLKSIZE keyword parameter) must be at least 80 bytes to meet the requirements of data management. If you omit the BLKSIZE parameter, data management assumes 256 bytes.

You can also intersperse data records with the checkpoint records on this file.

The DTFMT macro instruction does not generate executable code, so you must locate it in your program separate from your BAL instructions and imperative macro instructions.

Before the checkpoint file can be accessed (first execution of the CHKPT macro instruction for the file), you must open the file using the OPEN macro instruction. After the last execution of the CHKPT macro instruction, you must close the file using the CLOSE macro instruction. These macro instructions are fully explained in the data management user guide, UP-8068 (current version). It is advisable to become familiar with them before you attempt to structure a magnetic tape checkpoint file.

Here is a skeletal example, which shows basically what parameters the CHKPT macro instruction agrees with in regard to the other instructions in your program when using magnetic tape.

| 1 LABEL | △OPERATION△ | 10 | OPERAND | △ |
|---------|--|----|-------------------------|---|
| | OPEN | | CKTAPE | |
| | . | | | |
| | . | | | |
| | . | | | |
| | CHKPT | | CKTAPE, TAGA, ROUT1 | |
| | . | | | |
| | . | | | |
| ROUT1 | (your error recovery routine) | | | |
| | . | | | |
| | . | | | |
| TAGA | (instruction where program is to begin if restarted) | | | |
| | . | | | |
| | . | | | |
| | CLOSE | | CKTAPE | |
| | . | | | |
| | . | | | |
| | . | | | |
| | EOJ | | | |
| | . | | | |
| | . | | | |
| CKTAPE | DFMT | | FILABLE=STD, BLKSIZE=80 | |

9.2.3. Using a SAT Disk as a Checkpoint File

In addition to using magnetic tape to receive checkpoint records, you can use disk. As many checkpoint records as will fit are recorded in the disk space you allocate for the file (with an EXT job control statement). When the space is exhausted, a wraparound, in effect, takes place: the checkpoint records are written at the beginning of the file, over the existing records, thus losing those checkpoint records taken earlier. For this reason, you cannot intersperse any of your data with checkpoint records on disk, since you could lose data if wraparound occurs.

9.2.3.1. Estimate Space Requirements for a Disk Checkpoint File

Each checkpoint consists of a series of 256-byte records of the following type:

| <u>Data</u> | <u>Checkpoint Records</u> | |
|-----------------------|-------------------------------|---|
| Checkpoint header | 1 | |
| Prologue | | |
| Preamble | 1 | |
| TCB | n | (1 TCB record per task) |
| Remainder of prologue | n | $(n = \frac{\text{remaining size}}{256})$ |
| File list | 1 | |
| User program | n | $(n = \frac{\text{user program}}{256})$ |

Using this list, you can estimate the minimum disk space requirements. The total amount of space required depends on the size of your program. For example, assume your program consisting of one task occupies 8192 bytes of main storage plus a prologue of 1024 bytes. The checkpoint records would consist of the following:

| <u>Data</u> | <u>Checkpoint Records</u> | <u>Bytes</u> |
|-----------------------|-------------------------------|--------------|
| Checkpoint header | 1 | 256 |
| Prologue | | |
| Preamble | (1 record — 256 bytes) | |
| TCB | (1 record — 256 bytes) | |
| Remainder of prologue | (2 records — 512 bytes) | |
| Total prologue | 4 | 1025 |
| File list | 1 | 256 |
| User program | <u>32</u> | <u>8192</u> |
| Totals | 38 | 9728 |

Thus, a checkpoint for this program would consist of 38 records of 256 bytes each, or a total of 9728 bytes. If you were using an 8416 or an 8418 disk subsystem, one entire checkpoint would fit on a single track (each track can hold 40 records).

If you allocate only one track, each checkpoint taken would overwrite the preceding checkpoint. To avoid doing this, you must allocate at least twice the minimum space required, which in this case is approximately two tracks. The current checkpoint would then overwrite the records recorded from an earlier checkpoint, while the most recent checkpoint would always be available.

9.2.3.2. Define, Open, and Close a Disk Checkpoint File (DDCPF, DCPOP, DCPCLS)

When employing a disk checkpoint file, a different group of macro instructions are used to define, open, and close the file. We will now explain each.

In order to define a disk checkpoint file, use the DDCPF macro instruction (versus the DTFMT macro instruction used for tape). As you can see in the format, there are no parameters associated with this macro instruction.

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|---------|
| filename | DDCPF | |

There is only a *filename* in the label field. Just as with the DTFMT macro instruction, this *filename* is the symbolic address and is used as positional parameter 1 (*filename*) of both the CHKPT macro instruction and the LFD job control statement.

Since the DDCPF macro instruction does not generate executable code, it must be placed separate from your BAL instructions and imperative macro instructions.

Use the DCPOP macro instruction to open the disk checkpoint file (before the execution of the CHKPT macro instruction). It has this format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|-----------------------|
| [symbol] | DCPOP | { filename } (1) |

The *filename* parameter specifies the symbolic address of the DDCPF macro instruction that defines the checkpoint file. You can also preload this address in general register 1, and you indicate this by coding (1) in place of the *filename* parameter.

To close a disk checkpoint file, use the DCPCLS macro instruction (after the last time the CHKPT macro instruction is executed). The format is:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|-----------------------|
| [symbol] | DCPCLS | { filename } (1) |

The two parameter choices, *filename* or (1), have the same meaning as the parameters of the DCPOP macro instruction: *filename* is the symbolic address of the DDCPF macro instruction, and (1) indicates that the address is stored in general register 1.

Here is an example, showing the relationship between the parameters of the CHKPT macro instruction and the new macro instructions we just discussed.

| 1 LABEL | Δ OPERATION Δ | OPERAND | Δ |
|---------|--|---------------------|---|
| | 10 | 16 | |
| | DCP,OPN | CKDISK | |
| | . | | |
| | . | | |
| | . | | |
| | CHKPT | CKDISK,RESTART,ERRI | |
| | . | | |
| | . | | |
| | . | | |
| ERRI | (your error recovery routine) | | |
| | . | | |
| | . | | |
| | . | | |
| RESTART | (instruction where program is to begin if restarted) | | |
| | . | | |
| | . | | |
| | . | | |
| | DCPCLS | CKDISK | |
| | . | | |
| | . | | |
| | . | | |
| | EOT | | |
| | . | | |
| | . | | |
| | . | | |
| CKDISK | DDCPE | | |

9.2.4. Processing PIOCS Files (DCFLT)

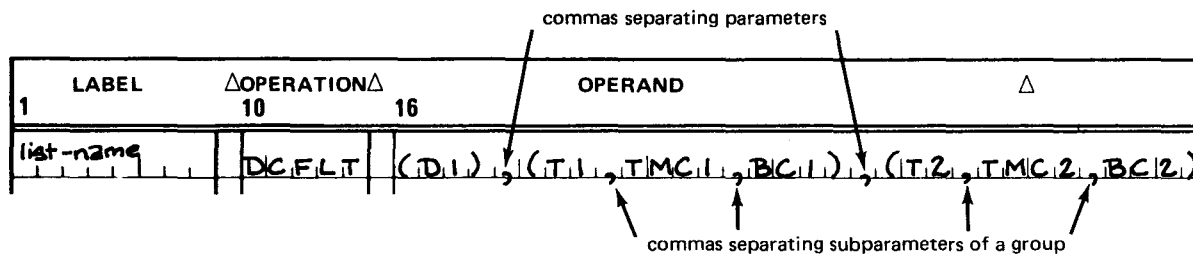
The checkpoint routine uses a file table list to identify the files open when the checkpoint occurs (thus indicating in the checkpoint record which files are needed to restart from this checkpoint). If you use data management files, the list is automatically generated and maintained by data management in the job prologue. However, if you are using PIOCS, you have to generate the file table list by using the DCFLT macro instruction in your program. The generated list locates the file definitions known as physical input/output control *blocks* (PIOCB) for all the files accessed via PIOCS. This list is required for the repositioning and other file-related activities when restarting the checkpointed job. (Remember, data management does this automatically, but PIOCS does not so you have to use this macro instruction only with PIOCS files.)

The DCFLT macro instruction is declarative, just like the DTFMT and DDCPF macro instructions; it does not generate executable code. So, it must be placed separate from your BAL instructions and imperative macro instructions.

The format of the DCFLT macro instruction is:

| LABEL | △OPERATION△ | OPERAND |
|-----------|-------------|---|
| list-name | DCFLT | <pre>{ (disk-PIOCB-1) (tape-PIOCB-1,tmc-1,bc-1) } [{ ,(...),(disk-PIOCB-n) { ,(...,...,...),(tape-PIOCB-n,tmc-n,bc-n) }]</pre> |

Before explaining the parameters, we will mention the coding conventions you have to follow when using this macro instruction. For disc PIOCS files, each parameter must be enclosed by parentheses, such as: (disk-PIOCB-1). For tape files, each parameter consists of a group of three subparameters. Each group of subparameters must be enclosed by parentheses, and each subparameter within the group must be separated by a comma, such as: (tape-PIOCB-1, tmc-1, bc-1). If more than one PIOCS file is used, each parentheses-enclosed parameter is separated from the next parentheses-enclosed parameter by a comma. For example, if you used one disk file and two tape files, assigned the values D1 to the disk file, and assigned T1, TMC1, BC1 to the parameter group for the first tape file and T2, TMC2, BC2 to the parameter group for the second tape file, it would be coded as:



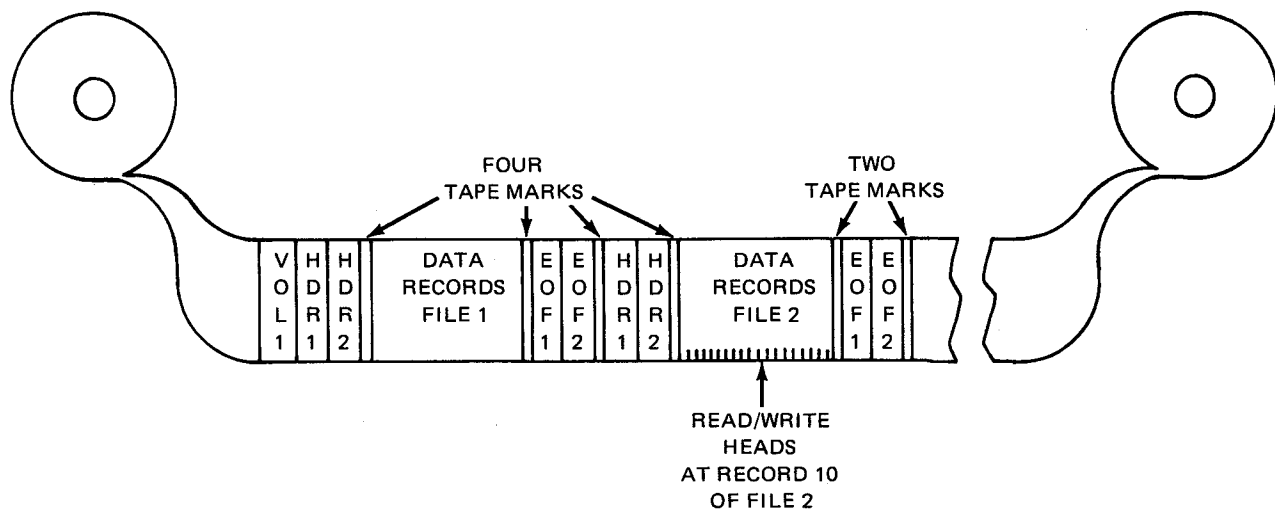
Notice every parameter is enclosed by parentheses.

The *list-name* in the *label* field is the symbolic address of the PIOCS file list table. This is also used as the *list-name* parameter (positional parameter 3) of the CHKPT macro instruction.

The *disk-PIOCB* parameter specifies the symbolic address of a PIOCB for a disk file in your program. You can have from 1 to *n* number of entries of the *disk-PIOCB* parameter, with *n* depending on how many disk files are used in the program.

The *tape-PIOCB* subparameter does for tape files what the *disk-PIOCB* parameter does for disk files. It specifies the symbolic address of a PIOCB for a tape file in your program. And just as for disk, you can have from 1 to *n* number of entries of the *tape-PIOCB* subparameter (and its associated subparameters), with *n* again depending on the number of tape files being used.

For every PIOCS tape file, there are also two other subparameters needed to complete the parameter group: *tmc* and *bc*. The *tmc* subparameter specifies the symbolic address of a half word in the program where you keep a binary count of the tape marks read between the tape load point and the current position of the tape (used to reposition the tape to the correct file when restarting). The *bc* subparameter specifies the symbolic address of a full word in the program where you keep a binary count of the blocks (physical data records) read from the most recent tape mark to the current tape position (used to position the tape to the correct data record of the file when restarting). For instance, suppose you were already processing the second file on a tape (which would be past the fourth tape mark) and were at the tenth data record. This could be shown as:

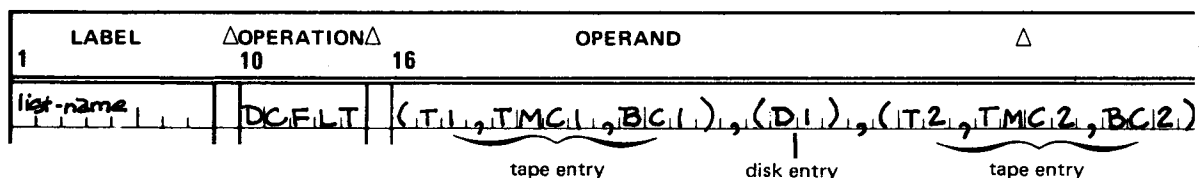


LEGEND:

| | |
|------|---------------------|
| VOL1 | Volume label 1 |
| HDR1 | File header label 1 |
| HDR2 | File header label 2 |
| EOF1 | End of file label 1 |
| EOF2 | End of file label 2 |

If a machine error occurred at this time, you would want to reposition the tape to this point when restarting the job. You would not want to read all the data from tape mark 1 to tape mark 2 and from tape mark 2 to data block 10. By looking at the storage areas in your program referenced by the *tmc* and *bc* parameters, the checkpoint routine knows where to reposition the tapes.

Entries for disk and magnetic tape files can be interspersed, such as:



Here is an example (with the macro instructions needed to open, close, and define the checkpoint file), showing the relationship of the parameters of the DCFLT macro instruction to the CHKPT macro instruction and the macro instructions used by PIOCS. Routines accessed by the *restart-addr* and *error-addr* parameters and also the end of job (EOJ) instruction are not shown.

| 1 | LABEL | △OPERATION△ 10 | 16 | OPERAND | △ |
|----|----------|-------------------|----|--|---|
| | | DCPOP | N | CKDISK | |
| | | . | | | |
| | | . | | | |
| | | . | | | |
| | | CHKPT | | CKDISK, FILELIST | |
| | | . | | | |
| | | . | | | |
| | | . | | | |
| | | DCPCLS | | CKDISK | |
| | | . | | | |
| | | . | | | |
| | | . | | | |
| | CKDISK | DDCFE | | | |
| 1. | FILELIST | DCFLT | | (PIOD1), (TIOT, TMCNT, BNCNT), (PIOD2) | |
| 2. | TIOT | PIOCB | | | |
| 3. | PIOD1 | PIOCB | | | |
| 4. | PIOD2 | PIOCB | | | |
| 5. | TMCNT | DS | | BL2 | |
| 6. | BNCNT | DS | | BL4 | |

1. Generates a file list table for one tape file (TIOT) and two disk files (PIOD1 and PIOD2).
2. Constructs a PIOC B for the tape file.
3. Constructs a PIOC B for one of the disk files.
4. Constructs a PIOC B for the other disk file.
5. Defines a storage area of one half word to keep a count of the tape marks read.
6. Defines a storage area of one full word to keep a count of the data records read since the last tape mark was encountered.

9.3. MONITOR AND TRACE CAPABILITY

Another means of debugging a program is the monitor routine. It enables you to track (or trace) the execution of a program (by using a hardware interrupt) so that errors can be found and fixed. As input, you provide monitor statements that indicate the type of diagnostic action to be performed at a specific point in the program.

The monitor routine interrupts each instruction before it is executed, and tests whether any of the following test conditions are stated in the monitor statement input and have been met by the instruction.

- A specified storage location is referenced (or the data stored at that location is changed).
- A specified location in the program is reached.
- A specific sequence of instructions occurs.
- A specified register is changed.

If any of these conditions are met, you get a printout of various types of program information, depending on which display option you chose. This is summarized in Table 9-1.

Then, you can:

- continue executing the program under monitor control;
- suspend program execution; or
- continue the normal execution of the program without intervention from the monitor routine.

Depending on how you call the monitor routine into main storage and the choice of actions you select, an entire task or only part of a task can be monitored.

To activate the monitor routine, you must ensure that the following provisions are met:

- The monitor routine must be in main storage.
- The monitor bit in the PSW must be set.
- The task to be monitored, location options, and actions must be specified to the monitor routine.
- A printer must be available.

9.3.1. How to Call the Monitor Routine

There are two ways to call the monitor routine into main storage. Which one you use depends on whether you want to trace instructions from the beginning of the job or wait until after the job begins executing.

Whenever you use the monitor routine, keep this in mind: it occupies 3K bytes of main storage. If you specify the minimum main storage as a parameter of the JOB control statement, make sure you do not overestimate the storage size needed by your job, because it is possible that there might not be enough main storage available for the monitor when you combine your job needs plus the 3K bytes needed by the monitor.

Another point to remember: the monitor routine cannot be run in a strict spooling environment, because the job being monitored always requires the sole use of a printer. You can accomplish this through the *addr* parameter of the DVC job control statement which, in effect, dedicates a printer strictly to this job. It's coded immediately following the logical unit number (separated by a comma). Every device has a hardware address number associated with it. Your site manager can provide you with the number you need. (In most cases, however, this number can be physically found on the device itself, generally on some type of label.) This number is then coded in the device assignment set for the print file in your job.

Assume the printer you want to dedicate has a hardware address number of 170. Using 20 as the logical unit number, the DVC job control statement would be:

| | | | | | |
|---------------|----|----|----|----|----|
| 1 | 10 | 20 | 30 | 40 | 50 |
| // DVC 20,170 | | | | | |

It is also recommended that the job be run as the first job immediately after the system is initialized (initial program load) to ensure that the job is scheduled; otherwise, you might have to wait for the job to be scheduled, depending on the work load.

9.3.1.1. Monitoring From the Beginning of the Job

If you want to begin monitoring with the first instruction executed, you must call the monitor routine into main storage before the job to be monitored is run. In this case, the monitor input is entered as embedded data in the control stream.

The system operator types MO at the system console, which brings the monitor routine into main storage. The monitor initializes itself and awaits activation.

NOTE:

This console procedure is a temporary expedient and will be replaced in a later release by appropriate job control statements.

If you want to use the monitor beginning with the first instruction of the program, you must enter the monitor statements as embedded data in the job control stream. The job step that contains the program to be monitored must include an OPTION job control statement with the TRACE parameter specified. This parameter activates the monitor routine by setting the monitor bit in the PSW and creates a link between this job step and the monitor statements. If the OPTION job control statement is not present in the proper job step (the one with the monitor statements— — the one you want to trace), it will not activate the monitor routine because an OPTION job control statement is effective only in the job step in which it is encountered. As soon as the program begins executing, monitoring begins, and it continues until the program completes or until the monitor is deactivated by meeting the conditions that accompany a Q action (9.3.5.3).

The control stream you submit when you want to monitor from the beginning of the job would look something like this:

| | 1 | 10 | 20 | 30 | 40 | 50 |
|----|-----|-------------------------|-------------------------|----|----|----|
| 1. | // | JOB | jobname | | | |
| 2. | | . | | | | |
| | | . | | | | |
| | | device assignment sets | | | | |
| | | and any other necessary | | | | |
| | | job control statements | | | | |
| | | . | | | | |
| | | . | | | | |
| | | . | | | | |
| 3. | // | OPTION TRACE | | | | |
| 4. | // | EXEC | program-name | | | |
| 5. | /\$ | | monitor input-explained | | | |
| | | | in 9.3.2 | | | |
| | /* | | | | | |
| 6. | // | PARAM | operands | | | |
| 7. | /\$ | | any data cards | | | |
| | | | needed by the program | | | |
| | /* | | | | | |
| 8. | /& | | | | | |
| | // | FIN | | | | |

1. Is the JOB control statement, which must be present at the beginning of every job.
2. Represents the device assignment sets and any other job control statements you might need to define the requirements for the job.
3. Is the OPTION job control statement indicating you want to monitor the job step. It is placed before the EXEC job control statement for the job step.
4. Calls the program from a library and initiates its execution.
5. Shows where you place the monitor statements. They are enclosed by the /\$ and /* job control statements (start of data and end of data). The monitor statements, in effect, are data for this job, but their presence does not affect processing of any other data for this job.
6. Indicates where you would place any PARAM job control statements that pertain to this job step: after the monitor statements, but before any other card data files.
7. Is the start of data, card data file, and end of data.
8. Ends the job and terminates the card reader operations. Of course, there could be more job steps than this, but for the sake of brevity, we have shown only a single-job-step job.

9.3.1.2. Monitoring After Execution Begins

It should be noted that, when the monitor is in use, it executes several instructions of its own for every monitored instruction in the program. For a large program, this could require excessive amounts of processor time, especially if the problem area is at the end of the program. (If it is at the beginning of the program, a Q action can be used to deactivate the monitor after the necessary data is obtained. The Q action is described in 9.3.5.3.) However, once you determine the particular area in which the problem exists, you can limit the monitor activities to this portion of the program. You do this by initiating the monitor routine via a console type-in *after* the job begins execution, and then entering the monitor statements through the card reader (or the system console if a card reader is not available). This requires some form of communication between you and the console operator, either oral or written.

The executing program must be temporarily suspended so the monitor can be activated before the area of code to be monitored is passed. There are three different methods for doing this.

First, if you have an instruction in the program that you do not need for this execution, you can use the ALTER job control statement to change that instruction to a supervisor call (SVC) for the YIELD routine. This changed instruction must be at a point in the program before the area to be monitored. The ALTER job control statement would look something like this:

```
1          10          20          30          40          50
// ALTER phase-name, address, X'0A04'
```

The ALTER job control statement and its parameter are explained in the job control user guide, UP-8065 (current version). The X'0A04' (positional parameter 3) is what replaces the existing instruction and makes it an SVC instruction for the YIELD routine. It causes the program to halt at the address on the ALTER job control statement. You tell the operator to have the monitor statements ready in the card reader. When the program halts, the operator types in 00 MO R, which activates the monitor routine. (This acts just like the TRACE parameter of the OPTION job control statement.) The monitor statements are then read into the system, and the program named on the monitor statement is matched against all the programs currently executing, until it arrives at the proper program (this applies to all three methods). In 9.3.2, we explain the format for the monitor statement input, which applies to input entered after execution begins or as embedded data in the control stream. However, it should be noted that when monitor statements are entered after execution begins, no /\$ or /* job control statements are needed to enclose the monitor input. Since all the job control statements are read before execution, an OPTION TRACE job control statement is not included in that control stream to activate the monitor. If you examine the control stream shown in 9.3.1.1 used to activate the monitor from the beginning of the job, you will see the difference between it and the following control stream used only to start the job and alter an instruction in your program. (It does not activate the monitor; a console type-in does.)

```
// JOB jobname
// ALTER phase-name, address, X'0A04'
.
.
device assignment sets and any other necessary job control statements
.
.
// EXEC program-name
// PARAM operands
/$ any data cards needed by the program
/*
/&
// FIN
```

The explanation for each job control statement is the same as for the corresponding job control statement in 9.3.1.1. Notice the absence of an OPTION job control statement and the monitor statements, and the presence of the ALTER job control statement.

After the monitor statements have been read, the operator must issue the GO command, using the same job name as that on the JOB control statement. This resumes program execution under monitor control.

The second method for suspending the executing program is the use of an OPR macro instruction with a REPLY parameter (10.3.2). By placing it in a location near the area you want to monitor, you can use the halt when the program is suspended and the message it generates to instruct the operator to activate the monitor. Once again, the operator must have the monitor statements ready in the card reader (no /\$ or /*). He then enters OO MO R, to activate the monitor. After the monitor statements have been read, he enters the reply you requested with the OPR macro instruction to resume processing under monitor control. The monitor input is exactly the same as when using the first method. That is, no /\$ or /* enclose it, and an OPTION TRACE job control statement is not submitted in the control stream. (And, in this case, no ALTER job control statement is submitted.)

The third method is to instruct the operator to type in the PAUSE command at some specific place in the program execution. This could be after a certain time limit has expired, or when a certain milestone is reached, such as the end of an input tape file. The operator places the monitor statements in the card reader and, when the system halts, types OO MO R to activate the monitor routine. After the monitor statements are read, he finally types GO and the job name from the JOB control statement to resume program execution under monitor control.

There might be a situation when there is no card reader available to read in the monitor statement (or no keypunch readily available to prepare the monitor statements). If this is the case, the operator can type in OO MOC at the system console. The C indicates to the system that the monitor statements are going to be input via the console, not via a card reader. (This applies to entering the monitor statements during all three methods of suspending program execution.) In this way the operator can enter the task, options, and actions at the console. He enters one card at a time, a line on the screen corresponding to a card in the monitor statement input, and indicates the end of each card by pressing the TRANSMIT key. After all monitor statements are sent, he enters the GO command followed by the job name.

9.3.2. Monitor Input Format

The monitor statements define what to monitor (task), when to monitor (option), and what to do when you monitor (action). This applies to monitor statements submitted via the control stream as embedded data before the job begins, and to the monitor statements used by the operator after program execution was begun. (Remember, the /\$ and /* job control statements are only needed when the monitor statements are submitted as embedded data.)

For the program you want to monitor, only one task can be specified. It must be coded as the first monitor statement of the input, and no options or actions can share this card with the task. These tasks are explained in 9.3.3. For the task, however, you can specify up to 15 different options. (Each option must be on its own card; no two options can be present on the same card.) Each option can specify as many actions as will fit on a single card. A space must be used to separate the option from the first action on the card, and each succeeding action is separated from the previous action by a semicolon (;).

So, if you want to specify one option and one action, it would be coded as:

| 1 | 10 | 20 | 30 | 40 | 50 |
|---------------|----|----|----|----|----|
| option action | | | | | |

If you wanted three different options, each with two actions, it would be coded as:

| | | | | | |
|----------------------------|--|--|--|--|--|
| option-1 action-1;action-2 | | | | | |
| option-2 action-1;action-2 | | | | | |
| option-3 action-1;action-2 | | | | | |

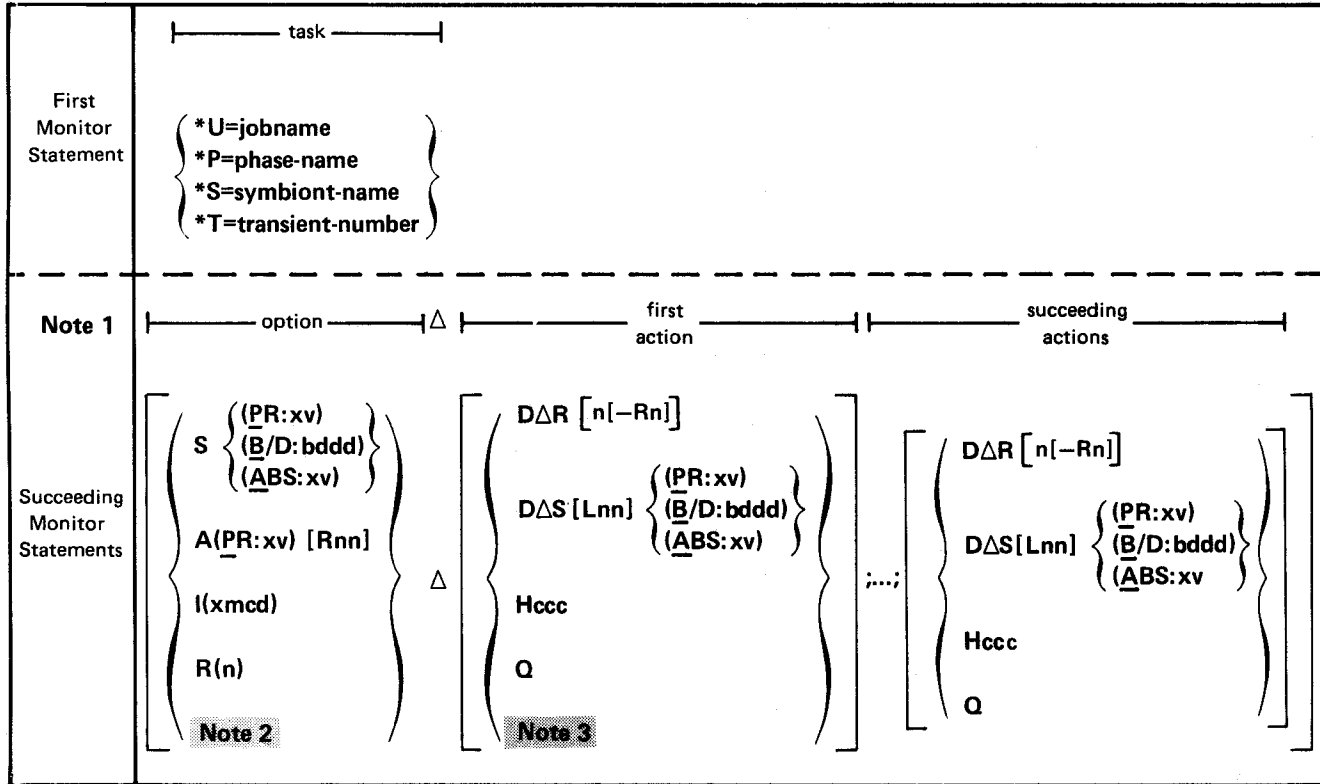
The last card used in the monitor input stream is a \$ card. (Do not confuse this with the /\$ job control statement, which indicates start of data.)

So, the order of a monitor input stream is:

- the task statement;
- the first option statement with its actions;
- any other option statements and their actions; and
- the \$ card.

The options are described in 9.3.4, and the actions are defined in 9.3.5.

Figure 9—1 shows the format of the monitor statements.



NOTES:

1. If no option is specified, the monitor routine assumes a default option (9.3.4.5) and default display (9.3.5.1.3).
2. If no action is specified, the monitor routine produces a default display (9.3.5.1.3). Also, remember that the first action is separated from the option by a blank space, and any succeeding actions are separated from the previous action by a semicolon.

Figure 9-1. Monitor Input Format

9.3.3. Defining What You Want to Monitor

The task you want to monitor can be one of four types:

1. Your entire program
2. A certain phase of your program
3. A symbiont, which is a system utility routine
4. A transient, which is an OS/3 routine that is nonresident and is called into a transient area when needed.

In this format:

```

{
  *U=jobname
  *P=phase-name
  *S=symbiont-name
  *T=transient-number
}
  
```

you can see that each type has its own specification, and each type is preceded by an asterisk.

If you want to monitor all the phases of your program, use the **U=jobname* entry. The jobname is the same as the *jobname* parameter on the JOB control statement. (Remember, if you have the operator enter the monitor statements after the program has started, you can limit monitoring to a part of the job step; otherwise, the job step is monitored from the beginning.)

For example, if the JOB control statement is:

```

1          10          20          30          40          50
// JOB POCO
  
```

the monitor task statement would be:

```

*U=POCO
  
```

Since a program can consist of more than one phase, it can be useful to use the specific phase name with the **P=phase-name* entry. (A program can also have more than one phase.) If you want to monitor a phase, you have to know its name. The names of the phases used in a program are listed on the allocation map provided by the linkage editor. (Remember, operator input can limit the monitor to a portion of a phase.)

If the phase name you want is this:

** ALLOCATION MAP **

| PHASE NAME | TRANS ADDM | FLAG | LABEL | TYPE | LSID | LNK ORG | HIADDR | LENGTH | OBJ ORG |
|---------------------------------------|-------------|------|---------|-------|------|----------|----------|----------|----------|
| LNKLODDU | NODE - ROOT | | | | | 00000000 | 000005CR | 000005CC | |
| *** START OF AUTO-INCLUDED ELEMENTS - | | | | | | | | | |
| - 75/10/04 05:59 - | | | | | | | | | |
| | | | PHX10E | OBJ | | | | | |
| | | | PHX10E | CSECT | 01 | 00000000 | 000004AF | 000004BD | 00000000 |
| | | | DPXCOM7 | ENTRY | 01 | 00000000 | | | 00000000 |
| | | | DPXCOM0 | ENTRY | 01 | 00000000 | | | 00000000 |
| | | | DPXCOM1 | ENTRY | 01 | 00000000 | | | 00000000 |
| | | | DPXCOM6 | ENTRY | 01 | 00000000 | | | 00000000 |
| | | | DPXCOM2 | ENTRY | 01 | 00000000 | | | 00000000 |
| | | | DPXCOM5 | ENTRY | 01 | 00000000 | | | 00000000 |
| | | | DPXCOM4 | ENTRY | 01 | 00000000 | | | 00000000 |
| | | | DPXCOM3 | ENTRY | 01 | 00000000 | | | 00000000 |

the monitor task statement is:

| | | | | | |
|---|----|----|----|----|----|
| 1 | 10 | 20 | 30 | 40 | 50 |
|---|----|----|----|----|----|

```
*P=L N K L O D O O
```

To monitor a symbiont, you have to obtain its name from the system load library file (SYSLOD), and use the *S=*symbiont-name* entry. For example, the name of the system utility symbiont (SU) is SL\$\$SU. To monitor it, you would code

```
*S=SL$$SU O O
```

as the monitor task statement.

Every transient has a decimal number associated with it. A list of these decimal numbers is maintained by the supervisor. If you want to monitor a supervisor transient your program is using, your Sperry Univac systems analyst can help you in determining the number of the transients you need. Once you have obtained it, you use it in the *T=*transient-number* entry. If, for example, the transient number is 24, you would code the monitor task statement as:

```
*T=24
```

9.3.4. Specifying Options

The second and succeeding cards used for monitor statements specify options and actions; that is, points in the program where one or more actions are to be taken by the monitor routine, and what is to occur.

The first entry in each of these cards specifies the option. This may be followed by one or more actions to be performed at the specified location (or else a default action applies). These actions are described in 9.3.5. In this discussion, all the options are discussed first, then the actions. You can tie the appropriate options and actions to a task to obtain your desired result.

If there are duplicate or overlapping options, only the first one specified is processed at execution time. For example, if the same instruction location is specified by two separate cards, the monitor routine performs the actions requested on the first card for that location, then executes the instruction. The second card is never considered for that location, even if the actions are totally different.

Options may be specified in any sequence; there is no need to list them according to any pattern. Remember, in the case of duplicate or overlapping options, only the first option is processed.

There are four types of option you can specify, using the following format:

$$\left(\begin{array}{l} S \left\{ \begin{array}{l} (\underline{PR}:xv) \\ (\underline{B/D}:bddd) \\ (\underline{ABS}:xv) \end{array} \right\} \\ A(\underline{PR}:xv) [Rnn] \\ I(xmcd) \\ R(n) \end{array} \right)$$

The *S* option is used for storage reference, the *A* option for instruction location, the *I* option for instruction sequence, and the *R* option for register change. Each, along with its associated parameters, is discussed in the following paragraphs.

9.3.4.1. Storage Reference Option (S)

This option requests the monitor routine to take action when the specified storage location is referenced or the data at that location is changed. There are three ways to express the location in a storage reference option:

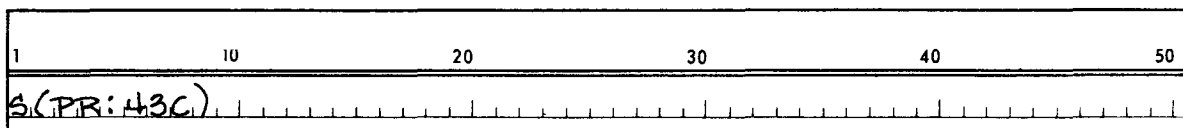
1. Program relative (PR)
2. Base/displacement (B/D)
3. Absolute (ABS)

9.3.4.1.1. Program Relative Address (PR)

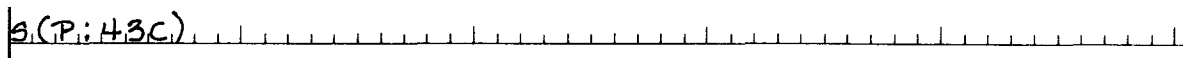
The format for the storage reference option using a program relative address is:

S(PR:xv)

The *xv* is the address, and can consist of from one to six hexadecimal characters, in the range 0_{16} to $FFFFFF_{16}$. Notice that it is separated from the PR by a colon. For example:



Since this format is shown with an underline under the P, you could also code it as:



This is explained in the statement conventions.

In this example, this option is selected if program execution reaches an instruction that references storage at program relative address 43C. The location specified need not be the first byte of a field. For example, a move instruction from location 42E for 18 bytes would be detected because the specified program relative address 43C falls within the field moved (42E to 43F).

For your program, a phase of your program, or a symbiont, a program-relative address is relative to the start of the load module. In other words, the address in the assembly listing must be added to the link origin (LNK ORG) address for the control section (CSECT) of your program. This shows up in the allocation map produced by the linkage editor.

For example, if you wanted to monitor from address 2A in this program listing:

| | LOC* | OBJECT CODE | ADDR1 | ADDR2 | LINE | SOURCE STATEMENT |
|---------------------|--------|----------------------------|-------|-------|------|--------------------------------|
| label for the phase | 000000 | 0560 | | | 2 | BEGIN BALR 6,0 |
| | 000002 | | | | 3 | USING 0,6 |
| | 000002 | 47F0 6010 | 00012 | | 4 | BRANCH B **16 |
| | 000006 | C1C2C3C40404040 | | | 5 | DC CLB'ABCD' |
| | 00000E | C5C6C7C8 | | | 6 | DC CL4'EFGH' |
| | 000012 | 4110 6102 | 00104 | | 7 | LA 1,LIST |
| | 000016 | | | | 8 | SNAP (1) |
| | 000016 | 0A10 | | | 9+ | DS OH |
| | 000018 | 0203 6008 600C 0000A 0000E | | | 10+ | SVC 29 SNAP 5VC |
| | 00001E | 0700 | | | 11 | TAG1 MVC BRANCH+8(4),BRANCH+12 |
| | 000020 | | | | 12 | TAG2 OPEN OUT |
| | 000020 | 4510 6026 | 00028 | | 13+ | CNOP 0,4 |
| | 000024 | 80 | | | 14+ | TAG2 EQU * |
| | 000025 | 000048 | | | 15+ | BAL 1,**(4*2) |
| | 000028 | 0A26 | | | 16+ | DC X'80' |
| address | 00002A | 0207 60F2 6004 00GF4 00006 | | | 17+ | UC AL3(OUT) |
| | 000030 | | | | 18+ | SVC 38 ISSUE 5VC |
| | | | | | 19 | MVC BUF(8),BRANCH+4 |
| | | | | | 20 | TAG3 PUT OUT |
| | | | | | 21+ | TAG3 DC 0Y(0) SET ALIGNMENT |

you would have to look at the allocation map for the LNK ORG of the CSECT (4B0):

```

** ALLOCATION MAP **
LOAD MODULE - LNKLOD          SIZE - 000005CC
PHASE NAME  TRANS ADDR  FLAG  LABEL  TYPE  ESID  LNK ORG  HIAUDR  LENGTH  OBJ ORG
LNKLOD000  MODL = ROOT
*** START OF AUTO-INCLUDED ELEMENTS -
- 75/10/04 05.59 -
PRX10E     OBJ      01     00000000 000004AF 00000480 00000000
DPXCOM7    ENTRY   01     00000000 00000000 00000000
DPXCOM6    ENTRY   01     00000000 00000000 00000000
DPXCOM1    ENTRY   01     00000000 00000000 00000000
DPXCOM6    ENTRY   01     00000000 00000000 00000000
DPXCOM2    ENTRY   01     00000000 00000000 00000000
DPXCOM5    ENTRY   01     00000000 00000000 00000000
DPXCOM4    ENTRY   01     00000000 00000000 00000000
DPXCOM3    ENTRY   01     00000000 00000000 00000000
*** END OF AUTO-INCLUDED ELEMENTS -
- 76/03/11 00.33 -
PROG       OBJ      01     000004B0 000005C6 0000011C 00000000
PROG       CSECT   01     000004B0 000005C6 0000011C 00000000
OUT        ENTRY   01     000004FB 00000498 00000048 00000000
OUTC       ENTRY   01     0000052A 0000007A 0000007A 00000000
OUTE       ENTRY   01     00000530 00000080 00000080 00000000
UUUU04B0

```

and add them together, producing 4DA as the program relative address. This applies to both single-phase and multiphase load modules. However, with the multiphase modules, additional considerations are necessary. One phase can overlay another phase, so the same program relative address can be used in more than one phase. In order to monitor the correct phase, you should use the **P=phase-name* entry discussed in 9.3.3.

If you want to monitor a transient routine, the address is relative to the start of the transient.

Another important point to note is that when using the storage reference option for a program relative address, you frequently will obtain two groups of monitor output for a given option. The first printout is produced just before the execution of the instruction that references the location. This may be either a read or write type of reference. The second printout is produced on the next instruction, but only if the data at that location has been changed. This may appear to be superfluous and even confusing (the second instruction shown will probably not even reference the area), so this printout should be considered as only a changed data confirmation.

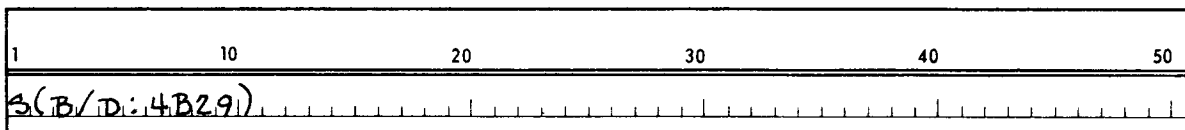
The real value of this second printout comes in those cases where the data is not changed directly, so no reference (first printout) occurs at all. This includes cases of areas changed by execute instructions (EX), supervisor call instructions (SVC), I/O operations, and occasionally even supervisor or symbiont routines running concurrently. So, in any case where a storage reference option printout seems invalid (the instruction printed does not reference the data location), check the preceding instruction in your program for an EX or SVC instruction or an I/O operation.

9.3.4.1.2. Base/Displacement Address (B/D)

To use the base/displacement address method for the storage reference option, you need this format:

S(B/D:bbbb)

Here, the *b* is the base register, and the *ddd* is the displacement; *b* can range from 0_{16} to F_{16} , and *ddd* can range from 000_{16} to FFF_{16} . For example, if you used:



an instruction that contains a storage reference of 4B29 must occur to make the monitor take action. In other words, for this option to be effective, your program must have a storage reference using base register 4 and a displacement of B29. Notice the colon separating the B/D from 4B29.

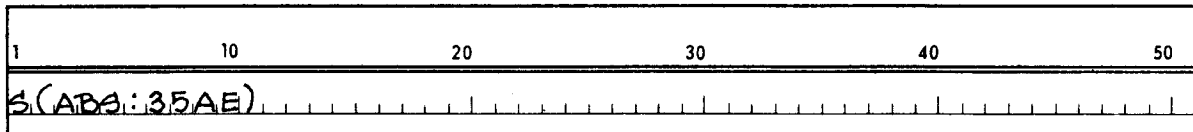
9.3.4.1.3. Absolute Address (ABS)

You use this type of option primarily when you are using system symbiont or transient routines that can refer to locations that are outside of their area. But you might also find it applicable to your program as well. It uses this format:

S(ABS:xv)

The *xv* is the absolute address, and can consist of one to six hexadecimal characters, in the range of 0_{16} to $FFFFFF_{16}$.

For example, if you want the monitor routine to take action when the program reaches an instruction that references storage at absolute address 34AE, you would code:



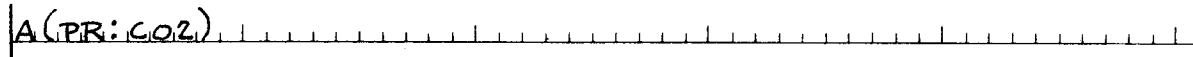
9.3.4.2. Instruction Location Option (A)

This option requests the monitor routine to take action when the specified instruction location is reached. Just as with the storage reference option, it uses the program relative address. However, you can also add a range to continue *this* monitor action for a specific number of bytes. It has only one format:

A(PR:xv) [Rnn]

The *xv* is the 1- to 6-hexadecimal-character program relative address (0_{16} to $FFFFF_{16}$). If the program reaches an instruction at this location (program relative), monitor action begins. You can also continue monitor action for this option for a length of up to 255 bytes by specifying a range (*Rnn*). The allowable values for this range field are 02_{16} to FF_{16} .

For example, if you coded either:

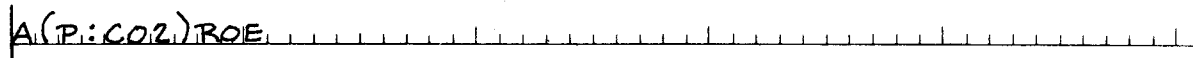


or

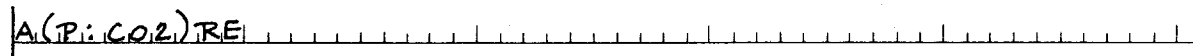


the monitor takes action for this option if the instruction at program relative address is reached.

If you coded (notice the convenient form P instead of PR):



monitor action begins when the instruction at program relative address C02 is reached, and continues for 14 bytes (OE). This means the monitor action is to continue until program relative address C10 is reached. Note that you must use two hexadecimal characters for the range even when it can be expressed in one. In the last example, if the leading 0 of OE was omitted, and it was coded as this:



monitoring would continue for 224 bytes to program relative address CE2.

9.3.4.3. Instruction Sequence Option (I)

This option requests the monitor routine to take action when the exact instruction sequence specified is reached. The monitor routine compares the machine code specified in the option entry to the actual instruction sequence of each instruction to be executed in the program being monitored, and takes action when an exact match occurs. The format for the instruction sequence option is:

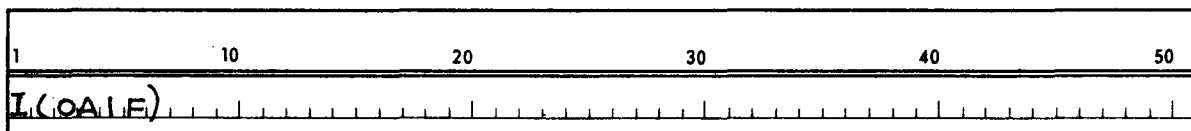
I(xmcd)

The *xmcd* stands for hexadecimal machine code. It may consist of from 2 to 64 hexadecimal characters (1 to 32 bytes). This is the value you want compared to the actual machine code being processed.

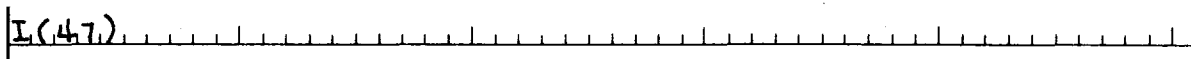
There are three different types of machine code sequences you can select:

- A single instruction
- Just the operation code of an instruction
- A string of instructions

For example, if you want monitor action to start when a supervisor call instruction for supervisor routine 31 occurs (SVC 31 in machine code = 0A1F), code it as:



If you want monitor action whenever *any* branch on condition instruction is reached (hexadecimal code = 47), you would code:



But if you want monitor action to occur whenever the following sequence of instructions occur (even though we are showing a series of inline expansion codes):

| LUC# | OBJECT CODE | ADDR1 | ADDR2 | LINE | SOURCE | STATEMENT |
|--------|----------------------------|-------|-------|-----------|--------|-------------------------------------|
| 000000 | | | | 1 | PROG | START 0 |
| 000000 | 0560 | | | 2 | BEGIN | BALR 6,0 |
| 000002 | | | | 3 | USING | **6 |
| 000002 | 47F0 6010 | | 00012 | 4 | BRANCH | B **14 |
| 000006 | C1C2C3C440404040 | | | 5 | DC | CL4*ANCD* |
| 000006 | C5C6C7CA | | | 6 | DC | CL4*EFGH* |
| 000012 | 4110 6L02 | | 00104 | 7 | LA | 1,LIST |
| | | | | 8 | SNAP | (1) |
| 000028 | 0A26 | | | A 18+ | SVC | 38 ISSUE SVC |
| 00002A | 0207 60F2 6004 000E4 00006 | | 00006 | A 19 | MVC | BUF(8),BRANCH*4 |
| | | | | A 20 | PUT | OUT |
| 000030 | | | | A 21+TAG1 | DC | OY(10) SET ALIGNMENT |
| 000030 | 5810 6116 | | 00118 | A 22+ | L | 1,*(OUT) LOAD R18, FILENAME ADDRESS |
| 000034 | 9220 1031 | | | A 23+ | MVI | 49(1),X*20* SET FUNCTION CODE |
| 000038 | 5810 1034 | | 00034 | A 24+ | L | 15,52(,1) LOAD ADDR OF COMMON 1/0 |
| 00003C | 05EF | | | A 25+ | BALR | 14,15 LINK TO COMMON |
| | | | | A 26 | TAG4 | CLOSE OUT |
| 00003E | | | | A 27+TAG4 | DC | OY(10) |
| 00003E | 5810 6116 | | 00118 | A 28+ | L | 1,*(OUT) LOAD R18, FILENAME ADDRESS |
| 000042 | 0A27 | | | A 29+ | SVC | 39 ISSUE SVC |

9.3.5. Specifying Actions

Action entries follow the option entry on the monitor statements. They share the same card; option is specified first, then any actions. Actions for an option must be completely specified on one card; no continuation to the next card is permitted. If there are duplicate or overlapping options, only the first one specified is processed, and any action specified on this second card for the same option is never considered.

There are four different types of actions $D\Delta R$, $D\Delta S$, H , or Q (plus a default), as shown in this format:

$$\left\{ \begin{array}{l} D\Delta R [n[-Rn]] \\ D\Delta S[Lnn] \left\{ \begin{array}{l} (\underline{P}R:xv) \\ (\underline{B}/D:bddd) \\ (\underline{A}BS:xv) \end{array} \right\} \\ Hccc \\ Q \end{array} \right\}$$

NOTE:

If no action is specified, the monitor routine produces a default display (9.3.5.1.3).

The $D\Delta R$ and $D\Delta S$ actions (for display register or display storage) print out program information, including specified registers ($D\Delta R$) or storage ($D\Delta S$), and continue monitor processing.

The H action (for halt) prints out the program information and suspends the job until it is told to continue.

The Q action (for quit) prints out the program information, then deactivates the monitor routine so that processing can return to normal.

If you omit an action entry, the monitor routine produces a default printout of program information (including changed registers and storage) and continues monitor processing until the end of the program.

9.3.5.1. Display Actions

There are two types of display specifications: register display ($D\Delta R$) and storage display ($D\Delta S$). But the addition of a default display provides you with the capability of having three types.

The three display actions have similar functions; that is, program information is printed, then the instruction causing the printout is executed, and program processing continues under monitor control. The printouts are basically the same, except for a few minor differences, depending upon the type of display action requested.

9.3.5.1.1. Register Display (DΔR)

If you select this type of action, you get the following items:

1. The jobname, TCB address, and program base address. Since this information does not change during the course of program execution, it is given only for the first option that causes a printout. Remember, you can have up to 15 different options; it would be senseless to print any information about the program that does not change.
2. PSW contents
3. Next instruction to execute (which is the instruction causing the printout)
4. Option causing this printout
5. The contents of the specified general registers (four bytes)

After this printout is given, the instruction executes and the program continues processing under monitor control (that is, all remaining instructions are traced to see if they match any other options that might have been specified).

You can cause one or more general registers to print by selecting one of three ways to display a register. The format shows the three different types (combined into one format):

DΔR [n[-Rn]]

- DΔR, which prints the contents of all 16 general registers
- DΔRn, which prints a specific register, with *n* being the hexadecimal number (0—F) of the register you want
- DΔRn—Rn, which allows you to print a consecutive number of registers. The first *n* indicates the first register (0—F), and the second *n* indicates the last register (0—F).

For example, if you wanted to display register 15 when the program reaches a program relative address based on the instruction at assembly address 2A in this listing (remember, the assembly address (2A) must be added to the LNK ORG address, which in this case is 4B0, to obtain the program relative address — 4BA):

| LOC. | OBJECT CODE | ADDR1 | ADDR2 | LINE | SOURCE STATEMENT |
|--------|----------------------------|-------|-------|---------|---|
| 000000 | | | | 1 | PROG START U |
| 000000 | 0540 | | | 2 | BEGIN BALR 6,0 |
| 000002 | | | | 3 | USING *,6 |
| 000002 | 47F0 6010 | | 00012 | 4 | BRANCH B **16 |
| 000006 | C1C2C3C40404040 | | | 5 | DC CL8*ABCD* |
| 00000E | C5C6C7C8 | | | 6 | DC CL4*EFGH* |
| 000012 | 4110 6102 | | 00104 | 7 | LA 1,LIST |
| | | | | 8 | SNAP (1) |
| 000016 | | | A | 9+ | DS OH |
| 000016 | 0A1D | | A | 10+ | SVC 29 SNAP SVC |
| 000018 | D203 6008 600C 0000A 0000E | | | 11 | TAG1 MVC BRANCH*8(4),BRANCH+12 |
| | | | | 12 | TAG2 OPEN OUT |
| 00001E | 0700 | | A | 13+ | CNOP 0,4 |
| 000020 | | | A | 14+TAG2 | EQU * |
| 000020 | 4510 6026 | | 00028 | A | 15+ |
| 000024 | 80 | | A | 16+ | BAL 1,*(4*2) |
| 000025 | 000048 | | A | 17+ | DC X*80* |
| 000028 | 0A26 | | A | 18+ | DC AL3(OUT) |
| | | | A | 19+ | SVC 38 ISSUE SVC |
| | | | A | 19 | MVC BUF(8),BRANCH+4 |
| | | | A | 20 | TAG3 PUT OUT |
| 000030 | | | A | 21+TAG3 | DC (Y(0) SET ALIGNMENT |
| 000030 | 5810 6116 | | 00118 | A | 22+ |
| | | | | | L 1,*(A(OUT) LOAD K18, FILENAME ADDRESS |

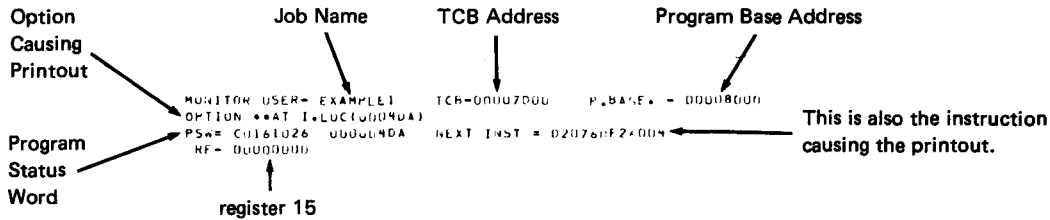
address → 00002A 0207 60F2 6004 000F4 00006

you would code:

```

1          10          20          30          40          50
|-----|-----|-----|-----|-----|
| A(PR:4DA) D R |-----|-----|-----|-----|
|-----|-----|-----|-----|-----|
  
```

and your monitor printout would look like this:



If, for the same program, you coded:

```

|-----|-----|-----|-----|-----|
| A(PR:4DA) D R |-----|-----|-----|-----|
|-----|-----|-----|-----|-----|
  
```

The contents of all 16 registers (plus items 1 through 4) are displayed, like this:

```

MONITOR USER- EXAMPLE3  TCB-00007000  P.BASE. = 00008000
OPTION **AT I.LOC(00040A)
PS= C0161026 00004DA  NEXT INST = 020760F2A004
R0- 00000061  R1- 800004F8  R2- 00000000  R3- 00000000  R4- 00000000  R5- 00000000  R6- 000004B2  R7- 00000000
R8- 00000000  R9- 00000000  RA- 00000000  RB- 00000000  RC- 00000000  RD- 00000000  RE- 000004DA  RF- 00000000
  
```

Both of these examples would have continued monitoring for the option until the end of the job. However, if you add a quit action (Q, which will be explained in 9.3.5.3) you would have obtained the same printout and discontinued monitor control. This holds true for all options. If you want to monitor only a specific area or instruction, it is advisable to end the option with a quit action, so additional processor time is not wasted by having the monitor search when there is nothing left to find. Coded with a quit action ending the option in the last example, it would have looked like:

```

|-----|-----|-----|-----|-----|
| A(PR:4DA) D R;Q |-----|-----|-----|-----|
|-----|-----|-----|-----|-----|
  
```

Notice that a semicolon is used to separate the actions.

9.3.5.1.2. Storage Display (DΔS)

Most of the information provided by a storage display type of action is similar to that of a register display (9.3.5.1.1): you get items 1, 2, 3, and 4. However, item 5 is different; the storage display action prints out the contents of specified storage locations.

After the printout is given, the instruction executes and program processing continues under monitor control.

You can specify up to 256 consecutive bytes of main storage with a length option, or the monitor prints (by default) 8 consecutive bytes starting at the specified storage location.

Just as in displaying registers, the storage display action has three different types, but each is shown in its own format, because of their diverse range of actions:

DΔS[Lnn](PR:xv)

DΔS[Lnn](B/D:bccc)

DΔS[Lnn](ABS:xv)

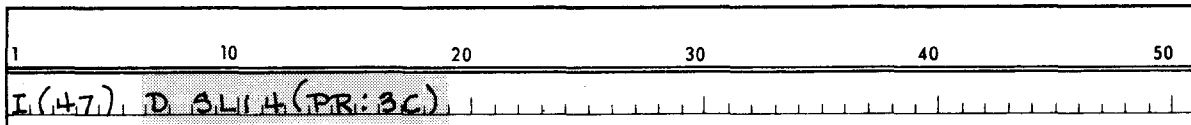
Each one has a length option, shown as *Lnn*, which allows you to specify how many consecutive bytes of main storage you want displayed. *L* indicates that this is a length specification, with *nn* as the length, in the range of 01_{16} to FF_{16} (allowing you to display 256 bytes).

The item after each length expresses the method in which you want to display a specified location in main storage. They have the same format and meaning as the storage reference options explained in 9.3.4.1, but are not to be confused as to function (action versus option):

- (PR:xv) is used to display a main storage area starting at a program relative address.
- (B/D:bddd) displays a main storage area using base/displacement.
- (ABS:xv) displays storage starting at an absolute address.

The *xv* is the address for program relative and absolute addressing locations, in the range of 0_{16} to $FFFFFF_{16}$. The *bddd* is for the base displacement method, where *b* indicates the number of the base register (the range is 0_{16} to F_{16}), and *ddd* is the displacement (in the range of 000_{16} to FFF_{16}).

For an example of the option, we will use an instruction sequence (I) to prevent any confusion that might initially arise by seeing similar codes (such as a program relative option (PR) and a storage display action starting at a program relative address) on the same line:



This displays 20 bytes (14_{16}) starting at program relative address 3C. This happens whenever any branch condition is reached in the program (hexadecimal code 47).

If you want to display eight bytes (default) starting at the address using base register 1 and a displacement of B29 whenever any branch condition is reached, you would code:

```

1          10          20          30          40          50
|-----|-----|-----|-----|-----|
I(47), D 8(B/D:1B29)
  
```

If you want to display the default eight bytes starting at absolute address 35AE whenever any branch condition is reached, code:

```

|I(47), D 8(ABS:35AE)
  
```

If you wanted only four bytes at absolute address 35AE whenever any branch condition is reached, code:

```

|I(47), D 4(ABS:35AE)
  
```

Notice that you must use two hexadecimal characters for the length even when it can be expressed in one.

The following example uses a program relative option and a program relative address for the action:

```

|A(PR:2A), D 8(PR:3C)
  
```

When the instruction at program relative address 2A is reached, a storage display of eight bytes starting at program relative address 3C is produced.

9.3.5.1.3. Default Display

You can omit the action specification; that is, you can enter an option without specifying any particular action you want taken when the monitor option becomes effective. In this case, the monitor routine prints out items 1, 2, 3, and 4 listed in 9.3.5.1.1, and (items 5 and 6):

5. The contents of any general registers that were changed since the last printout was given. If this is the first action taken by the monitor routine for this program, the present contents of all the general registers is printed.
6. The contents of the storage locations referenced by the instruction causing the printout.

The instruction causing the printout is then executed, and program processing continues under monitor control.

For example, assume that the following option statement was the only input to the monitor routine (and the task statement):

| | | | | | |
|-------------|----|----|----|----|----|
| 1 | 10 | 20 | 30 | 40 | 50 |
| S(B/D:4B29) | | | | | |

When the program reaches an instruction that references an address using base register 4 and a displacement of B29, a default display is given.

Remember, you can also get a default by omitting the option statement (9.3.4.5). The only difference between the default display caused by omitting the option and the default display caused by omitting the action is that the omission of the option means that the option causing the display is not printed.

9.3.5.2. Halt Action (H)

This action, like the other actions, prints out items 1, 2, 3, and 4 (detailed in 9.3.5.1.1). It then prints a halt message on the system console and suspends program execution until a reply from the console operator allows execution to continue.

The halt message sent to the system console has the following format:

HALT ccc. TYPE-IN GO jobname TO RESUME

Program execution is then suspended until the operator issues the GO command followed by the job name (same as that on the JOB control statement). You can then provide the operator with special instructions about what to do before entering the GO command, such as taking a main storage dump. After he completes these special instructions, and enters the GO command, the instruction causing the halt is executed, and program processing continues under monitor control.

The format for the halt action is:

Hccc

The *ccc* is a 3-character EBCDIC code that you specify to identify the halt, and corresponds to the *ccc* in the halt message displayed to the operator.

For example, assume that your JOB control statement has a job name of TWESTMON, and uses the following monitor statement:

| |
|----------------|
| A(PR:1B4) HDMP |
|----------------|

When the program reaches the instruction at program relative address 1B4, the monitor routine prints out the program information and displays the following message on the system console:

HALT DMP TYPE-IN GO TWESTMON TO RESUME

You would instruct the operator to take your desired action when he sees this message. In this case, assume it is a dump. After issuing the DUMP command (and a dump of main storage is given), the operator would then type:

GO TWESTMON

to reactivate the interrupted job. The instruction at program relative address 1B4 is then executed, and program processing continues under monitor control.

9.3.5.3. Quit Action (Q)

The quit action (Q) prints out items 1 through 4 and nothing else. The instruction causing the printout is then executed, and program processing continues without any further monitor intervention (pertaining to the option to which this action applies).

This action is useful when you want to monitor a problem area in the beginning of your program, and then exit from the monitor routine without tracing all the remaining instructions in the program (thus not wasting execution time).

The format for the quit action is:

Q

For example, if you coded:

| | | | | | |
|-------------|----|----|----|----|----|
| 1 | 10 | 20 | 30 | 40 | 50 |
| A(PR:F18) Q | | | | | |

the monitor routine would print out the program information when program execution reaches the instruction at program relative address F18. This instruction is then executed, and program processing continues without monitor intervention.

When the quit action is not used as one of the actions for an option, monitor processing continues until the end of the job step.

Table 9-2 summarizes the program information that is displayed by each action.

Table 9—2. Summary of Actions and Program Information Printed

| Program Information Printed | Action | | | | |
|---------------------------------|------------------------|-----------------------|-----------------|----------|----------|
| | Display Register (D R) | Display Storage (D S) | Default Display | Halt (H) | Quit (Q) |
| Job name* | x | x | x | x | x |
| TCB address* | x | x | x | x | x |
| Program base address* | x | x | x | x | x |
| PSW contents | x | x | x | x | x |
| Next instruction to execute | x | x | x | x | x |
| Option causing this printout | x | x | x | x | x |
| Contents of specified registers | x | | | | |
| Contents of specified storage | | x | | | |
| Contents of changed registers | | | x | | |
| Contents of referenced storage | | | x | | |
| HALT message | | | | x | |

*These items are included for only the first option that causes a printout.

9.3.6. Cancel of Monitor

If the monitor routine is terminated abnormally, either by a CANCEL command or by a program exception within the monitor routine, all programs requesting the monitor routine will continue normal program processing without any type of monitor intervention. The monitor routine itself will dump and leave the system.

9.4. SYSTEM DEBUGGING AIDS

Several debugging aids are built into the OS/3 supervisor to aid in solving system problems which cannot be identified through a normal SYSDUMP. These aids are useful only with some knowledge of the internal supervisor structure and are therefore not intended for general use. This section is provided for informational purposes only.

Table 9—3 summarizes the debugging aids described on the following pages. Refer to the appropriate operations handbook for your system for the correct procedure of altering main storage using the maintenance panel.

9.4.1. Supervisor Debug Option

The supervisor debug option is set at initial program load (IPL) time by entering D as the final character (following the comma) of the initial IPL message. This is described in the operations handbook. Use of this D option causes the supervisor being loaded to be expanded in size (by about 1K or less) to support the supervisor debug option.

The following functions are provided:

- A normal halt (HPR code 99130001) between IPL and supervisor initialization. This allows changes to be made to the supervisor (via the maintenance panel) prior to loading the supervisor initialization load module. Normally, however, you should simply press the RUN switch on the maintenance panel to continue.
- A pseudo monitor to detect when any byte within the supervisor has been changed. This function is activated when a 2-byte (nonzero) address is stored into absolute location 000E via the maintenance panel. The byte specified by the address in location E is checked on every interrupt and on every pass through the switcher. When changed, the supervisor halts (HPR code 99130002) without restoring the original contents. If you want to continue, simply press RUN. The *new* value becomes the *original* value and the supervisor will halt if the byte is changed again.
- Verification of the 12 low-order bytes of main storage (locations 0—B) on every interrupt and every pass through the switcher. When changed, the supervisor saves the incorrect setting, restores the correct setting and halts (99130003). Although you may continue past this HPR by pressing RUN, you should take a SYSDUMP here to determine why these bytes are being altered.
- A resident supervisor monitor to detect when any byte within the supervisor has been changed. This function is activated when a 2-byte (nonzero) address is stored into absolute location 00D2 via the maintenance panel. The byte specified by the address in location D2 is monitored on every instruction executed by supervisor critical code (interrupt processing), transients, symbionts, and job control. The only code not monitored is code being executed under a key other than 0 (i.e., user jobs). Monitoring user jobs is unnecessary because the hardware key protection feature of the processor prevents user jobs from destroying any part of the supervisor.

When the byte specified by location D2 is changed, the resident monitor halts (99130004) without restoring the original contents. The double word at location 80 contains the PSW at the time the byte was changed. If you want to continue, simply press RUN. The *new* value becomes the *original* value and the supervisor will halt if the byte is changed again.

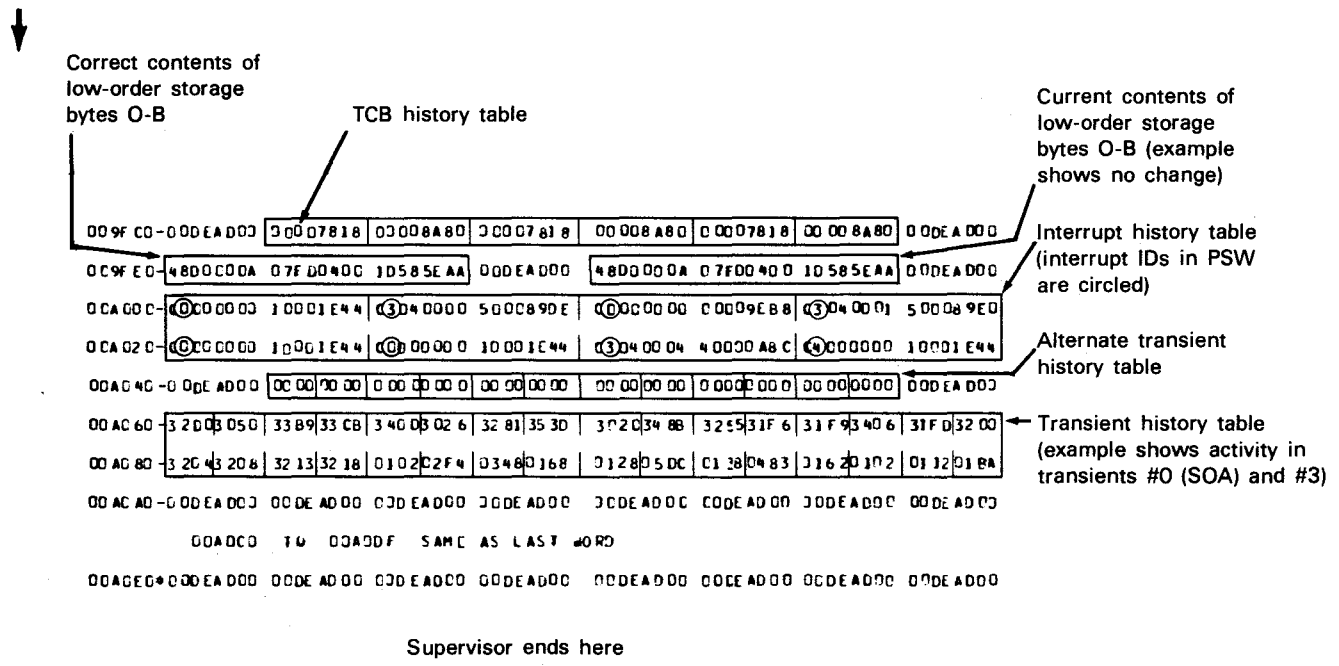
When using the resident monitor, you may notice that the operating system is performing slower than it normally would. This is because the software is not executing approximately 8 instructions for every previous 1. For this reason, it is advisable to set up the resident monitor as near to the suspected problem as possible.

To change the byte being monitored, simply change the half word at location D2. To turn off the resident monitor, reset location D2 to 0000. In both cases, the change will automatically take effect at the next interrupt.

The resident supervisor monitor must not be used when either the standard monitor (9.3) or the mini monitor (9.4.2) is active.

- History tables that provide the following information:
 - TCB History Table. This shows the absolute addresses of the last 6 TCBs given control by the switcher. If the switcher gives control to the same task which had control prior to an interrupt, the TCB address is not listed again.
 - Interrupt History Table. This shows the PSW (8 bytes) at the time of the last 8 interrupts. Bits 4—7 of each PSW is set to an interrupt ID, as follows:
 - 0 = IOST interrupt
 - 1 = Machine check interrupt
 - 2 = Program exception interrupt
 - 3 = SVC interrupt
 - 4 = Timer interrupt
 - Alternate Transient History Table. This shows the transient IDs of the last 12 transients loaded from the alternate transient file (\$Y\$STRANA). This table would normally be all zeros.
 - Transient History Table. This shows the transient IDs (12 bits) of the last 32 transients loaded by transient management. These are listed in 2-byte entries, with the high order 4 bits containing the transient area number (0 means supervisor overlay area (SOA).) Reused transients are not included.

The history tables always reside at the end of the supervisor. They can be easily identified in a dump by the pattern 00DEAD00 repeated across the line immediately following the tables. The entries in each table are always arranged from the oldest (lower addresses) to the newest (higher addresses). Following is an example of a history table maintained by the supervisor debug option.



9.4.2. Mini Monitor

The mini monitor is a small, specialized version of the standard monitor described in 9.3. It is intended for system debugging only and does not replace the standard monitor symbiont.

The mini monitor can monitor any one byte in main storage during execution of user, transient, or supervisor resident code and will halt when the specified byte is changed. It is called by the MM console command.

Format:

MM value,address,RTUE

Positional Parameter 1:

value

A 2-character hexadecimal value that specifies the correct contents of the byte being monitored.

Positional Parameter 2:

address

Specifies the absolute main storage address of the byte to monitor. This must be specified as a 5-character hexadecimal value (zero fill on the left).





Positional Parameter 3:

RTUE

Flags that specify how the mini monitor is to function. Any combination of 1 to 4 characters may be specified:

- R — Monitor resident code, including I/O, program exception, supervisor call (SVC), and interval timer interrupt processing.
- T — Monitor transients, including the supervisor overlay area (SOA).
- U — Monitor user jobs and symbionts active at the time MM was keyed in.
- E — Instead of halting when the specified byte is changed, the halt will occur when the monitored byte equals the value specified in positional parameter 1.

The mini monitor HPR code is 9912. The monitor interrupt old PSW (low memory location 80—87) contains the address of the instruction that immediately follows the instruction that altered the byte. If the byte was altered by code not being monitored, the halt will occur at the first monitored instruction.

Once the mini monitor is called, it cannot be turned off or changed except by again loading and initializing the system. Only one version of the mini monitor can be executed at a time and it cannot be run with the standard monitor in main storage or with the resident monitor.

Because execution time of monitored code is increased by a factor of about 9, use of the mini monitor should be limited. It is advisable to make the MM keyin as near to the suspected problem as possible.

You may have noticed that the mini monitor offers many of the same features as the pseudo monitor and the resident supervisor monitor. All can monitor resident (critical) code and transients (including SOA). The differences are:

■ Mini Monitor

- This monitor has the additional ability to halt when a byte is changed to a specified value.
- User jobs can be monitored (if the absolute address of the job is known).
- Impact on the system can be minimized by monitoring only certain TCBs. For example, you could monitor resident interrupt processing without monitoring transients, symbionts, or user jobs.





- Resident Supervisor Monitor
 - This monitor can be turned on while the processor is halted. This enables you to monitor during a more precise period of time.
 - The monitor can be turned off or changed easily with no necessity to IPL again.
 - New tasks being created (e.g., new symbionts or new job control TCBs) will automatically be monitored.
- Pseudo Monitor
 - All the advantages of the resident monitor except that the byte is only checked on interrupts and switcher calls.
 - The impact on the system is unnoticeable. You can monitor over a long period of time.

9.4.3. Console Debug Options

Three debug options that can be set by console commands are available for supervisor debugging:

- PIOCS debug option. Causes system halt (HPR code 990F) on any CCB checksum error or program check during PIOCS. The console command is:

SET DE,IO

- Transient debug option. Causes system halt (HPR code 9908) on any transient error (i.e., error normally producing a lxx error code). This is useful because normal recovery from a lxx error code often causes the offending transient to be overlaid by other transients. The console command is:

SET DE,TR

- Loader debug option. Causes system halt (HPR code 9915) whenever the loader detects any error other than 51 (module not found). A SYSDUMP taken at this halt will provide useful information in determining the exact cause of any loader error (52—5F) which cannot otherwise be diagnosed. The console command is:

SET DE,LD





9.4.4. Transient Halt Location

When trapping system problems, it is often desirable to halt the processor whenever a particular transient is loaded. Every transient is uniquely identified with a *transient ID*. By storing this ID into location 000C (2 bytes) in low order main storage, you can cause the system to halt (HPR code 990C) whenever transient management loads that transient or overlay into a transient area. The halt occurs less than 10 instructions before the transient is given control and you can continue normally by pressing RUN.

Note that this halt occurs only when a transient has just been loaded from SYSRES. Some transients can be reused in memory; the halt will, therefore, occur only when it is initially loaded.

When this halt occurs, problem register 15 can be used to find the address of the transient area into which the transient was just loaded. The appropriate operations handbook for your system describes the procedure for reading problem registers from the maintenance panel.

9.4.5. Symbiont Halt Location

The symbiont halt location is a 4-byte field at location 00DC in low-order main storage. It is used to halt the processor whenever a particular symbiont or phase of a symbiont is loaded. This could be useful when debugging a particular symbiont.

To set the symbiont halt, set the half word at location DC to the EBCDIC value of the 2-character symbiont ID. If you want to stop a phase other than the root phase, also set the half word at location DE to the EBCDIC phase number (00—99).

The halt (HPR code 997C) occurs less than 10 instructions prior to the symbiont phase being given control following the LOAD or FETCH. To continue normally following the halt, press RUN.

Examples:

| Contents of symbiont halt location | Processor halts when this symbiont phase is loaded | Symbiont ID |
|------------------------------------|--|-------------|
| D7D9F0F0 | SL\$\$OW00 | PR |
| D9E4F1F4 | JL\$\$RU14 | RU |
| E2E40000 | SL\$\$SU00 | SU |
| E2E4F0F0 | SL\$\$SU00 | SU |



Table 9-3. Summary of System Debugging Aids

| Function | Use | How to Set | Results |
|-------------------------|--|---|--|
| Supervisor Debug Option | | Key in D in last blank of initial IPL message | Normal HPR code 99130001 at end of IPL (press RUN to continue) |
| Pseudo monitor | To identify the routine changing a particular byte | Set location E to address of byte to check | HPR code 99130002 (press RUN to continue) |
| Verify bytes 0-8 | To identify the routine destroying low memory | Included automatically | HPR code 99130003 (press RUN to continue) |
| Resident monitor | To identify the instruction changing a particular byte | Set location D2 to address of byte to check | HPR code 99130004 (press RUN to continue) |
| History tables | To provide some recent history information in SYSDUMPs | Included automatically | Continuous updating of resident tables |
| Mini monitor | To identify the instruction changing a particular byte | Console command MM value, address, RTUE | HPR 9912 (press RUN to continue) |
| Console debug option | | Console command | |
| PIOCS | To identify checksum errors or internal PIOCS problems | SET DE,IO | HPR code 990F |
| Transient | To halt on 1xx errors | SET DE,TR | HPR code 9908 |
| Loader | To halt on errors 52-5F | SET DE,LD | HPR code 9915 (press RUN to continue) |
| Transient halt location | To halt if and when a particular transient is loaded | Maintenance Panel put 2-byte transient ID into location C | HPR code 990C (press RUN to continue) |
| Symbiont halt location | To find out if and when a particular symbiont (or symbiont phase) is loaded. | Maintenance panel DC-DD=EBCDIC ID DE-DF=EBCDIC phase number | HPR code 997C (press RUN to continue) |

10. Message Display, Logging, and Operator Communication

10.1. GENERAL

Successful operation of a computer system requires constant communication. You use job control statements, assembler instructions, and supervisor macro instructions to tell the CPU what to do, and how and when to do it. The operating system tells the operator what to do, and tells you what was done and when. The operator gets a message from the supervisor (or from you) and answers a question or performs an action.

OS/3 provides several methods by which you can communicate with the operating system and with the console operator. These consist of a system log, display to the operator, and a canned message file, which can be used singly or in combination.

A system log file is maintained by the supervisor spooling function. Job logs are subfiles of the system log file and receive all log and accounting information for the job including messages you write to the log using macro instructions in your program. (See Section 11 for a description of spooling, job logs, and job accounting.)

You can display a message to the operator at the system console. The message may be for information only, or you may request a reply by the operator. Also, you can combine a log entry and a display. In this case, the message displayed and any reply from the operator is written to the system log, and also to a console log if one is configured at system generation. ←

You can display or log a message either from main storage or from a file of "canned" messages maintained on disk by the supervisor. The canned message file is a set standard message identified by canned message numbers that may be used by any BAL program, and may be examined, printed, displayed, logged, etc. A canned message may also contain blanks which are replaced by variable characters from a buffer whose address is specified as a parameter in a macro instruction. If a canned message is specified and there are user supplied variable characters, these characters are automatically inserted into the canned message before the completed message is displayed, logged, or stored in your specified buffer area.

You may need a canned message for some output other than display or log. In this case you can retrieve it from the canned message file, with or without variable characters, and store it in a buffer specified in your program.

There are four macro instructions you can use to retrieve, log, or display messages; these are:

- **WTL**
Writes a message into the system log file.
- **WTLD**
Writes a message into the system log file after displaying it on the system console.
- **GETMSG**
Gets a message from the canned message file.
- **OPR**
Displays a message to the operator on the system console.

The WTL, WTLD, and GETMSG macro instructions are described in 10.2. The OPR macro instruction is described in 10.3. Table 10—1 shows some of the options and characteristics of these four macro instructions.

Table 10—1. Summary of Message Macro Instructions

| Macro Name | Message From | | Output To | | | Message Length | |
|------------|--------------|---------------------|------------|-----------------|--------------|----------------|---------|
| | Main Storage | Canned Message File | System Log | System Console* | Main Storage | Maximum | Default |
| | | | | | | | |
| WTL | x | x | x | | | 120 | 60 |
| WTLD | x | x | x | x | | 60 or 120** | 60 |
| OPR | x | x | | x | | 60 | 60 |
| GETMSG | | x | | | x | 120 | 60 |

*Output also to console log if configured at system generation.

**Maximum 60 characters if operator reply is requested; maximum 120 characters if operator reply is not requested.

10.1.1. The Canned Message File

The canned message file is a contiguous data set embedded within the supervisor transient file. This is a file of generalized messages used by all operating system modules for display, logging, error notification, etc. The generalized messages are added to the canned message file included with a release of the system. Each message is assigned a canned message number by the message file librarian. This 2-byte number identifies the message and specifies its position within the file, and is used by the macro instructions to locate the message within the file.

10.1.1.1. Canned Messages

An important advantage of the use of canned messages is that communication can be standardized between the operating system, the operator, and you. Console messages and job log entries can be standardized within your job step and your job, and throughout the entire installation. Also, it keeps the amount of main storage required in your program for messages and log entries to a minimum. This is especially beneficial when your program contains many messages, particularly long ones.

For each of the four macro instructions mentioned earlier, the first parameter specifies the address of a buffer in main storage. This buffer contains either the actual message, or the number of the canned message. If the buffer contains a message, this is the information displayed or logged. If the buffer contains a canned message number, the routine gets this message from the file of canned messages.

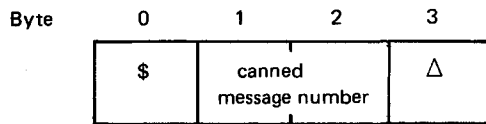
10.1.1.2. Inserting Variable Characters in a Canned Message

If you use a canned message in which variable characters are to be inserted, the buffer also contains the actual insert characters. In this case, you create the canned message so that an underline (EBCDIC hexadecimal code 6D) represents a byte into which a character is to be inserted. The macro instruction routine scans the canned message from left to right and, when an underline is found, moves a character from the string of insert characters in the buffer to this position in the message. The next character in the string replaces the next underline in the message, etc. This process continues until either an EBCDIC hexadecimal 08 is found in the character string or the length of the canned message has been scanned.

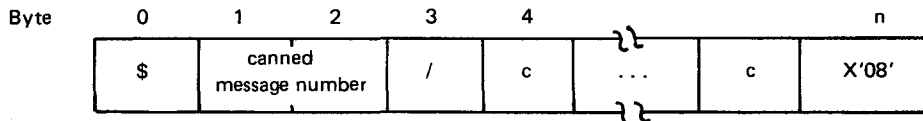
Messages that are not intended for the system console can contain a maximum of 120 characters after any variable characters have been inserted (WTL and GETMSG macro instructions). Messages intended for the system console, but not the system log, are limited to 60 characters (OPR macro instruction with or without a request for a reply). Messages to be logged and displayed are limited to 60 characters if a reply is requested, and 120 characters if a reply is not requested (WTL macro instruction).

The format for the canned message buffer is shown in Figure 10—1. The insertion of variable characters from the buffer into a canned message is pictured in Figure 10—2.

Buffer Format for Canned Messages Without Insert Characters:



Buffer Format for Canned Messages With Insert Characters:



\$

First character of the buffer; indicates that this buffer pertains to a canned message. Do not use a dollar sign as the beginning character of any other type of message.

canned message number

The (16 bit) canned message number as a binary value.

Δ(space)

A space in byte 3 indicates there are no characters to be inserted into the canned message.

/

A slash in byte 3 indicates there are characters to be inserted into the canned message.

c

If present, the characters to be inserted into the canned message are contained in the buffer starting at byte 4.

X'08'

Hex code '08' used to terminate the character string.

Figure 10-1. Canned Message Buffer Formats

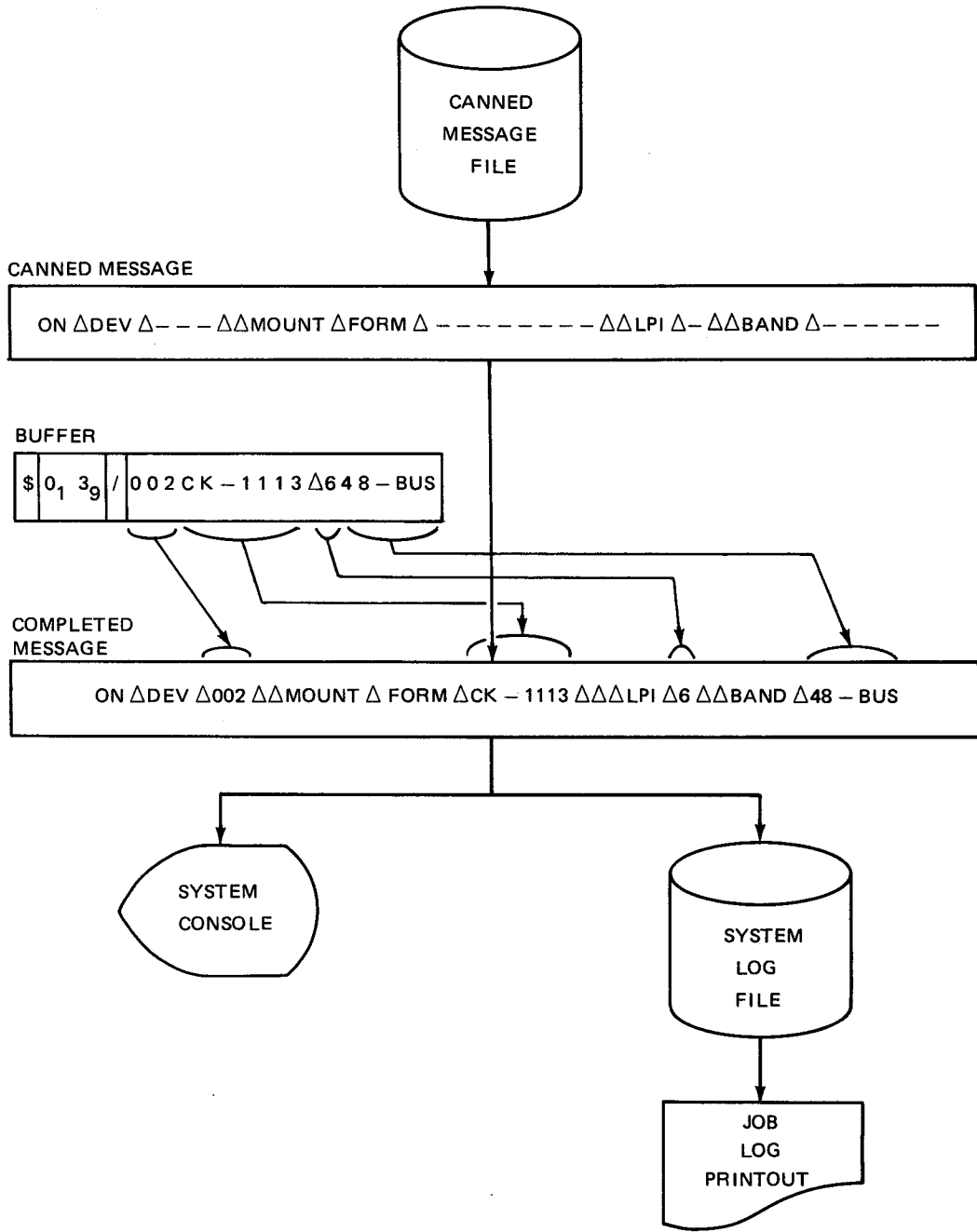


Figure 10-2. Insertion of Variable Characters in a Canned Message

10.1.2. The System Log

The system log file is a set of subfiles (job logs) within the system spoolfile that receives all job log and accounting information for each job. The job log includes all system console messages to the operator generated by the operating system; for example, device assignments and any reply entered by the operator. The log should contain any information which you may find pertinent to your job's execution. At a minimum, the log should contain a concise message for all errors encountered and all milestones passed. You use the WTL and WTLD macro instructions to write messages to the log file.

The job accounting portion includes accounting information generated by the operating system for the job, such as the number of input/output operations, CPU time charged to the job, etc.

Log file entries are destined for a printer and therefore are limited to 120 characters per line. Each print line is considered to be a logical record within the log file. Each record contains control information defining whether the record is a logged message or an accounting record. Normally, the log for a job is printed on a high speed printer as soon after job termination as possible, although printing of a job log can be initiated before the job terminates. (See 11.1.) The log can also be retained on magnetic tape for later printing or use.

10.2. MESSAGE AND LOGGING MACRO INSTRUCTIONS

10.2.1. Write to the Log (WTL)

Function:

The WTL macro instruction writes a message to the system log file for subsequent printing on a high speed printer. The message may be either currently in main storage or retrieved from the canned message file. If you specify a canned message, the macro instruction routine inserts any user-supplied variables into the message before writing it to the log. The format of the canned message buffer is shown in Figure 10-1. The insertion of variable characters is illustrated in Figure 10-2.

Because messages written to the log are destined for the printer, they are limited to a maximum of 120 characters. Each message occupies one print line or less than a line. Normally, job logs are printed as soon after job termination as possible. However, printing of a job log can be initiated before job termination. (See 11.1.)

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|---|
| [symbol] | WTL | $\left\{ \begin{array}{c} \text{buff-addr} \\ (1) \end{array} \right\} \left[\left\{ \begin{array}{c} \text{msg-length} \\ (0) \\ 60 \end{array} \right\} \right] \left[\left\{ \begin{array}{c} \text{error-addr} \\ (r)_3 \end{array} \right\} \right]$ |

Positional Parameter 1:

buff-addr

Specifies the symbolic address of the message to be logged. This may be either the address of a buffer area in main storage containing the complete message or the address of a buffer area in main storage containing the canned message number and any variable characters to be inserted.

If a canned message is specified, the buffer must be at least four bytes long (See Figure 10—1.) The first character in the canned message buffer must be a dollar sign (\$). Do not use a dollar sign as the beginning character of any other type of message.

(1)

Indicates that register 1 has been preloaded with the address of the message area.

Positional Parameter 2:

msg-length

Specifies the length in bytes of the message to be logged. For canned messages, this specifies the length of the completed message including any inserted variable characters. Maximum length for the completed message is 120 bytes.

(0)

Indicates that register 0 has been preloaded with the length of the message.

If omitted, a length of 60 bytes is assumed.

Positional Parameter 3:

error-addr

Specifies the symbolic address of an error routine that receives control if an error occurs.

(r)₃

Specifies that the designated register (other than 0 or 1) has been preloaded with the address of the error routine.

If omitted, the requesting task is abnormally terminated if an error occurs.

Following is an example of how the WTL macro instruction can be used to log a message from main storage.

Example:

| 1 | LABEL | Δ OPERATION Δ | OPERAND | Δ | COMMENTS |
|----|-------------|---------------|--|---|----------|
| | | 10 | 16 | | |
| 1. | | WTL | BUF. OUT. 1, 38, ERRORAD | | |
| 2. | | . | | | |
| 3. | | . | | | |
| 4. | | . | | | |
| 5. | ERRORAD | EDJ | | | |
| 6. | BUF. OUT. 1 | DC | C 'COUNT COMPLETED FOR BRASS CASTING DEPT' | | |

Line 1 writes a message consisting of 38 characters from main storage location BUFOUT1 to the log. If an error occurs during execution of this macro instruction, control is transferred to line 5, which specifies a normal job step termination. Line 6 defines a 38-byte output buffer containing the output message.

Assume that the same message 'COUNT COMPLETED FOR BRASS CASTING DEPT' is message number 89 in the canned message file. You can log this message using a buffer of only four bytes, as shown in the following example.

Example:

| 1 | LABEL | Δ OPERATION Δ | OPERAND | Δ |
|----|---------|---------------|-------------|---|
| | | 10 | 16 | |
| 1. | | WTL | BUFOUT2, 38 | |
| 2. | | . | | |
| 3. | | . | | |
| 4. | | . | | |
| 5. | BUFOUT2 | DC | C '\$' | |
| 6. | | DC | X L2 '89' | |
| 7. | | DC | C ' ' | |

Line 1 logs a message specified in the output buffer BUFOUT2, which in this case contains the canned message number '89'. Thus we can write a 39-byte message using a buffer in the program of only four bytes. In this example, we have omitted parameter 3, which means the task is abnormally terminated if an error occurs. Lines 5, 6, and 7 define the 4-byte output buffer.

Suppose we want to use the same canned message to refer to several departments, such as: BRASS CASTING, BRONZE CASTING, SHOT BLASTING, PAPER BOX SLITTING, etc. We can store the variable characters, in this case BRASS CASTING, in the output buffer. These characters will be inserted into the locations marked by underlines in canned message number 90, which looks like:

COUNT COMPLETED FOR _____ DEPT

After the variable characters have been inserted, the following message is logged:

COUNT COMPLETED FOR BRASS CASTING _____ DEPT

Following is an example of how we can write a 45-byte message using a buffer in the program of only 17 bytes.

Example:

| 1 | LABEL | ΔOPERATIONΔ | OPERAND | Δ |
|----|---------|-------------|------------------|---|
| | | 10 | 16 | |
| 1. | | WTL | BUFDUT3,45 | |
| 2. | | . | | |
| 3. | | . | | |
| 4. | | . | | |
| 5. | BUFDUT3 | DC | C:'\$' | |
| 6. | | DC | X.L.2'910' | |
| 7. | | DC | C:'/' | |
| 8. | | DC | C:'BRASSCASTING' | |
| 9. | | DC | X.'08' | |

Line 1 writes canned message 90 with a length of 45 bytes to the log. Lines 5 to 9 define the 17-byte output buffer.

Other WTL macro instructions in the program can use the same canned message, substituting an output buffer containing another department name, such as: BRONZE CASTING, SHOT BLASTING, etc, as we described earlier.

10.2.2. Display a Message and Write to the Log (WTLD)

This macro instruction operates in a manner similar to the WTL macro instruction in that you specify a message from main storage or a canned message with or without variable characters. However, the message to be logged is first displayed on the system console. Also, there is the additional capability to request a reply by the operator. In this case, you can specify a second buffer to receive the operator's reply.

Function:

The WTLD macro instruction writes a message to the system log file for subsequent printing on a high speed printer and simultaneously displays the message on the system console for operator reply or information. The message may be either currently in main storage or retrieved from the canned message file. If you specify a canned message, the macro instruction routine inserts any user-supplied variables into the message before the visual display and writing to the log. The format of the canned message buffer is shown in Figure 10-1. The insertion of variable characters is illustrated in Figure 10-2.

Because messages written to the log are destined for the printer, they are limited to a maximum of 120 characters. Each message occupies one print line or less than a line.

Messages are displayed on the console 60 characters per line with messages longer than 60 characters occupying two lines.

When an operator reply is requested, do not use a message longer than 60 characters because the reply will be written with the message to the system log file making a total of 120 characters.

Normally, job logs are printed as soon after job termination as possible. However, printing of a job log can be initiated before job termination. (See 11.1.)

Format:

| LABEL | Δ OPERATION Δ | OPERAND |
|----------|---------------|--|
| [symbol] | WTLD | $\left\{ \begin{array}{l} \text{buff-addr-1} \\ (1) \end{array} \right\} \left[, \left\{ \begin{array}{l} \text{msg-length} \\ (0) \\ 60 \end{array} \right\} \right] \left[, \left\{ \begin{array}{l} \text{error-addr} \\ (r)_3 \end{array} \right\} \right]$ $\left[, \text{REPLY} \right] \left[, \left\{ \begin{array}{l} \text{buff-addr-2} \\ (r)_4 \end{array} \right\} , \left\{ \begin{array}{l} \text{buff-length-2} \\ (r)_5 \end{array} \right\} \right]$ |

Positional Parameter 1:

buff-addr-1

Specifies the symbolic address of the message to be logged and displayed. This may be either the address of a buffer area in main storage containing the complete message, or the address of a buffer area in main storage containing the canned message number and any variable characters to be inserted.

If a canned message is specified, the buffer must be at least four bytes long. (See Figure 10—1.) The first character in the canned message buffer must be a dollar sign (\$). Do not use a dollar sign as the beginning character of any other type of message.

If the message to be displayed is a canned message with a reply but positional parameters 5 and 6 are omitted, the reply will overlay this buffer area for the number of bytes specified in positional parameter 2.

(1)

Indicates that register 1 has been preloaded with the address of the message buffer area.

Positional Parameter 2:**msg-length**

Specifies the length in bytes of the message to be logged and displayed. For canned messages, this specifies the length of the completed message including any inserted variable characters. If REPLY is specified in positional parameter 4 but positional parameter 5 and 6 are omitted, this is the length of the reply. Maximum length for the completed message is 120 bytes. If an operator reply is requested, maximum length is 60 bytes. Similar to the OPR macro instruction, a minimum of 60 characters is displayed and logged when a canned message is specified.

(0)

Indicates that register 0 has been preloaded with the length of the message buffer area or the length of a canned message reply.

If omitted, a length of 60 bytes is assumed.

Positional Parameter 3:**error-addr**

Specifies the symbolic address of an error routine that receives control if an error occurs.

(r)₃

Specifies that the designated register (other than 0 or 1) has been preloaded with the address of the error routine.

If omitted, the requesting task is abnormally terminated if an error occurs.

Positional Parameter 4:**REPLY**

Specifies that a reply is required from the operator. Program control is not returned to the problem program until the operator's reply is received, written to the log, and available in the appropriate buffer area. The message text of the reply is stored beginning at the first byte of the buffer area specified in positional parameter 5 for the length specified in positional parameter 6. If parameter 5 is omitted, then the buffer area specified in positional parameter 1 is overlaid for the length specified in positional parameter 2.

The maximum length of a reply is limited to 60 bytes or to the length of the message buffer, whichever is smaller. Replies that exceed the length of the message buffer area are truncated. If the reply is shorter than the message buffer area, the remaining positions in the buffer area are space filled.

After the reply is received, the message and the reply are written to the system log file.

If omitted, the message is logged and displayed and no reply is expected.

Positional Parameter 5:

buff-addr-2

Specifies the symbolic address of a buffer area in main storage that is to receive a reply from the operator.

This parameter gives the caller the option of specifying an output buffer that will not be destroyed by an incoming reply.

If REPLY was not specified in positional parameter 4, this field is ignored.

(r)₄

Specifies that the designated register (other than 0 or 1) has been preloaded with the address of the buffer area in main storage that is to receive a reply from the operator.

If omitted and REPLY was specified in positional parameter 4, any reply will overlay the buffer area specified in positional parameter 1 for the length specified in positional parameter 2.

Positional Parameter 6:

buff-length-2

Specifies the length in bytes of the buffer area specified in positional parameter 5. Length may be from 1 to 60 bytes.

This parameter must be present if positional parameter 5 was specified.

(r)₅

Specifies that the designated register (other than 0 or 1) has been preloaded with the length of the buffer area specified in positional parameter 5.

If omitted and positional parameter 5 was specified, the macro instruction does not execute.

In the following example we see how you might use the WTLD macro instruction.

Example:

| 1 | LABEL | Δ OPERATION Δ | OPERAND | Δ | COMMENTS |
|-----|---------|---------------|--|---|----------|
| | | 10 | 16 | | |
| 1. | | WTLD | BUFOUT4,15 | | |
| 2. | | . | | | |
| 3. | | . | | | |
| 4. | | . | | | |
| 5. | | WTLD | BUFOUT5,50 | | |
| 6. | | . | | | |
| 7. | | . | | | |
| 8. | | . | | | |
| 9. | | WTLD | BUFOUT6,24,REPLY | | |
| 10. | | . | | | |
| 11. | | . | | | |
| 12. | | . | | | |
| 13. | | MVC | DWGOUT(7),DWGNO | | |
| 14. | | WTLD | BUFOUT7,47,REPLY,BUFINI,39 | | |
| 15. | | . | | | |
| 16. | | . | | | |
| 17. | | . | | | |
| 18. | BUFOUT4 | DC | C'NO ERRORS FOUND' | | |
| 19. | BUFOUT5 | DC | C'MOUNT FORM CK-1113 ON DEV 002 LPT 6 BAND 48-BUS' | | |
| 20. | BUFOUT6 | DC | C'IS PRINTER READY? Y OR N' | | |
| 21. | BUFOUT7 | DC | C'ENTER REPLACEMENT NUMBERS FOR ASSLY DWG' | | |
| 22. | DWGOUT | DC | CL7' | | |
| 23. | BUFINI | DC | CL39' | | |

For simplicity, assume all the messages are from main storage, and no error addresses are specified. Line 1 of the example displays and logs a 15-byte message from the buffer area BUFOUT4. The message is defined in line 18. Because parameters 4, 5, and 6 are omitted, no operator reply is expected.

Line 5 displays and logs a 50-byte message from the buffer area BUFOUT5 and instructs the operator to mount a special printed form on a printer. The message is defined in line 19. Again, because parameters 4, 5, and 6 are omitted, no operator reply is expected.

Line 9 displays and logs a 24-byte message from the buffer area BUFOUT6 and requests a reply of Y or N. The message is defined in line 20. Because no input buffer is specified (parameters 5 and 6), the reply will appear on the screen and in the first byte of the BUFOUT6 buffer area.

Line 14 displays and logs a 47-byte message from the buffer area BUFOUT7 and requests a reply. The message is defined in lines 20 and 21 with the actual drawing number displayed having been moved to the output area by line 13. A 39-byte input buffer for the reply is defined in line 23.

10.2.3. Get a Canned Message (GETMSG)

This macro instruction operates in a manner similar to the WTL macro instruction, except that its use is limited to canned messages and there is no display or log capability. However, after bringing a canned message into main storage with the GETMSG macro instruction, and perhaps making some modification, you can still log or display the message using a WTL, WTL, or OPR macro instruction.

Function:

The GETMSG macro instruction retrieves a message of variable length from the system canned message file, inserts the variables if any are furnished, and stores the completed message in the specified buffer area in main storage. This receiving buffer area is specified either in positional parameter 5 or 1 and must be large enough to contain the completed message text which can be from 1 to 120 characters in length. The format of the canned message buffer is shown in Figure 10-1. The insertion of variable characters is illustrated in Figure 10-2.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|--|
| [symbol] | GETMSG | $\left\{ \begin{array}{c} \text{buff-addr-1} \\ (1) \end{array} \right\} \left[\cdot \left\{ \begin{array}{c} \text{msg-length} \\ (0) \\ 60 \end{array} \right\} \right] \left[\cdot \left\{ \begin{array}{c} \text{error-addr} \\ (r)_3 \end{array} \right\} \right]$ $[\cdot] \left[\cdot \left\{ \begin{array}{c} \text{buff-addr-2} \\ (r)_4 \end{array} \right\} \cdot \left\{ \begin{array}{c} \text{buff-length-2} \\ (r)_5 \end{array} \right\} \right]$ |

Positional Parameter 1:

buff-addr-1

Specifies the symbolic address of a buffer area in main storage containing the number of the canned message to be retrieved and any variable characters to be inserted into the message.

The first character in the canned message buffer must be a dollar sign (\$). Do not use a dollar sign as the beginning character of any other type of message.

If positional parameter 5 is blank, the retrieved message will overlay this area for the length specified in positional parameter 2.

(1)

Indicates that register 1 has been preloaded with the address of the message buffer area.

Positional Parameter 2:**msg-length**

Specifies the length in bytes of the message to be retrieved from the canned message file. Length may be from 1 to 120 bytes.

(0)

Indicates that register 0 has been preloaded with the length of the buffer area.

If omitted, a length of 60 bytes is assumed.

Positional Parameter 3:**error-addr**

Specifies the symbolic address of an error routine that receives control if an error occurs.

(r)₃

Specifies that the designated register (other than 0 or 1) has been preloaded with the address of the error routine.

If omitted, the requesting task is abnormally terminated if an error occurs.

Positional Parameter 4:

This parameter is not applicable, but a comma must be entered in this position.

Positional Parameter 5:**buff-addr-2**

Specifies the symbolic address of a buffer area in main storage that is to receive the retrieved message from the canned message file.

This parameter gives the caller the option of specifying another buffer that will not destroy the original.

(r)₄

Specifies that the designated register (other than 0 or 1) has been preloaded with the address of the buffer area in main storage that is to receive the retrieved message.

If omitted, the retrieved message will overlay the buffer area specified in positional parameter 1 for the length specified in positional parameter 2.

Positional Parameter 6:**buff-length-2**

Specifies the length in bytes of the buffer area specified in positional parameter 5. Length may be from 1 to 120 bytes.

This parameter must be present if positional parameter 5 was specified.

- (r)₅ Specifies that the designated register (other than 0 or 1) has been preloaded with the length of the buffer area specified in positional parameter 5.

If omitted and positional parameter 5 was specified, the macro instruction does not execute.

As an example of how to use the GETMSG macro instruction, suppose you wish to get a canned message and move it to an output area to be printed. This is shown in the following illustration using the same canned message (message 90) that we used earlier for one of the WTL macro instructions.

Example:

| 1 | LABEL | ΔOPERATIONΔ 10 16 | OPERAND | Δ |
|-----|----------|----------------------|------------------------|---|
| 1. | | GETMSG | BUFDOUT8, 45 | |
| 2. | | MVC | PRINT1(45), BUFDOUT8 | |
| 3. | | . | | |
| 4. | | . | | |
| 5. | | . | | |
| 6. | BUFDOUT8 | DC | C' \$ ' | |
| 7. | | DC | XL2' 910' | |
| 8. | | DC | C' / ' | |
| 9. | | DC | C' BRASSCASTING | |
| 10. | | DC | X' 08' | |
| 11. | | DC | CL28' ' | |
| 12. | | . | | |
| 13. | | . | | |
| 14. | | . | | |
| 15. | | GETMSG | BUFIN2, , , BUFIN2, 45 | |
| 16. | | MVC | PRINT2(45), BUFIN2 | |
| 17. | | . | | |
| 18. | | . | | |
| 19. | | . | | |
| 20. | BUFDOUT9 | DC | C' \$ ' | |
| 21. | | DC | XL2' 910' | |
| 22. | | DC | C' / ' | |
| 23. | | DC | C' BRASSCASTING' | |
| 24. | | DC | X' 08' | |
| 25. | BUFIN2 | DC | CL45' ' | |

Line 1 of the example refers to the buffer area (BUFOUT8 which specifies the canned message number 90. Because a second buffer is not specified (parameters 5 and 6 are omitted), the macro instruction retrieves message 90 from the canned message file, inserts the variable characters BRASS CASTING, and stores the completed 45-byte message in main storage starting at the first byte of BUFOUT8. Lines 5 to 11 define the 45-byte output buffer BUFOUT8. Line 7 defines the canned message number 90, line 9 defines the insert characters BRASS CASTING, and line 11 defines an additional 28 bytes to increase the size of the buffer to 45 bytes to accommodate the completed canned message. For our example, line 2 moves the completed message in BUFOUT8 to a work area (PRINT1) where it can be edited and printed.

As you can see, when used this way the GETMSG macro instruction overwrites the buffer specifying the canned message number and insert characters. If you want to avoid this, then you specify a second buffer area. Line 15 does this by specifying BUFIN2 as parameter 5 with a length of 45 bytes (parameter 6). The macro instruction stores the completed message in this buffer area and line 3 moves it from BUFIN2 to a work area PRINT2. Lines 20 to 24 define a 17-byte output buffer BUFOUT9. Line 21 defines the canned message number 90, and line 23 defines the insert characters BRASS CASTING. Line 25 defines the second buffer area BUFIN2 which receives the completed message.

10.3. USER-OPERATOR COMMUNICATION

10.3.1. General

The operating system communicates with the system operator via WTLD macro instructions within the modules of the supervisor and other system elements. When the message is displayed at the system console, it is automatically prefixed with a job identification (ID) number and a message increment number. A message to the console operator can be for:

- Information

This type of message is issued when information is passed to the operator for his information and for inclusion in the system log, as, for example, notification of normal job termination.

- Reply

This type of message is issued when the operating system reaches a point in its processing where a reply to a question (perhaps a choice between alternate courses of action) must be made by the operator before processing can continue. For example, the operator may be asked to decide whether to retry an error recovery procedure or to abort the user program.

The job/message ID of this type of message is followed by a question mark (?). A question is not deleted from the screen until it has been answered.

■ Action

This type of message is used when operator intervention and assistance are required before processing of the requesting task can continue. For example, the operator may be requested to mount a disk pack or turn on power to a device.

The job/message ID of this type of message is followed by an asterisk (*). An action message is not deleted from the screen until the operator has complied with the request and reactivated the job with a GO command.

Normally, the console operator replies to a message from the system. However, he can also communicate with the system without any prompting or direction. This type of communication is called an "unsolicited message" from the operator. If the operator enters an unsolicited message for a job, control is passed to the job step's operator communication island code, which is a routine you must write to handle a specific event. If there is no island code for this job step, or the island code is busy, the unsolicited message is ignored. This is described in 8.6 under the headings relating to operator communication. The operator/system console communications procedure is described in the appropriate operations handbook for operators, UP-8072 (current version).

The action type message is reserved for use by the operating system. However, you can use the WTLD macro instruction to display and log a message for operator information or to request a reply from the operator.

You can also display a message to the operator without making an entry in the system log. In this case, you would use the OPR macro instruction.

10.3.2. Display a Message to the Operator (OPR)

This macro instruction operates in a manner similar to the WTLD macro instruction except that the message is only displayed and not written to the system log. In this way you would keep the size of the log file and subsequent printout to a minimum. You would use the WTLD macro instruction to display messages that require an entry in the log, and use the OPR macro instruction to display messages to the operator that you feel do not require a log entry. However, if the system has a communications output printer (COP) at the console and you use an OPR macro instruction to display a message to the operator, this message will also be printed on the COP. Also, the OPR message, and any reply, will be written to the console log if one was configured at system generation.

Function:

The OPR macro instruction displays a message on the system console for operator reply or information. The message may be either currently in main storage or retrieved from the canned message file. If you specify a canned message, the macro instruction routine inserts any user-supplied variables into the message before the visual display. The format of the canned message buffer is shown in Figure 10-1. The insertion of variable characters is illustrated in Figure 10-2.

Use this macro instruction for console communication with the operator. Upon execution, program control is released until either the message is displayed, the reply is transferred to the appropriate buffer, or an error is encountered.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|---|
| [symbol] | OPR | $\left\{ \begin{array}{c} \text{buff-addr-1} \\ (1) \end{array} \right\}, \left[\begin{array}{c} \text{msg-length} \\ (0) \\ 60 \end{array} \right], \left[\begin{array}{c} \text{error-addr} \\ (r)_3 \end{array} \right]$ $[,REPLY], \left[\begin{array}{c} \text{buff-addr-2} \\ (r)_4 \end{array} \right], \left[\begin{array}{c} \text{buff-length-2} \\ (r)_5 \\ 60 \end{array} \right]$ |

Positional Parameter 1:

buff-addr-1

Specifies the symbolic address of the message to be displayed. This may be either the address of a buffer area in main storage containing the complete message or the address of a buffer area in main storage containing the canned message number and any variable characters to be inserted.

If a canned message is specified, the buffer must be at least four bytes long. (See Figure 10—1.) The first character in the canned message buffer must be a dollar sign (\$). Do not use a dollar sign as the beginning character of any other type of message.

If the message to be displayed is a canned message with a reply, but positional parameters 5 and 6 are omitted, the reply will overlay this buffer area for the number of bytes specified in positional parameter 2.

(1)

Indicates that register 1 has been preloaded with the address of the message buffer area.

Positional Parameter 2:

msg-length

Specifies the length in bytes of the message to be displayed. For canned messages, this specifies the length of the completed message including any inserted variable characters. If REPLY is specified in positional parameter 4, but positional parameters 5 and 6 are omitted, this is the length of the reply.

Maximum length is 60 bytes.

(0)

Indicates that register 0 has been preloaded with the length of the message buffer area or the length of a canned message reply.

If omitted, a length of 60 bytes is assumed.

Positional Parameter 3:**error-addr**

Specifies the symbolic address of an error routine that receives control if an error occurs.

(r)₃

Indicates that the designated register (other than 0 or 1) has been preloaded with the address of the error routine.

If omitted, the requesting task is abnormally terminated if an error occurs.

Positional Parameter 4:**REPLY**

Specifies that a reply is required from the operator. Program control is not returned to the problem program until the operator's reply is received and available in the appropriate buffer area. The first nonblank character of the message text of the reply is stored, beginning at the first byte of the buffer area specified in positional parameter 5 for the length specified in positional parameter 6. If parameter 5 is omitted, then the buffer area specified in positional parameter 1 is overlaid for the length specified in positional parameter 2.

After the reply is received, register 0 contains the number of characters typed by the operator, including the character under the cursor.

The maximum length of a reply is limited to 60 bytes or to the length of the message buffer, whichever is smaller. Replies that exceed the length of the message buffer area are truncated. If the reply is shorter than the message buffer area, the remaining positions in the buffer area are space filled. If the reply is all spaces, the buffer will be space filled.

If omitted, the message will be displayed and no reply expected.

Positional Parameter 5:**buff-addr-2**

Specifies the symbolic address of a buffer area in main storage that is to receive a reply from the operator.

This parameter gives the caller the option of specifying an output buffer that will not be destroyed by an incoming reply.

If REPLY was not specified in positional parameter 4, this field is ignored.

(r)₄

Specifies that the designated register (other than 0 or 1) has been preloaded with the address of the buffer area in main storage that is to receive a reply from the operator.

If omitted and REPLY was specified in positional parameter 4, any reply will overlay the buffer area specified in positional parameter 1 for the length specified in positional parameter 2.

Positional Parameter 6:

buff-length-2

Specifies the length in bytes of the buffer area specified in positional parameter 5. Length may be from 1 to 60 bytes.

(r)₅

Specifies that the designated register (other than 0 or 1) has been preloaded with the length of the buffer area specified in positional parameter 5.

If omitted and positional parameter 5 was specified, a length of 60 bytes is assumed.

There are three ways in which the OPR macro instruction differs from the WTLD macro instruction. These are:

1. You cannot write to the system log; you can only display a message.
2. The length of the message to be displayed cannot exceed 60 bytes.
3. If you set up a second buffer to receive the operator reply (parameter 5), but omit the buffer length (parameter 6), the macro instruction assumes a length of 60 bytes. (With the WTLD macro instruction, if the second buffer is specified (parameter 5), the length (parameter 6) must also be specified, else the macro instruction does not execute.

Following is an example of how to use the OPR macro instruction. In this case we've used the same parameters as for the WTLD macro instruction example.

Example:

| 1 | LABEL | Δ OPERATION Δ | OPERAND | Δ | COMMENTS |
|-----|---------|---------------|--|---|----------|
| | | 10 | 16 | | |
| 1. | | OPR | BUFOUT4, 15 | | |
| 2. | | . | | | |
| 3. | | . | | | |
| 4. | | . | | | |
| 5. | | OPR | BUFOUT5, 24, REPLY | | |
| 6. | | . | | | |
| 7. | | . | | | |
| 8. | | . | | | |
| 9. | | MVC | DWGOUT(7), DWGNO | | |
| 10. | | OPR | BUFOUT6, 47, REPLY, BUF.IN1, 40 | | |
| 11. | | . | | | |
| 12. | | . | | | |
| 13. | | . | | | |
| 14. | BUFOUT4 | DC | C'NO ERRORS FOUND' | | |
| 15. | BUFOUT5 | DC | C'IS PRINTER READY? Y OR N' | | |
| 16. | BUFOUT6 | DC | C'ENTER REPLACEMENT NUMBERS FOR ASSLY DWG' | | |
| 17. | DWGOUT | DC | CLT' ' | | |
| 18. | BUF.IN1 | DC | CL40' ' | | |

Assume that all the messages are from main storage, and also assume that no error addresses are specified. Line 1 of the example displays a 15-byte message from the buffer area BUFOUT4. The message is defined in line 14. Because parameters 4, 5, and 6 are omitted, no operator reply is expected. Line 5 displays a 24-byte message from the buffer area BUFOUT5 and requests a reply of Y or N. The message is defined in line 15. Because no input buffer is specified (parameters 5 and 6), the reply will appear on the screen and in the first byte of the BUFOUT5 buffer area.

Line 10 displays a 47-byte message from the buffer area BUFOUT6 and requests a reply. The message is defined in lines 16 and 17 with the actual drawing number displayed having been moved to the output area by line 9. A 40-byte input buffer for the reply is defined in line 18.

11. Other Services

11.1. SPOOLING

11.1.1. General

Spooling is the technique of buffering data files for low speed input and output devices to a high speed storage device independently of the program that uses the input data or generates the output data. Data from card readers or from remote sites is stored on disk for subsequent use by the intended program. Data output by the program is stored on disk for subsequent punching or printing. The spooling function also handles diskette files. It treats input from diskette as though it were from a card reader, and output to a diskette as though it were to a card punch. In this description of spooling, any reference to a card reader, card input, or card file also includes diskette input; any reference to a card punch, card output, or card file also includes diskette output. The data management user guide, UP-8068 (current version) shows the formats for diskette records.

Spooling enhances system performance by releasing large production programs and system software from the constraint of the slower speed devices, thereby freeing the main storage occupied by these programs sooner; and by driving the slower speed devices at their rated speed on a continuous basis, thereby making full use of the devices during the time that is normally lost to systems overhead or to job steps not using printers.

The spooling function comprises five elements: initialization, input reader, spooler, output writer, and special functions. These elements are described on the following pages. Figure 11—1 gives a simplified picture of the relationship between the slow and high speed input/output devices and the software components of the spooling function and the supervisor.

11.1.1.1. Initialization

Spool initialization provides for the establishment, data recovery, or reestablishment of the spoolfile at supervisor initialization. Based on system generation parameters or operator specified options at supervisor initialization, it allocates the spoolfile and builds the system spool control table, or it recovers an existing spoolfile. In the case of an existing spoolfile, it clears the file, recovers closed subfiles, or recovers and closes all subfiles.

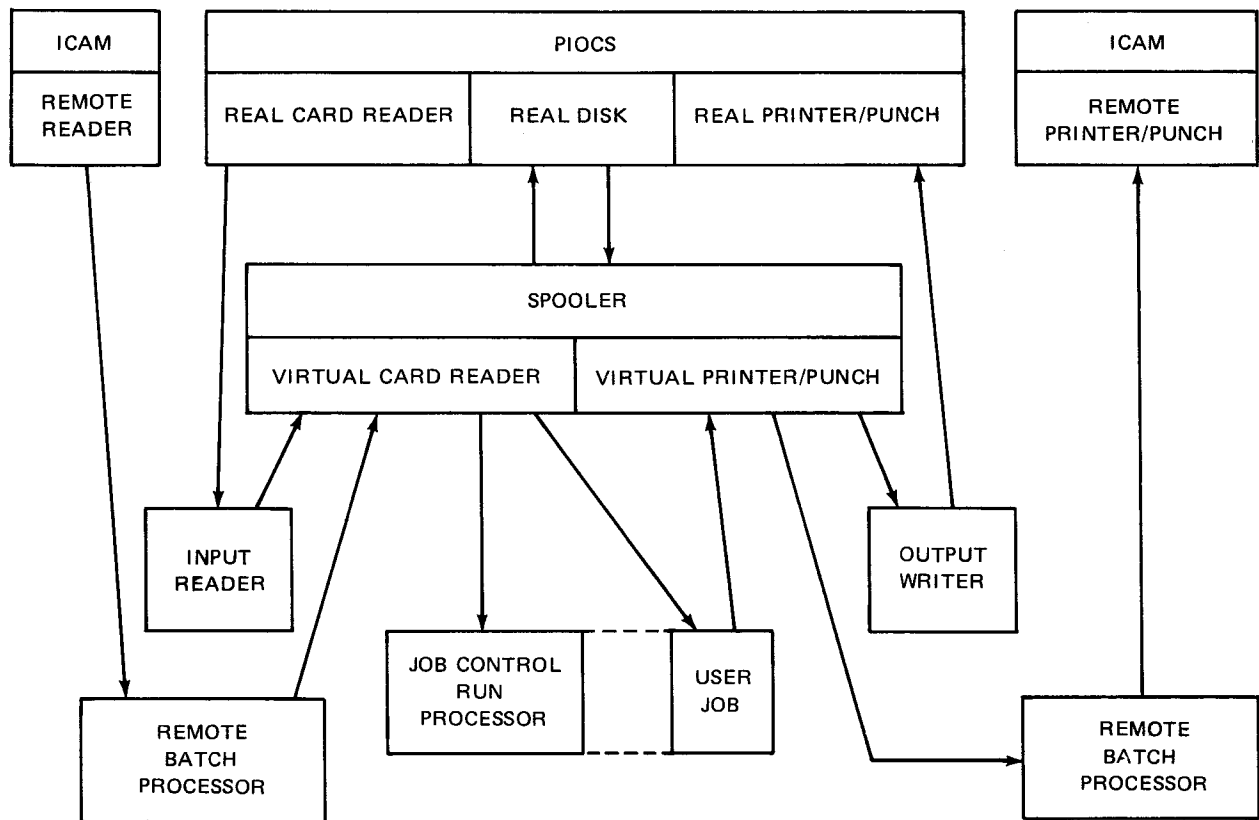


Figure 11—1. Relationship of Spooling Devices and Programs

11.1.1.2. Input Reader

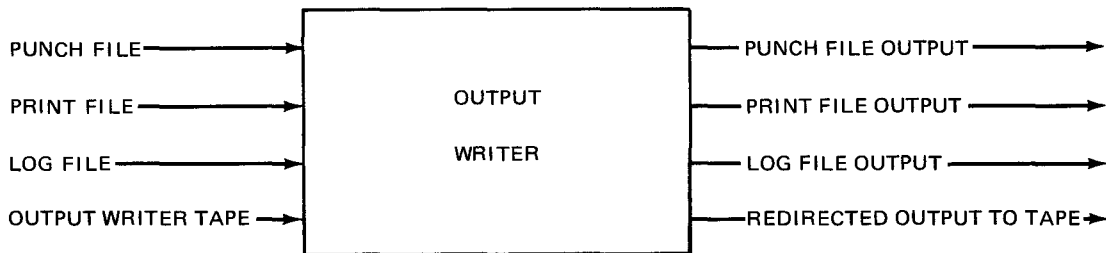
➔ Using physical IOCS the input reader reads cards from a real card reader or records from a diskette and writes these images to the spoolfile via a virtual card reader and the spooler. It closes the previous subfile if one exists and opens a new subfile. A given input reader can handle only one card reader or diskette at a time; however, any number of input readers can be active.

11.1.1.3. Spooler

The spooler is the hub of the spooling package and is linked as part of the resident supervisor. It provides record level input and output to and from the spoolfile for each element in the system needing access to that file. It intercepts all input/output commands to virtual printer, punch, and card reader devices, and accesses the disc when necessary using the system access technique (SAT) for accesses to the spoolfile. All input/output requests (EXCP macro instructions) addressing virtual devices are trapped and routed to the spooler for processing rather than physical IOCS. The spooler supports both reads and writes to virtual devices while simulating the action of physical IOCS as far as error handling, page spacing, and synchronization are concerned. It allocates tracks to subfiles and maintains control of the user's spool control tables. It can handle any number of print, punch, and read files simultaneously, including multiple files per job.

11.1.1.4. Output Writer

The output writer reads data from the system spoolfile and prints or punches this data on the physical devices. As an alternative, it can output this data to a tape so that the tape may be reintroduced at a later time to the output writer as input, rather than using the spool file as input, or for processing at a later time by the user.



The output writer configures itself dynamically to the printer or punch assigned, thereby keeping main storage requirements to a minimum. It is loaded automatically whenever there are files to be printed or punched and there is a printer or punch available. As with the input reader, a copy of the output writer can handle only one printer or punch. However, for every printer or punch on a system, there can be a version of this element running that device.

A number of capabilities and options are available:

- Processing may be handled in either burst or nonburst mode.
- The operator may refine burst mode by selecting a subcriteria.
- A maximum of 255 copies of a given file may be printed or punched.
- Subfiles may be retained after they have been printed or punched.
- Printer or punch output may be initially assigned, or redirected, to tape.

The output writer determines which file to process based on criteria entered at system generation, or later by an operator system command or function to the output writer by the operator. For example, let us assume nonburst was specified at system generation. This means an output subfile cannot be printed or punched until after the job has terminated and the job log has been closed. Also, each job's output is handled as a continuous entity. The operator can change this to burst mode processing, which means that an output subfile can be printed or punched after it has been closed, or after a breakpoint has been created (see 11.1.3), and does not have to wait until the job has terminated. He can specify file selection by various criteria such as first-in/first-out by device type, account number, job number, etc. Operator commands and responses are described in the appropriate operations handbook for your system.

If a system is generated with the block numbering capability, the output writer will always create output tapes with block numbers. When the output tape is reintroduced as input, the input commands will be issued with the assumption that there is a block number. This means that tapes created with block numbers by the output writer cannot be reintroduced into a system without block numbering. Also, tapes created by the output writer on a system that does not have the block numbering capability, cannot be reintroduced into a system that does have block numbering. If an operator attempts to print or punch from a spool tape that is not compatible with the system, the output writer will be terminated and an INPUT SPOOL TAPE INVALID message will be displayed.

11.1.1.5. Special Functions

There are a number of special functions, such as open, close, find, delete, that can be used by symbionts accessing the spoolfile but are not available for use by user programs. However, the breakpoint function is available to user programs and to the operator. A breakpoint is the closing and reopening of a spool subfile to permit output to the physical device to start before the job step terminates. For example, if a spool subfile is getting full, a message to the operator notifies him of this so that he can create a breakpoint to the output file. The user program can also create a breakpoint by using the BRKPT macro instruction (described in 11.1.3).

11.1.2. To Use Spooling

At system generation, you can select:

- no spooling;
- output spooling only;
- input/output spooling; or
- ■ input/output and remote batch spooling.

Also, you can specify first-in/first-out processing in the nonburst mode, or accept the burst mode, which is the default condition. This can be changed later or qualified by the operator.

Statements input to job control enable it to set up the files, buffers, linkages, and control tables by which the spooling functions are performed. If the system does not have the spooling function, these job control statements are ignored.

Job control options for spooling are entered using the JOB, SPL, DATA, and DST job control statements. These are described in the job control user guide, UP-8065 (current version). Initialization options are also entered by the system operator. These are described in the appropriate operations handbook for your system.

There are no changes required to a user program to use spooling. You can define your files using either data management macro instructions, or physical IOCS macro instructions. A job that runs on a nonspooling system will also run on a spooling system, and vice versa. If you use the BRKPT macro instruction in your program, it will be ignored if your job is run on a nonspooling system.

11.1.3. Create a Breakpoint in a Spool Output File (BRKPT)

Function:

The BRKPT macro instruction creates a breakpoint in a printer or punch spoolfile. It closes and reopens the subfile as it is being generated by the spooler. Each segment created at this breakpoint is considered a logical subfile so that output to the physical device can be started prior to job step termination.

If this macro instruction is included in a program executing in a system that does not have the spooling capability, the macro instruction is ignored.

Format:

| LABEL | △ OPERATION △ | OPERAND |
|----------|---------------|------------------------------|
| [symbol] | BRKPT | { filename } { CCB-name } |

Positional Parameter 1:

filename

Specifies the symbolic address of the DTF macro instruction in the program which defines the file in which a breakpoint is to be created. Use this parameter if you are using data management macro instructions to define and access the file.

CCB-name

Specifies the symbolic address of the command control block (CCB) associated with the file in which the breakpoint is to be created. Use this parameter if you are using physical IOCS macro instructions to define and access the file.

11.2. JOB ACCOUNTING

11.2.1. General

The job accounting package consists of resident routines which are linked with the supervisor and elements of the job step processor at system generation time. These routines provide a count of the facilities utilized by each job step during its execution within the system. The message logging facility of the spooling function transfers this data from main storage to disk as part of the output spoolfile. The output writer prints the job step and job values as part of the normal message log output for each job. Optionally, the output writer can write the accounting information to a standard SAM magnetic tape file for offline processing by user-developed accounting routines or by OS/3 data utility routines. You can assign an account number using the JOB job control statement which is carried along with the accounting records. This enables you to accumulate statistics from the SAM file for computer time and resources charged against an account number, which could represent a project, department, cost center, etc. The job accounting function requires the use of the spooling package and the optional timer facilities. These must be included at system generation time. Also, the job accounting versions of SVC decode and the switcher must be included within the supervisor at link edit time.

11.2.2. Accounting Data

Accounting data is accumulated in a job accounting table (Figure 11—2) in the job prologue. Fields in this table serve as counters for job step and job statistics.

| Byte | 0 | 1 | 2 | 3 |
|------|--|----------------------------|------------------------------|---|
| 0 | count of SVCs in job step | | | |
| 4 | count of SVCs in job | | | |
| 8 | count of transient calls in job step | | | |
| 12 | count of transient calls in job | | | |
| 16 | CPU time used by job step | | | |
| 20 | CPU time used by job | | | |
| 24 | length of largest job step (in bytes) | | | |
| 28 | time of day that job step started | | | |
| 32 | time of day that job started | | | |
| 36 | accumulated time of day of all job steps | | | |
| 40 | count of EXCPs in job | | | |
| 44 | count of I/Os not fitting in device count table | | | |
| 48 | switch priority | not used | termination code of job step | |
| 52 | PUB acctg ID | count of EXCPs to that PUB | | |
| | (device count table — one entry for each device) | | | |
| | PUB acctg ID | count of EXCPs to that PUB | | |

Figure 11—2. Job Accounting Table Format

11.2.2.1. Job Step Level Data

Counters in the job accounting table are dynamically incremented during job step execution. The following data is collected for each job step:

- Central processor time

This consists of the total time in milliseconds charged to tasks of this job, or supervisor tasks working for this job. This means that all supervisor overhead, such as processing SVCs and the processing of supervisor tasks is charged to the requesting job. Supervisor idle (wait) time is not charged to any job.

- Total SVCs executed

This consists of the total number of SVCs executed by the job's tasks or by supervisor tasks working in behalf of the job.

- Total transient functions

This consists of the total number of transient functions executed by the job's tasks or by supervisor tasks working in behalf of the job. This does not include overlays to transients.

- Total I/O requests

This consists of the total number of I/O requests executed for each device by the job's tasks or by supervisor tasks working in behalf of the job. I/O requests per device include spooling activity in terms of the number of cards read from the spool file and print lines written to the spool file by this job step.

In addition to the counts dynamically maintained in the job accounting table, the job step processor furnishes the following values for job step accounting:

- Total wall clock time required for the job step to execute. This does not include time during which the job step was rolled out, nor does it include the period between the time a checkpoint was taken and the job step was restarted from the last checkpoint.

- Total main storage into which programs were loaded by the loader.

This value represents only that amount of main storage used by the job step as recorded by the loader, and does not include the prologue or those available areas within the job region which are used but not for loading.

- Initial switch priority of the job step.

- Termination code of the job step. Normal termination code is 000.

11.2.2.2. Job Level Data

Some of the data collected for the job steps of a particular job is totalled for the job's accounting record. In addition, data is collected on the job level which cannot be acquired by just summing the job step values. That data which is collected solely for the job is recorded at job termination time and consists of the following:

- Size of the largest job step.

- Job date

This is the date from the job preamble representing the date the job was run.

- Total job main storage including prologue.

- Total wall clock time for the job, including all of the job step processor overhead.

Wall clock time is defined as the point in time when a job is initiated to execute up to the point in time when the job termination message is displayed, and does not include spool time.

- Total wall clock time for all job steps.

This is a sum of the total wall clock time for each job step and does not include job control time.

- Total CPU time for all job steps.

This is a sum of the CPU time for each job step and does not include job control time.

- Total SVC count for all job steps.

This is a sum of the SVC counts for each job step and does not include job control counts.

- Total transients called for all job steps.

This is a sum of the transients called by the job steps and does not include job control counts.

- Total I/O count for all job steps.

This is a sum of the I/Os executed by the job steps and does not include job control counts.

11.2.3. Data Printout

When printing the job's log, the output writer also prints the accounting records for that job. Also, the output writer can write all the job log records to a magnetic tape for offline processing, or only the log records, or the accounting records. This gives you the ability to create a system log file and a system accounting file for subsequent statistical processing and evaluation. Figure 11—3 shows the format of the job accounting record printout.



Index

| Term | Reference | Page | Term | Reference | Page |
|----------------------------------|---------------------|-------|--|-----------|------|
| A | | | | | |
| Abnormal termination | | | description | 3.3 | 3—5 |
| description | 8.3.2 | 8—13 | Fig. 3—1 | | 3—6 |
| dumps | 9.1.3 | 9—10 | label field | 3.3.1 | 3—6 |
| Abnormal termination island code | | | operand field | 3.3.3 | 3—7 |
| attaching | 8.6.1.1 | 8—35 | operation field | 3.3.2 | 3—7 |
| description | 8.6.6 | 8—13 | sequence field | 3.3.6 | 3—8 |
| example using symbolic addresses | Fig. 8—6 | 8—44 | ATTACH macro instruction | | |
| exiting | 8.6.4.2 | 8—40 | function | 7.3.2 | 7—9 |
| multitasking | 8.6.9.2 | 8—51 | multitasking | 7.3 | 7—5 |
| Absolute address (ABS) | 9.3.4.1.3 | 9—34 | task creation | 7.2.2 | 7—3 |
| Accounting | See job accounting. | | Automatic volume recognition | | |
| Action messages | 10.3.1 | 10—18 | description | 2.2.10 | 2—8 |
| Actions, monitor statements | | | interrupt module function | 2.2.6 | 2—4 |
| description | 9.3.5 | 9—38 | AWAKE macro instruction | | |
| display (D) | 9.3.5.1 | 9—38 | function | 7.3.5 | 7—12 |
| halt (H) | 9.3.5.2 | 9—43 | multitasking | 7.3 | 7—6 |
| quit (Q) | 9.3.5.3 | 9—44 | queue driven task | 7.2.5 | 7—4 |
| summary | Table 9—2 | 9—45 | B | | |
| Activate waiting table (POST) | 7.4.4 | 7—17 | Base displacement address (B/D) | 9.3.4.1.2 | 9—34 |
| Address adjustment module | 2.2.2.4 | 2—4 | BCW | | |
| ALLOC macro instruction | | | function | 4.2.1 | 4—2 |
| disk | 5.3.1 | 5—5 | 4.2.3 | | 4—5 |
| diskette | 5.5.1 | 5.14 | format for integrated disk adapter | Fig. 4—2 | 4—7 |
| Allocate routine, disk | 5.2.1 | 5—2 | format for integrated peripheral channel | Fig. 4—3 | 4—10 |
| ALTER statement | 9.3.1.2 | 9—25 | format for multiplexer channel | Fig. 4—4 | 4—13 |
| ARGLST macro instruction | 8.5.5 | 8—30 | Block addressing | | |
| Assembler coding form | | | by key | 8.2.2 | 8—3 |
| comments field | 3.3.4 | 3—7 | by relative block number | 8.2.3 | 8—4 |
| continuation column | 3.3.5 | 3—7 | Block level device handler | 6.1 | 6—1 |
| | | | Block loader | 8.2.1 | 8—2 |
| | | | Block modules | 8.2 | 8—2 |
| | | | Block number processing, TSAT | | |
| | | | description | 6.10 | 6—56 |
| | | | facilities required | 6.10.1 | 6—57 |

| Term | Reference | Page | Term | Reference | Page |
|--------------------------------------|---------------------|------|-------------------------------------|---------------------|------|
| Block number processing, TSAT (cont) | | | Canned message file | | |
| initialized | 6.10.2.1 | 6—58 | buffer formats | Fig. 10—1 | 10—4 |
| noninitialized | 6.10.2.2 | 6—58 | description | 10.1.1 | 10—3 |
| Block numbered tape files | | | inserting variable characters | 10.1.1.2 | 10—3 |
| block number field | 4.4.1 | 4—33 | messages | Fig. 10—2 | 10—5 |
| description | Fig. 4—9 | 4—34 | | 10.1.1.1 | 10—3 |
| input/output buffer | 4.4 | 4—33 | CAW | Fig. 4—6 | 4—17 |
| physical IOCS requirements | 4.4.3 | 4—35 | CCB | | |
| processing | 4.4.5 | 4—36 | format | Fig. 4—7 | 4—22 |
| tape restrictions | 4.4.4 | 4—35 | function | 4.2.1 | 4—2 |
| Block numbers, relative | See relative | | | 4.2.5 | 4—18 |
| | block numbers. | | CCW | | |
| Block transfer, wait | 6.4.4 | 6—22 | format for selector channel | Fig. 4—5 | 4—16 |
| | 6.9.4 | 6—54 | function | 4.2.1 | 4—2 |
| Blocks | | | | 4.2.4 | 4—15 |
| accessing multiple | 6.2.6 | 6—8 | Channel address word (CAW) | Fig. 4—6 | 4—17 |
| accessing physical | 6.4.6 | 6—24 | Channel command word (CCW) | See CCW. | |
| logical | See logical blocks. | | Channel interrupt processor modules | 2.2.2.8 | 2—5 |
| output logical | 6.4.3 | 6—21 | Channel program, execute | 4.2.8 | 4—28 |
| retrieve next logical | 6.4.2 | 6—20 | Channel program processor module | 2.2.2.1 | 2—2 |
| wait for transfer | 6.4.4 | 6—22 | Channel scheduler modules | 2.2.2.5 | 2—4 |
| Branch, FETCH macro instruction | 8.2.9 | 8—11 | Channels | | |
| Breakpoint function | 11.1.1.5 | 11—4 | integrated peripheral | See integrated | |
| | 11.1.3 | 11—4 | multiplexer | peripheral channel. | |
| BRKPT macro instruction | 11.1.3 | 11—4 | selector | See multiplexer | |
| Buffer control word (BCW) | See BCW. | | | channel. | |
| Buffer format, earned message | 10.1.1.2 | 10—3 | | See selector | |
| | Fig. 10—1 | 10—4 | CHAP macro instruction | | |
| Buffering data files, spooling | 11.1.1 | 11—1 | function | 7.3.6 | 7—13 |
| Buffers, I/O | 4.4.3 | 4—35 | multitasking | 7.3 | 7—6 |
| | | | Characters, canned messages | 10.1.1.2 | 10—3 |
| | | | | Fig. 10—1 | 10—4 |
| | | | | Fig. 10—2 | 10—5 |
| | | | Checkpoint and restart capability | | |
| | | | description | 9.2 | 9—10 |
| | | | error codes | Table 9—1 | 9—13 |
| | | | generating checkpoint records | 9.2.1 | 9—12 |
| | | | processing PIOCS files | 9.2.4 | 9—18 |
| | | | using magnetic tape as a | | |
| | | | checkpoint file | 9.2.2 | 9—14 |
| | | | using SAT disk as a checkpoint | | |
| | | | file | 9.2.3 | 9—15 |
| | | | Checkpoint dump | 6.8.1 | 6—45 |

C

| Term | Reference | Page | Term | Reference | Page |
|--|--------------------------|------|----------------------------------|-----------------------|------|
| EOF1 and EOF2 labels | See file trailer labels. | | F | | |
| EOJ macro instruction | | | FCB | | |
| function | 8.3.4 | 8—13 | format | Fig. 4—8 | 4—25 |
| general | 8.3 | 8—12 | general | 4.2.1 | 4—2 |
| normal termination | 8.3.1 | 8—13 | location | 4.2.7 | 4—26 |
| EOV1 and EOV2 labels | See file trailer labels. | | Features | 1.2 | 1—2 |
| Error acceptance options | 4.2.5 | 4—18 | FETCH macro instruction | 8.2.9 | 8—11 |
| Error codes | | | File control block (FCB) | See FCB. | |
| checkpoint/restart | Table 9—1 | 9—13 | File header labels | | |
| disk space management | 5.6 | 5—13 | first (HDR1) | 6.6.2.1 | 6—29 |
| program loader | 8.2.5 | 8—5 | Fig. 6—7 | 6—30 | |
| Error control, program and machine | 2.2.7 | 2—7 | Fig. 6—18 | 6—60 | |
| Error control module | 2.2.2.9 | 2—5 | second (HDR2) | 6.6.2.2 | 6—31 |
| Error editing root overlay | 2.2.2.10 | 2—5 | Fig. 6—8 | 6—32 | |
| Error logging | 2.2.15 | 2—11 | Fig. 6—19 | 6—61 | |
| Error message interface, standard system | 2.2.9.4 | 2—8 | File organization | | |
| Error reply overlay | 2.2.2.12 | 2—5 | disk SAT | 6.2 | 6—1 |
| Error status field | 6.4.4 | 6—22 | tape SAT | 6.7 | 6—37 |
| Event control block | | | File termination operations | 6.4.7 | 6—24 |
| format | Fig. 7—1 | 7—8 | File trailer labels | | |
| generating | 7.3.1 | 7—6 | description | 6.6.3 | 6—33 |
| program check | 8.6.5 | 8—40 | EOF1 and EOV1 field descriptions | Table 6—4 | 6—35 |
| Exception branching | 4.3.1 | 4—31 | EOF1 and EOV1 formats | Fig. 6—9 | 6—34 |
| EXCP | 2.2.2.1 | 2—2 | Fig. 6—20 | 6—62 | |
| EXCP macro instruction | 4.2.1 | 4—3 | EOF2 and EOV2 field descriptions | Table 6—5 | 6—37 |
| | 4.2.8 | 4—28 | EOF2 and EOV2 formats | Fig. 6—10 | 6—36 |
| EXCP processor | 2.2.2.1 | 2—2 | Fig. 6—21 | 6—63 | |
| EXEC job control statement | 9.3.1.1 | 9—23 | Filelocks | 6.3.1.1 | 6—12 |
| Execute channel program (EXCP) processor | 2.2.2.1 | 2—2 | Files | | |
| | 4.2.8 | 4—28 | assign space | 5.3.1 | 5—5 |
| EXIT macro instruction | | | 5.3.2 | 5—7 | |
| function | 8.6.4.1 | 8—39 | checkpoint | See checkpoint files. | |
| general | 8.6 | 8—34 | defining new | 6.3.1 | 6—10 |
| EXTEND macro instruction | 5.3.2 | 5—7 | disk SAT | See disk SAT files. | |
| Extend routine | 5.2.2 | 5—3 | processing PIOCS | 9.2.4 | 9—18 |
| | | | renaming | 5.3.4 | 5—10 |
| | | | scratching | 5.2.3 | 5—3 |
| | | | 5.3.3 | 5—9 | |
| | | | spooling | 11.1.1 | 11—1 |
| | | | tape | See tape files. | |
| | | | tape SAT | See tape SAT files. | |
| | | | First file header label | See HDR1 label. | |
| | | | Format illustrations | 3.2 | 3—1 |

| Term | Reference | Page | Term | Reference | Page |
|------------------------------------|------------|-------|---|---------------------------------|-------|
| Format write option | 6.2.6 | 6—8 | Information control | See system information control. | |
| G | | | Information messages | 10.3.1 | 10—17 |
| Generate buffer control word (BCW) | 4.2.3 | 4—5 | Initial space allocation formula | 6.2.4 | 6—4 |
| GET macro instruction | | | Initialized block number processing | 6.10.2.1 | 6—58 |
| disk processing | 6.4.2 | 6—20 | Input format, monitor | 9.3.2 | 9—27 |
| magnetic tape processing | 6.9.2 | 6—52 | Fig. 9—1 | 9—29 | |
| GETCOM macro instruction | 8.7.1 | 8—52 | Input/output buffer | 4.4.3 | 4—35 |
| GETCS macro instruction | 8.8.3 | 8—57 | Input/output control system, physical | See PIOCS. | |
| GETIME macro instruction | | | Input/output synchronization | 4.3 | 4—30 |
| example | Fig. 8—1 | 8—19 | Input reader | 2.2.8 | 2—7 |
| function | 8.4.1.3 | 8—17 | 11.1.1.2 | 11—2 | |
| timer services | 8.4 | 8—15 | Instruction location option (A) | 9.3.4.2 | 9—35 |
| GETINF macro instruction | 8.7.3 | 8—53 | Instruction sequence option (I) | 9.3.4.3 | 9—36 |
| GETMSG macro instruction | | | Integrated disk adapter, BCW format | Fig. 4—2 | 4—7 |
| function | 10.2.3 | 10—14 | Integrated peripheral channel, BCW format | Fig. 4—3 | 4—10 |
| general | 10.1 | 10—2 | | | |
| | Table 10—1 | 10—2 | Interfaces | | |
| H | | | disk SAT files | 6.3 | 6—10 |
| Halt action (H) | 9.3.5.2 | 9—43 | standard system error message | 2.2.9.4 | 2—8 |
| Hardware program check interrupt | 8.6.5 | 8—40 | supervisor | See supervisor interfaces. | |
| HDR1 label | | | tape SAT files | 6.8 | 6—45 |
| description | 6.6.2.1 | 6—29 | Interlacing | | |
| field descriptions | Table 6—2 | 6—31 | accessing | Fig. 6—4 | 6—7 |
| formats | Fig. 6—7 | 6—30 | definition of variables | Fig. 6—3 | 6—6 |
| | Fig. 6—18 | 6—60 | lace factor calculation | 6.2.5.2 | 6—8 |
| HDR2 label | | | operation | 6.2.5.1 | 6—6 |
| description | 6.6.2.2 | 6—31 | record | 6.2.5 | 6—5 |
| field descriptions | Table 6—3 | 6—33 | Interrupt handling | 2.1 | 2—1 |
| formats | Fig. 6—8 | 6—32 | Interrupt levels | 8.6 | 8—34 |
| | Fig. 6—19 | 6—61 | Interrupt module | 2.2.2.6 | 2—4 |
| Header, program phase | | | Interrupt servicing routine | 4.3.1 | 4—31 |
| format | 8.2.8.1 | 8—10 | Interrupts, timer | 8.4.2 | 8—20 |
| locate | 8.2.8 | 8—9 | Interval timer island code | | |
| Hierarchical structure, tasks | 7.2.6 | 7—4 | attaching | 8.6.1.1 | 8—35 |
| I | | | description | 8.6.7 | 8—45 |
| ICAM, main storage consolidation | 2.2.11 | 2—9 | | | |
| Imperative macro instructions | 3.4.2 | 3—8 | | | |

| Term | Reference | Page | Term | Reference | Page |
|------------------------------------|-----------|------|--------------------------------|--------------------------|------|
| Interval timing island code (cont) | | | Job control device assignments | 4.2.6 | 4—24 |
| example | Fig. 8—7 | 8—45 | Job control statement | 9.3.1.2 | 9—26 |
| exiting from | 8.6.4.1 | 8—39 | Job level data | 11.2.2.2 | 11—7 |
| I/O completion, wait | 4.3.1 | 4—31 | Job preamble | 8.6.9.2 | 8—51 |
| I/O scheduler | 4.2.8 | 4—28 | Job prologue | 8.7 | 8—51 |
| I/O status tables (IOST) | See IOST. | | Job step level data | 11.2.2.1 | 11—6 |
| I/O usage requirements | 4.2.2 | 4—4 | | | |
| I/O wait, multiple | 4.3.2 | 4—32 | | | |
| IOST | | | K | | |
| description | 2.2.2.7 | 2—4 | Keys | | |
| interrupt module | 2.2.2.6 | 2—4 | block addressing | 6.2.2 | 6—3 |
| Island code linkage | | | processing blocks | 6.3.3.1 | 6—18 |
| abnormal termination | 8.6.6 | 8—43 | READE/READH macro instructions | 6.4.5 | 6—23 |
| attaching to a task | Fig. 8—6 | 8—44 | | | |
| description | 8.6.1 | 8—35 | L | | |
| detaching from a task | 8.6.1.1 | 8—35 | Label field, coding form | 3.3.1 | 3—6 |
| entrance | 8.6.1.2 | 8—36 | Labels, tape | See tape labels. | |
| exit | 8.6 | 8—34 | Lace factor | | |
| interval timer | 8.6.2 | 8—38 | calculation | 6.2.5.2 | 6—8 |
| multitasking | 8.6.3 | 8—39 | description | 6.2.5 | 6—5 |
| operator communication | 8.6.4 | 8—39 | LBL job control statement | 6.6.1 | 6—27 |
| program check | 8.6.4.1 | 8—39 | LFD job control statement | 6.6.1 | 6—27 |
| user-supplied | 8.6.4.2 | 8—40 | Library search order | 8.2.3 | 8—4 |
| | 8.6.7 | 8—45 | Linkage | | |
| | Fig. 8—7 | 8—45 | island code | See island code linkage. | |
| | 8.6.9 | 8—49 | program | See program linkage. | |
| | 8.6.8 | 8—46 | Linkage procedure | 8.5.2 | 8—26 |
| | Fig. 8—8 | 8—47 | Linkage register conventions | 8.5.1 | 8—25 |
| | Fig. 8—9 | 8—48 | LOAD macro instruction | 8.2.6 | 8—5 |
| | 8.6.5 | 8—40 | Loader, program | See program loader. | |
| | Fig. 8—4 | 8—41 | Loader error processing | 8.2.5 | 8—5 |
| | Fig. 8—5 | 8—42 | LOADI macro instruction | 8.2.8 | 8—9 |
| | 2.2.7 | 2—7 | LOADR macro instruction | 8.2.7 | 8—7 |
| | | | Lockable file | 6.9.1 | 6—51 |
| J | | | | | |
| Job | | | | | |
| cancel | 8.3.5 | 8—14 | | | |
| definition | 1.1 | 1—1 | | | |
| end-of-job step | 8.3.4 | 8—13 | | | |
| Job accounting | | | | | |
| data | 11.2.2 | 11—6 | | | |
| data printout | 11.2.3 | 11—8 | | | |
| description | 11.2.1 | 11—5 | | | |
| job level data | 11.2.2.2 | 11—7 | | | |
| job step level data | 11.2.2.1 | 11—6 | | | |
| record printout format | Fig. 11—3 | 11—9 | | | |
| table format | Fig. 11—2 | 11—6 | | | |

| Term | Reference | Page | Term | Reference | Page |
|-----------------------------|-------------------|--------------|---|----------------------|--------------|
| Output, logical block (PUT) | 6.4.3 6.9.3 | 6—21 6—53 | Preemptive scheduling priority | 2.2.12 | 2—9 |
| Output writers | 2.2.8 11.1.1.4 | 2—7 11—3 | Prefix, scratch | 5.2.3.2 | 5—4 |
| | | | Primary task | 7.1.1.1 | 7—2 |
| | | | Printout | | |
| | | | job accounting record | 11.2.3 Fig. 11—3 | 11—8 11—9 |
| | | | program termination | 8.3.3 | 8—13 |
| | | | Priority, task | 7.2.3 7.3.6 | 7—4 7—13 |
| | | | Program and machine error control | 2.2.7 | 2—7 |
| | | | Program check island code | | |
| | | | attaching | 8.6.1.1 | 8—35 |
| | | | common, all tasks in a job | | |
| | | | step | Fig. 8—11 | 8—50 |
| | | | description | 8.6.5 | 8—40 |
| | | | discrete, each task in a job | | |
| | | | step | Fig. 8—10 | 8—49 |
| | | | examples | Fig. 8—4 Fig. 8—5 | 8—41 8—42 |
| | | | exiting from | 8.6.4.1 | 8—39 |
| | | | multitasking | 8.6.9.1 | 8—49 |
| | | | Program initiation and loading | 8.1.1 | 8—1 |
| | | | Program linkage | | |
| | | | call a program | 8.5.4 | 8—28 |
| | | | description | 8.5 | 8—25 |
| | | | procedure | 8.5.2 | 8—26 |
| | | | register conventions | 8.5.1 | 8—25 |
| | | | register save area | 8.5.3 | 8—27 |
| | | | | Fig. 8—3 | 8—27 |
| | | | | Table 8—1 | 8—28 |
| | | | restore registers and return | 8.5.7 | 8—32 |
| | | | save register contents | 8.5.6 | 8—30 |
| | | | Program loader | | |
| | | | block loader | 8.2.1 | 8—2 |
| | | | description | 8.2 | 8—2 |
| | | | error processing | 8.2.5 | 8—5 |
| | | | library search order | 8.2.3 | 8—4 |
| | | | load a program phase (LOAD) | 8.2.6 | 8—5 |
| | | | locate a program phase header (LOADI) | 8.2.8 | 8—9 |
| | | | load a program phase and relocate (LOADR) | 8.2.7 | 8—7 |
| | | | read pointer, repetitive loads | 8.2.4 | 8—4 |
| | | | relocation | 8.2.2 | 8—3 |
| | | | Program management | | |
| | | | control stream reader | 8.8 | 8—55 |
| | | | description | 8.1 | 8—1 |

| Term | Reference | Page | Term | Reference | Page |
|------------------------------------|-----------------|------|--|-----------|------|
| S | | | | | |
| SAT | | | SETCS macro instruction | 8.8.5 | 8—59 |
| block number processing | 6.10 | 6—56 | SETIME macro instruction | | |
| controlling disk file processing | 6.4 | 6—19 | continue processing until | | |
| controlling tape file processing | 6.9 | 6—51 | interrupt | 8.4.2.2 | 8—22 |
| description | 6.1 | 6—1 | example | Fig. 8—2 | 8—23 |
| disk file interface | 6.3 | 6—10 | function | 8.4.2.1 | 8—21 |
| disk file organization and | | | interval timer | 8.6.7 | 8—45 |
| addressing methods | 6.2 | 6—1 | timer services | 8.4 | 8—15 |
| system standard tape labels | 6.6 | 6—26 | Shared filelock capability | 6.3.1.2 | 6—13 |
| tape file interface | 6.8 | 6—45 | SIB | 8.4.1.1 | 8—16 |
| tape files | 6.5 | 6—25 | SNAP macro instruction | 9.1.1 | 9—1 |
| tape volume and file organization | 6.7 | 6—37 | SNAPF macro instruction | 9.1.1 | 9—1 |
| See also disk SAT files | | | Snapshot display | 2.2.9.2 | 2—8 |
| and tape SAT files. | | | Snapshot dumps | 9.1.1 | 9—1 |
| SAT macro instruction | 6.8.1 | 6—45 | Space assignment | | |
| Save area, register | 8.5.3 | 8—27 | existing file | 5.3.2 | 5—7 |
| | Fig. 8—3 | 8—27 | new file | 5.3.1 | 5—5 |
| | Table 8—1 | 8—28 | Space control, disk | 6.2.4 | 6—4 |
| Save area address | 8.6.8 | 8—46 | Spooler | 11.1.1.3 | 11—2 |
| SAVE macro instruction | | | Spooling | | |
| function | 8.5.6 | 8—30 | breakpoint in output file | 11.1.3 | 11—4 |
| program linkage | 8.5 | 8—25 | description | 2.2.8 | 2—7 |
| Scratch routine, disk | | | initialization | 11.1.1.1 | 11—1 |
| description | 5.2.3 | 5—3 | input reader | 11.1.1.2 | 11—2 |
| scratch all by date | 5.2.3.3 | 5—4 | output writer | 11.1.1.4 | 11—3 |
| scratch by prefix | 5.2.3.2 | 5—4 | relationship of devices and programs | Fig. 11—1 | 11—2 |
| scratch file | 5.2.3.1 | 5—4 | special functions | 11.1.1.5 | 11—4 |
| Scratching files | 5.2.3.1 | 5—4 | spooler | 11.1.1.3 | 11—2 |
| | 5.3.3 | 5—9 | use | 11.1.2 | 11—4 |
| SCRATCH macro instruction | | | Standard load modules | 8.2 | 8—2 |
| disk | 5.3.3 | 5—9 | Standard system error message interface | 2.2.9.4 | 2—8 |
| diskette | 5.5.2 | 5—16 | Standard tape labels | | |
| Search order, library | 8.2.3 | 8—4 | system | 6.6 | 6—26 |
| Second file header label | See HDR2 label. | | tape volume organization | 6.7.1 | 6—38 |
| SEEK macro instruction | 6.2.1 | 6—2 | Standard tape volume organization | | |
| | 6.4.6 | 6—24 | description | 6.7.1 | 6—38 |
| Seek separation, disk | 2.2.14 | 2—10 | multifile volume with | | |
| Selective dynamic dump | 9.1.1 | 9—1 | end-of-file | Fig. 6—12 | 6—40 |
| Selector channel, BCW | | | multifile volumes with | | |
| format | Fig. 4—5 | 4—16 | end-of-volume | Fig. 6—13 | 6—41 |
| Sequence field | 3.3.6 | 3—8 | volumes containing a single file | Fig. 6—11 | 6—39 |
| Service request macro instructions | | | Start-of-data (/ \$) job control statement | | |
| (imperative) | 4.2.1 | 4—3 | control stream embedded data | 8.8.3 | 8—57 |
| | | | monitor input | 9.3.1.1 | 9—23 |
| | | | | 9.3.1.2 | 9—25 |

| Term | Reference | Page | Term | Reference | Page |
|---------------------------------------|---------------------------|-------|---|--------------------------------------|------|
| Statement conventions | 3.2 | 3—1 | System information block (SIB) | 8.4.1.1 | 8—16 |
| Storage display action | 9.3.5.1.2 | 9—40 | System information control description | 8.7 | 8—51 |
| Storage displays | | | get data from communication region | 8.7.1 | 8—52 |
| abnormal termination | 9.1.3 | 9—10 | get data from system control tables | 8.7.3 | 8—53 |
| checkpoint and restart description | 9.2 | 9—10 | put data into communication region | 8.7.2 | 8—53 |
| monitor and trace | 9.1 | 9—1 | System library file | 6.3.1 | 6—14 |
| normal termination dumps | 9.3 | 9—22 | System log | 10.1.2 | 10—6 |
| snapshot dumps | 9.1.2 | 9—5 | | | |
| | 9.1.1 | 9—1 | System standard tape labels | See tape labels, system standard. | |
| Storage reference option (S) | 9.3.4.1 | 9—32 | | | |
| STXIF macro instruction | 8.6 | 8—34 | | | |
| | 8.6.1 | 8—35 | | | |
| Subtask | 7.1.1.2 | 7—2 | | | |
| Supervisor | | | | | |
| description | 1.1 | 1—1 | Table generation macro instruction (declarative) | 4.2.1 | 4—2 |
| diagnostic and debugging aids | Section 9 | | Tape block number | 4.4.1 | 4—33 |
| disk space management | Section 5 | | Fig. 4—9 | 4—34 | |
| interrupt handling | 2.1 | 2—1 | Tape control appendage (TCA) | See TCA macro instruction. | |
| job accounting | 11.2 | 11—5 | Tape data management system | 6.5 | 6—25 |
| macro instructions | Section 3 | | Tape files, block numbered | 4.4 | 4—33 |
| main storage requirements | 1.2.2 | 1—2 | Tape labels, system standard description | 6.6 | 6—26 |
| message display and logging | 10.1 | 10—1 | file header | 6.6.2 | 6—29 |
| | 10.2 | 10—6 | file trailer | 6.6.3 | 6—33 |
| modular functions | See modular functions. | | nonstandard | 6.7.2 | 6—42 |
| multijobbing and multitasking | 1.2.3 | 1—3 | standard tape volumes | 6.7.1 | 6—38 |
| | Section 7 | | unlabeled | 6.7.3 | 6—44 |
| operator communication | 10.3 | 10—17 | volume | 6.6.1 | 6—27 |
| operator intervention | 1.2.4 | 1—3 | Tape restrictions | 4.4.2 | 4—33 |
| program management | Section 8 | | Tape SAT files | | |
| PIOCS | Section 4 | | block number processing | 6.10 | 6—56 |
| spooling | 11.1 | 11—1 | close | 6.9.6 | 6—55 |
| system access technique | Section 6 | | control tape unit functions | 6.9.5 | 6—54 |
| SWAP macro instruction | 4.2.1 | 4—1 | defining | 6.8.1 | 6—45 |
| | 4.2.9 | 4—29 | description | 6.5 | 6—25 |
| | | | get next logical block | 6.9.2 | 6—52 |
| Symbolic addresses | | | interface | 6.8 | 6—45 |
| abnormal termination island code | Fig. 8—6 | 8—44 | open | 6.9.1 | 6—51 |
| interval timer island code | Fig. 8—7 | 8—45 | output next logical block processing | 6.9.3 | 6—53 |
| operator communication island code | Fig. 8—8 | 8—47 | system standard labels | 6.9 | 6—51 |
| program check island code | Fig. 8—4 | 8—41 | tape control appendage | 6.6 | 6—26 |
| System access technique | See SAT. | | volume and file organization | 6.8.2 | 6—47 |
| System control tables | 8.7.3 | 8—53 | wait for block transfer | 6.7 | 6—37 |
| System debugging aids | | | 6.9.4 | 6—54 | |
| history tables | 9.4.1 | 9—46 | Tape system access technique | See TSAT. | |
| mini monitor | 9.4.2 | 9—48 | Tape unit functions | 6.9.5 | 6—54 |
| pseudo monitor | 9.4.1 | 9—46 | | | |
| resident supervisor monitor | 9.4.1 | 9—46 | | | |
| summary | Table 9—3 | 9—52 | | | |

| Term | Reference | Page | Term | Reference | Page |
|-----------------------------------|--------------------------|------|--------------------------------------|---------------------------------|-------|
| Tape volume and file organization | | | Timer interrupt facilities | | |
| description | 6.7 | 6—37 | cancel previous request | 8.4.2.4 | 8—24 |
| nonstandard | 6.7.2 | 6—42 | continue processing until interrupt | 8.4.2.2 | 8—22 |
| standard | 6.7.1 | 6—38 | description | 8.4.2 | 8—20 |
| unlabeled | 6.7.3 | 6—44 | set (SETIME) | 8.4.2.1 | 8—21 |
| Tape volume label group | 6.6.1 | 6—27 | wait for interrupt | 8.4.2.3 | 8—24 |
| Task control | 2.2.1 | 2—2 | Timer services | | |
| Task control block (TCB) | 7.2.1 | 7—2 | current date | 8.4.1.1 | 8—16 |
| Task management | | | description | 2.2.6 | 2—6 |
| creation | 7.2.2 | 7—2 | get current date and time | 8.4 | 8—15 |
| description | 7.3.2 | 7—9 | (GETIME) | 8.4.1.3 | 8—17 |
| generate event control block | 7.2 | 7—2 | interrupt facilities | Fig. 8—1 | 8—19 |
| hierarchical structure | 7.2.1 | 7—2 | time of day | See timer interrupt facilities. | |
| macro instructions | 7.3.1 | 7—6 | Trace | 8.4.1.2 | 8—17 |
| priority | 7.2.6 | 7—4 | See monitor and trace. | | |
| queue driven task | 7.3 | 7—5 | Trace job control option | 9.3.1.1 | 9—23 |
| reactivate a task | 7.2.3 | 7—3 | Transient loader | 2.2.3 | 2—5 |
| termination | 7.3.6 | 7—13 | Transient overlay | 2.2.3 | 2—5 |
| yield until task completion | 7.2.5 | 7—4 | Transient routines | 1.2.2 | 1—2 |
| Task switches | 8.6.3 | 8—39 | Transient scheduler | 2.2.3 | 2—5 |
| Task synchronization | | | TYIELD macro instruction | | |
| activate waiting task | 7.4.4 | 7—17 | function | 7.3.4 | 7—11 |
| description | 7.4 | 7—14 | multitasking | 7.3 | 7—6 |
| multiple task wait | 7.4.1 | 7—14 | | | |
| wait for task completion | 7.4.3 | 7—16 | | | |
| Tasks | | | | | |
| attaching island code | 8.6.1 | 8—35 | Unit of store | 6.3.2 | 6—15 |
| definition | 1.1 | 1—1 | Unlabeled tape volume | | |
| detaching island code | 8.6.2 | 8—38 | organization | 6.7.3 | 6—44 |
| TCA macro instruction | 6.8.1 | 6—45 | Fig. 6—16 | 6—44 | |
| 6.8.2 | 6—47 | | Unsolicited message | 10.3.1 | 10—18 |
| TCB | 7.2.1 | 7—2 | User-operator communication | 10.3.1 | 10—17 |
| Termination, program | See program termination. | | User program switch indicator (UPSI) | 8.7 | 8—51 |
| Termination dumps | | | | | |
| abnormal | 9.1.3 | 9—10 | | | |
| normal | 9.1.2 | 9—5 | | | |
| Time of day | 8.4.1.2 | 8—17 | | | |
| | 8.4.1.3 | 8—17 | | | |

| Term | Reference | Page |
|---|----------------------------------|------|
| VCALL macro instruction function program linkage | 8.5.4 | 8—28 |
| | 8.5 | 8—25 |
| VOL job control statement | 6.6.1 | 6—27 |
| Volume labels, description | 6.6.1 | 6—27 |
| Volume recognition, automatic | 2.2.10 | 2—8 |
| Volume serial number (VSN) | 6.6.1 | 6—27 |
| | Fig. 6—6 | 6—28 |
| Volume table of contents (VTOC) | See VTOC. | |
| Volumes nonstandard tape standard tape | See nonstandard tape volumes. | |
| | See standard tape volumes. | |
| VOL1 label description field description formats | 6.6.1 | 6—27 |
| | Table 6—1 | 6—29 |
| | Fig. 6—6 | 6—28 |
| | Fig. 6—17 | 6—59 |
| VTOC description disk space management user block access | 2.2.5 | 2—6 |
| | 5.1 | 5—1 |
| | 3.2.5 | 5—12 |

| Term | Reference | Page |
|--|------------|------|
| W | | |
| WAIT macro instruction I/O synchronization task synchronization | 4.3.1 | 4—31 |
| | 7.4.2 | 7—15 |
| WAIT parameter, SETIME macro instruction | 8.4.2.2 | 8—22 |
| | 8.4.2.3 | 8—24 |
| WAITF macro instruction disk processing magnetic tape processing | 6.4.4 | 6—21 |
| | 6.9.4 | 6—54 |
| WAITM macro instruction I/O synchronization task synchronization | 4.3.2 | 4—32 |
| | 7.4.4 | 7—16 |
| WTL macro instruction | 10.2.1 | 10—6 |
| | 10.1 | 10—2 |
| | Table 10—1 | 10—2 |
| WTLD macro instruction | 10.2.2 | 10—9 |
| | 10.1 | 10—2 |
| | Table 10—1 | 10—2 |



Comments concerning this manual may be made in the space provided below. Please fill in the requested information.

System: _____

Manual Title: _____

UP No: _____ Revision No: _____ Update: _____

Name of User: _____

Address of User: _____

Comments:

CUT



CUT




FIRST CLASS
PERMIT NO. 21
BLUE BELL, PA.

BUSINESS REPLY MAIL

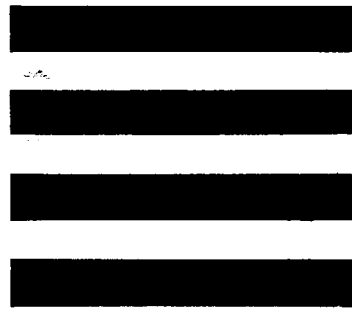
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY

SPERRY  **UNIVAC**

P.O. BOX 500
BLUE BELL, PA.
19424

ATTN: SYSTEMS PUBLICATIONS DEPT.



FOLD

FOLD

Comments concerning this manual may be made in the space provided below. Please fill in the requested information.

System: _____

Manual Title: _____

UP No: _____ Revision No: _____ Update: _____

Name of User: _____

Address of User: _____

Comments:

CUT

OLD

FIRST CLASS
PERMIT NO. 21
BLUE BELL, PA.

BUSINESS REPLY MAIL

NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY

SPERRY  **UNIVAC**

P.O. BOX 500
BLUE BELL, PA.
19422

ATTN: SYSTEMS PUBLICATIONS DEPT.

CUT

OLD