SPERRY UNIVAC
COMPUTER SYSTEMS

PUBLICATIONS UPDATE

Operating System/3 (OS/3)

Supervisor Macroinstructions

User Guide/Programmer Reference

UP-8832-B

This Library Memo announces the release and availability of Updating Package B to "SPERRY UNIVAC Operating System/3 (OS/3) Supervisor Macroinstructions User Guide/Programmer Reference", UP-8832.

This update documents the following changes for release 8.0:

- Enhancement of the OC STXIT routine
- Restrictions to the monitor routine
- Addition of the Soft-Patch Symbiont to the system debugging aids

This update also includes minor technical corrections to material applicable to the supervisor macroinstructions prior to release 8.0.

Copies of Updating Package B are now available for requisitioning. Either the updating package only, or the complete manual with the updating package may be requisitioned by your local Sperry Univac representative. To receive the updating package, order UP-8832-B. To receive the complete manual, order UP-8832.

LIBRARY MEMO ONLY	LIBRARY MEMO AND ATTACHMENTS	THIS SHEET IS
Mailing Lists BZ, CZ and MZ	Mailing Lists B00, B18, 28U, and 29U (Package B to UP-8832, 41 pages plus Memo)	Library Memo for UP-8832-B
		RELEASE DATE:
		September, 1982

SPERRY UNIVAC
COMPUTER SYSTEMS

SPERRY UNIVAC
SUITE 906
1177 WEST HASTINGS ST

VANCOUVER BC V6E 2K3

ATTN: CHARLIE GIBBS

00155
CAV208M45541 UP 8832-A

UAS

CAV

PUBLICATIONS UPDATE

Operating System/3 (OS/3)

Supervisor Macroinstructions

User Guide/Programmer Reference

UP-8832–A

This Library Memo announces the release and availability of Updating Package A to "SPERRY UNIVAC Operating System/3 (OS/3) Supervisor Macroinstructions User Guide/Programmer Reference", UP-8832.

This update discusses the newly added support of checkpoint files on diskette and magnetic tape available with release 7.1 of OS/3. It also contains minor corrections and clarifications.

Copies of Updating Package A are now available for requisitioning. Either the updating package only or the complete manual with the updating package may be requisitioned by your local Sperry Univac representative. To receive only the updating package, order UP-8832–A. To receive the complete manual, order UP-8832.

LIBRARY MEMO ONLY	LIBRARY MEMO AND ATTACHMENTS	THIS SHEET IS:
Mailing Lists BZ, CZ (less DE, GZ, HA) and MZ	Mailing Lists DE, GZ, HA, 28U and 29U (Package A to UP-8832, 40 pages plus Memo)	Library Memo for UP-8832–A
		RELEASE DATE: September, 1981

SPERRY UNIVAC
COMPUTER SYSTEMS

PUBLICATIONS RELEASE

Operating System/3 (OS/3)

Supervisor
Macroinstructions

User Guide/
Programmer Reference

[illegible]

This Library Memo announces the release and availability of "SPERRY UNIVAC® Operating System/3 (OS/3) Supervisor Macroinstructions User Guide/Programmer Reference", UP-8832.

This manual describes the Operating System/3 (OS/3) Supervisor macroinstructions and gives instructions and examples for their use. It also contains Supervisor diagnostic and debugging aid information.

Additional copies may be ordered by your local Sperry Univac representative.

LIBRARY MEMO ONLY	LIBRARY MEMO AND ATTACHMENTS	THIS SHEET IS
Mailing Lists BZ, CZ (less DE, GZ and HA) and MZ	Mailing Lists DE, GZ, HA, 28U and 29U (Covers and 245 pages)	Library Memo
		RELEASE DATE: October, 1980

Supervisor Macroinstructions

OS/3

User Guide/
Programmer Reference

Environment: System 80

H

SPERRY UNIVAC

UP-8832

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 consult the appropriate release documentation or 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 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.

PRINTED IN U.S.A.

PAGE STATUS SUMMARY

ISSUE: Update D – UP-8832
RELEASE LEVEL: 8.2 Forward

Part/Section	Page Number	Update Level
Cover/Disclaimer		Orig.
PSS	1	D
Preface	1	A
	2	Orig.
Contents	1,2	Orig.
	3	D
	4	Orig.
	5	A
	6,7	B
1	1 thru 7	Orig.
	8	D
	9	Orig.
	10	A
2	1	Orig.
	2	B
	2a	B
	3 thru 48	Orig.
	49	B
	50 thru 56	Orig.
	56a, 56b	D*
	57	A
	58 thru 63	Orig.
3	1 thru 5	Orig.
4	1	A
	2 thru 9	Orig.
	10	B
	11 thru 15	Orig.
	16	A
	17 thru 22	Orig.
	23	A
	24	Orig.
	25	B
	26 thru 44	Orig.
	45 thru 48	A
	49 thru 55	Orig.
	56	B
	57 thru 63	Orig.
5	1 thru 19	Orig.
6	1	Orig.
	2	B
	3 thru 5	Orig.
7	1	Orig.
	2	B
	3 thru 9	Orig.
	10 thru 13	A
	14	Orig.
87(cont.)	15, 16	A
	17 thru 21	Orig.
	22	B
	23 thru 25	Orig.
	26	A
	27 thru 39	Orig.
	40	B
	40a	B
	41	A
	42 thru 46	Orig.
	47, 48	B
	49	C
	50 thru 53	B
Glossary	1 thru 5	Orig.
	6,7	B
	8,9	Orig.
Index	1	Orig.
	2,3	A.
	4	Orig.
	5	D
	6	A
	7	Orig.
	8	D
	9	Orig.
	10	B
	11,12	Orig.
User Comment Sheet		

Part/Section	Page Number	Update Level

*New pages

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.

PAGE STATUS SUMMARY

ISSUE: Update B – UP-8832

RELEASE LEVEL: 8.0 Forward

Part/Section	Page Number	Update Level
Cover/Disclaimer		Orig.
PSS	1	B
Preface	1	A
	2	Orig.
Contents	1 thru 4	Orig.
	5	A
	6, 7	B
1	1 thru 9	Orig.
	10	A
2	1	Orig.
	2	B
	2a	B*
	3 thru 48	Orig.
	49	B
	50 thru 56	Orig.
	57	A
	58 thru 63	Orig.
3	1 thru 5	Orig.
4	1	A
	2 thru 9	Orig.
	10	B
	11 thru 15	Orig.
	16	A
	17 thru 22	Orig.
	23	A
	24	Orig.
	25	B
	26 thru 44	Orig.
	45 thru 48	A
	49 thru 55	Orig.
	56	B
	57 thru 63	Orig.
5	1 thru 19	Orig.
6	1	Orig.
	2	B
	3 thru 5	Orig.
7	1	Orig.
	2	B
	3 thru 9	Orig.
	10 thru 13	A
	14	Orig.
	15, 16	A
	17 thru 21	Orig.
	22	B
	23 thru 25	Orig.
	26	A
7 (cont)	27 thru 39	Orig.
	40	B
	40a	B
	41	A
	42 thru 46	Orig.
	47	B
	48 thru 53	B*
Glossary	1 thru 5	Orig.
	6, 7	B
	8, 9	Orig.
Index	1	Orig.
	2, 3	A
	4, 5	Orig.
	6	A
	7	Orig.
	8	B
	9	Orig.
	10	B
	11, 12	Orig.
User Comment Sheet		

*New pages

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 describes the SPERRY UNIVAC Operating System/3 (OS/3) Supervisor macroinstructions and gives instructions and examples for their use. A user of this manual should have a knowledge of the OS/3 assembler, job control, and data management. This document is one of three manuals that explain the supervisor; the others are: the Introduction manual, UP-8830, which gives a general description of the supervisor, and Concepts and Facilities, UP-8831, which provides a functional description and gives information on generating the supervisor. This manual is arranged as follows:

- Section 1. Introduction

  This section provides an introduction and gives the conventions for writing the macroinstruction statements that request the supervisor services.

- Section 2. Program Management Macros

  This section gives the function and format of the supervisor macroinstructions that provide for management of program design and execution.

- Section 3. Disk and Diskette Space Management

  This section gives the function and format of the macroinstructions that provide for automatic disk and diskette space management.

- Section 4. System Access Technique Macros

  This section gives the function and format of the macroinstructions for the system access technique, which provides efficiency in the handling of disk, format label diskette, and tape files.

- Section 5. Multitasking Macros

  This section gives the function and format of the macroinstructions associated with the system's multijobbing and multitasking capability.

- Section 6. Spooling – Breakpoint Macroinstruction

  This section describes spooling and gives the function and format of the breakpoint macro.

- Section 7. Diagnostic and Debugging Aids

  This section describes the supervisor diagnostic and debugging aids and gives the function and format of the associated macros.

- Glossary

  Briefly describes some of the terms used in this manual.

Contents

FIGURES

TABLES

1. Introduction

1.1. GENERAL

The services provided by the System 80 Operating System/3 (OS/3) supervisor are described in the OS/3 Supervisor Concepts and Facilities Manual, UP-8831. The basic assembly language (BAL) programmer utilizes these services through a complement of supervisor macroinstructions whose function and format, along with examples of their use, are contained in this manual. (These macroinstructions are used in assembly language and cannot be directly called by higher-level languages.)

Following are the conventions used in writing the supervisor macroinstructions and general rules for using the assembler coding form. This section also includes a summary listing of the supervisor macroinstructions, which includes a list of the pages where the format of each macroinstruction is located.

1.2. MACROINSTRUCTION FORMAT AND STATEMENT CONVENTIONS

The general format of a macroinstruction is:

LABEL	△OPERATION△	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 macroinstruction 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 macroinstructions 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:

  $$\left\{\begin{array}{l}\texttt{PC}\\\texttt{IT}\\\texttt{AB}\end{array}\right\}$$

  $$\left\{\begin{array}{c}\texttt{input-area}\\\texttt{(1)}\end{array}\right\}$$

- 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]
  ```

  $$\left[\texttt{,}\left\{\begin{array}{l}\texttt{CCB-name}\\\texttt{ALL}\\\texttt{(1)}\end{array}\right\}\right]$$

  ```
  [,ERROR=symbol]
  [,WAIT=YES]
  ```

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

  $\underline{\text{P}}\text{R:xv}$

  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:

  $\left[,\begin{Bmatrix}\text{S}\\\text{M}\end{Bmatrix}\right]$

- 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 macroinstruction with one mandatory positional parameter (phase-name) and four optional positional parameters (load-addr, error-addr, and R):

  Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LOAD	$\begin{Bmatrix}\text{phase-name}\\\text{(1)}\end{Bmatrix}\left[,\begin{Bmatrix}\text{load-addr}\\\text{(Ø)}\end{Bmatrix}\right]$ $\left[,\begin{Bmatrix}\text{error-addr}\\\text{(r)}\end{Bmatrix}\right]$ [,R] [,DA]

Macroinstruction statements might be written:

```
1          10     16
_____________________________________________________
           LOAD   RECAPLØ5,INADDR,ERADDR,R
           LOAD   RECAPLØ5,,ERADDR
           LOAD   RECAPLØ5,INADDR
           LOAD   RECAPLØ5
```

- 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 macroinstruction 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]

  Macroinstruction statements might be written:

```
1          10     16                                                      72
_____________________________________________________________________________
           PCA    IOAREA1=WORKAREA,BLKSIZE=256,FORMAT=NO,EODADDR=ENDNAME,  X
                  SEQ=YES,SIZE=1,UOS=1,VERIFY=YES
           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 macroinstruction, you intend to load the designated register with the required data prior to the execution of the macroinstruction. For example, in the format illustration:

LABEL	△OPERATION△	OPERAND
[symbol]	GETCS	$\left\{\begin{matrix}\text{input-area}\\(1)\end{matrix}\right\}\left[,\left\{\begin{matrix}\text{number-of-records}\\(Ø)\\1\end{matrix}\right\}\right]$ $\left[,\left\{\begin{matrix}\text{error-addr}\\(r)\end{matrix}\right\}\right]\left[,\left\{\begin{matrix}n\\8Ø\end{matrix}\right\}\right]$

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 macroinstruction statement. For example, the instruction:

```
1        10    16
         GETCS WORK,(Ø),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 macroinstruction 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.

1.3. ASSEMBLER CODING FORM

To convert your written program to a form that can be conveniently input to the computer, you enter your written work in the form of 80-column card images. 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 assembler coding form. (See Figure 1-1.)

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 of preparing your input card images.

The following subsections 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-8913 (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.

LABEL		△OPERATION△	OPERAND	
1	10	16		72

Figure 1—1. Assembler Coding Form

1.3.1. Label Field

The first eight columns of the assembler coding form may contain a symbol. This symbol can be used to identify a line of coding, or to identify a main storage or constant area. The rules for using the label field are:

1. The symbol must start in column 1.
2. The symbol must begin with an alphabetic or special character.
3. The symbol must not exceed eight characters in length.
4. The symbol must not contain embedded blanks or other special characters.
5. The field must be terminated by a blank.
6. An asterisk (*) entered in column 1 indicates to the assembler that the entry on this line is to be treated as comments.

1.3.2. Operation Field

The operation code is written in the operation field (columns 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 or proc instructions.
3. The operation field must be terminated by a blank.
4. An operation code consisting of six characters (for example, the macroinstruction ATTACH), will fall in columns 10 through 15. In this case, column 16 must be blank to terminate the operation field.

1.3.3. 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.

1.3.4. 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.

1.3.5. 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.

1.3.6. 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 updating the input card images as needed. 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 card images already in the program. Some programmers use letters in addition to the numbers. This is useful in identifying the program from which card images have come if they have been removed for any reason.

1.4. MACROINSTRUCTIONS

1.4.1. Declarative Macroinstructions

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

1.4.2. Imperative Macroinstructions

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

1.4.3. Summary of Supervisor Macroinstructions

Table 1-1 lists the OS/3 supervisor macroinstructions and gives a brief description of their function, along with the page number where the format for each appears. ARGLST, DTFPF, PCA, SAT, TCA, ECB, and DDCPF are the declarative macroinstructions in this list; the remainder are imperative macroinstructions. The macroinstructions are arranged in this manual in the same groups as they appear in this table.

Table 1—1. Supervisor Macroinstructions (Part 1 of 3)

PROGRAM MANAGEMENT		Format Page
Program Loader		
LOAD	Load a program phase and return control.	2-4
LOADR	Load a program phase, relocate address-constants, and return control.	2-6
LOADI	Locate a program phase and store its phase header in a workarea.	2-8
FETCH	Load a program phase and branch.	2-10
Job and Task Termination		
EOJ	Terminate a job step normally.	2-13
CANCEL	Terminate a job abnormally.	2-14
Timer Services		
GETIME	Obtain current time and date.	2-17
SETIME	Set an elapsed time counter for the requesting task.	2-21
Subroutine Linkage		
CALL/VCALL	Call a program.	2-29
ARGLST	Generate an argument list.	2-30
SAVE	Save register contents.	2-31
RETURN	Restore registers and return.	2-32
Island Code Linkage		
STXIT	Link to island code subroutine.	2-38
EXIT	Exit from island code subroutine.	2-41
System Information Control		
GETCOM	Retrieve data from job communication area.	2-54
PUTCOM	Place data into job communication area.	2-55
GETINF	Retrieve data from system control tables.	2-55
GETLDA	Transfer data from LDA	2-56a
PUTLDA	Transfer data to LDA	2-56b
Control Stream Reader		
GETCS	Retrieve embedded data file submitted in job control stream.	2-60
SETCS	Reset pointer to embedded data file.	2-61

Table 1—1. Supervisor Macroinstructions (Part 2 of 3)

			Format Page
DISK AND DISKETTE SPACE MANAGEMENT			
Disk			
	OBTAIN	Access VTOC user block.	3-2
Diskette			
	OBTAIN	Obtain diskette label information.	3-4
SYSTEM ACCESS TECHNIQUE (SAT)			
Disk SAT			
	DTFPF	Define a partitioned file.	4-11
	PCA	Define a partition control appendage.	4-14
	OPEN	Open a disk file.	4-19
	GET	Retrieve next logical block.	4-20
	PUT	Output a logical block.	4-21
	WAITF	Wait for block transfer.	4-22
	READE	Search track by key, equal.	4-23
	READH	Search track by key, equal or higher.	4-23
	SEEK	Access a physical block.	4-24
	CLOSE	Close a disk file.	4-25
Tape SAT			
	SAT	Defines magnetic tape file.	4-45
	TCA	Defines a tape control appendage.	4-47
	OPEN	Open a tape file.	4-51
	GET	Get next logical block.	4-52
	PUT	Output next logical block.	4-53
	WAITF	Wait for block transfer.	4-53
	CNTRL	Control tape unit functions.	4-54
	CLOSE	Close a tape file.	4-55
MULTITASKING			
Task Management			
	ECB	Generate an event control block.	5-5
	ATTACH	Create and activate an additional task.	5-7
	DETACH	Terminate a task normally.	5-9
	TYIELD	Deactivate a task.	5-10
	AWAKE	Reactivate an existing nonactive task.	5-11
	CHAP	Change the priority of a task.	5-12
Task Synchronization			
	WAIT	Wait for a task request to complete.	5-14
	WAITM	Wait for one of several task requests to complete.	5-15
	POST	Activate the waiting task.	5-16
	TPAUSE	Deactivate one or more tasks other than the issuing task.	5-17
	TGO	Reactivate one or more tasks other than the issuing task.	5-18

Table 1—1. Supervisor Macroinstructions (Part 3 of 3)

DIAGNOSTIC AND DEBUGGING		Format Page
Storage Displays		
SNAP/SNAPF	Printout portions of main storage and return control.	7-2
DUMP	Printout the job main storage and terminate the job step.	7-7
Checkpoint Facility		
CHKPT	Record a checkpoint.	7-12
DDCPF	Define a SAT checkpoint file.	7-15
DCPOPN	Open a SAT checkpoint file.	7-15
DCPCLS	Close a SAT checkpoint file.	7-16
Monitor and Trace		
// OPTION TRACE	Monitor from start of job. (This is a job control statement, not a macroinstruction.)	7-20
SPOOLING		
DMBRK	Create a breakpoint in a spool output file.	6-5

1.5. PROGRAMMING CONSIDERATIONS FOR MACROINSTRUCTIONS

When the assembler encounters a macroinstruction, it generates machine code, which is called inline expansion code. This code can consist of machine instructions and machine data. Data, in turn, may consist of constants defined by the macroinstruction at assembly time and reserved main storage, which is not used until program execution time.

A macroinstruction, expecially if it is a request to the supervisor for some service, will usually generate a supervisor call (SVC) instruction. When the program is later executed, the SVC will either be processed by the resident supervisor, or a transient will be called. The actual processing of the request is then performed by the supervisor; the inline expansion code generated by your program is normally just used to set up parameters. Because SVC instructions are processed in the supervisor region and not in your program region, you can use macroinstructions freely, without having to allocate extra main storage to your job.

Of the 16 general registers available to assembler programs, registers 2 through 13 are generally left unchanged by macroinstructions. Registers 0 and 1, however, are the principal means by which data and program control are passed between user programs and macroinstructions. Consequently, these two registers cannot be counted upon to remain unchanged during execution of any macroinstruction. If it is important to keep intact the data in either of these two registers, you should save the data in main storage before calling the macroinstruction and load the data back into the register afterwards. In addition to registers 0 and 1, program linkage macroinstructions (2.4) change registers 14 and 15. Most other macroinstructions, though, leave these two registers intact.

2. Program Management Macros

2.1. 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 macroinstructions are available:

- LOAD

  Load a phase and return control.

- LOADR

  Load a phase, relocate address constants, and return control.

- LOADI

  Locate a phase and store its phase header in a workarea.

- FETCH

  Load a phase and give it control.

The use of these macroinstructions is described in the following subsections.

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.

2.1.1. Block Loader

The LOAD, LOADR, LOADI, and FETCH macroinstructions 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-8841 (current version)). This may be done at any time before the job is executed and there is no need to specify in the macroinstructions loading the phase whether the load module phase is in the standard format or in block format.

NOTE:

The following problem may exist if when loading a blocked module you consistently get the 54 error code. If the total number of partition 1 and partition 2 extents is greater than 14, then certain blocked modules start getting the 54 error code. The temporary solution is to copy your load file to a temporary file. Then, scratch and allocate the load file, and copy the temporary file to the load file.

If the load file is $Y$LOD, you must IPL the system again from another SYSRES to scratch $Y$LOD from your original SYSRES.

2.1.2. 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 relocation of address constants (A-cons) when the phase is located 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 because the loader has no way of knowing where that phase was loaded.

The following examples illustrate when the loader performs relocation:

Macro	Call	User Program Relocation
LOAD	phase-name	No
LOADR	phase-name	No
FETCH	phase-name	No
LOAD	phase-name,altad	No*
LOADR	phase-name,altad	Yes
FETCH	phase-name,entpt	No

*****Phase being loaded should be self-relocating.*

2.1.3. 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. Temproary 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.

2.1.4. 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 disk address (DA) optional parameter is used with the LOAD, LOADR, or FETCH macroinstruction, 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 macroinstructions, and should only be used for repetitive loads of the same module. It is not available for use with the LOADI macroinstruction. If you do not wish to add the DA capability to an assembled program, there is no need to reassemble.

2.1.5. 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).

2.1.6. Load a Program Phase (LOAD)

Function:

The LOAD macroinstruction 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 macroinstruction.

After execution of this macroinstruction, 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 macroinstruction does not relocate address constants regardless of whether an alternate load address is specified (positional parameter 2).

Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	LOAD	$\left\{\begin{matrix}\text{phase-name}\\ \text{(1)}\end{matrix}\right\}\left[,\left\{\begin{matrix}\text{load-addr}\\ \text{(0)}\end{matrix}\right\}\right]\left[,\left\{\begin{matrix}\text{error-addr}\\ \text{(r)}\end{matrix}\right\}\right]$[,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 nnnnnnpp 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.

(Ø)

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.

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 macroinstruction. This read pointer is used to find the phase on the second and all subsequent executions of this macroinstruction.

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 considerably reduce 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 macroinstruction is executed, and the 8-byte phase name is not overwritten.

2.1.7. Load a Program Phase and Relocate (LOADR)

Function:

The LOADR macroinstruction 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 LOADR macroinstruction.

After execution of this macroinstruction, 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 macroinstruction is identical to the LOAD macroinstruction except that all address constants in the phase are relocated if an alternate load address is specified (positional parameter 2).

This macroinstruction is used to load a phase at an address other than that at which it was linked.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LOADR	$\begin{Bmatrix}\text{phase-name}\\ \text{(1)}\end{Bmatrix}\left[,\begin{Bmatrix}\text{load-addr}\\ \text{(Ø)}\end{Bmatrix}\right]\left[,\begin{Bmatrix}\text{error-addr}\\ \text{(r)}\end{Bmatrix}\right]\text{[,R][,DA]}$

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 nnnnnnpp 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.

(Ø)

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.

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 macroinstruction. This read pointer is used to find the phase on the second and all subsequent executions of this macroinstruction.

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 considerably reduce 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 macroinstruction is executed, and the 8-byte phase name is not overwritten.

2.1.8. Locate a Program Phase Header (LOADI)

Function:

The LOADI macroinstruction locates the header record of a program phase and stores it in a workarea.

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 2.1.8.1.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	LOADI	$\left\{\begin{matrix}\text{phase-name}\\ \text{(1)}\end{matrix}\right\},\left\{\begin{matrix}\text{work-area-addr}\\ \text{(Ø)}\end{matrix}\right\}\left[,\left\{\begin{matrix}\text{work-area-length}\\ \text{13}\end{matrix}\right\}\right]$ $\left[,\left\{\begin{matrix}\text{error-addr}\\ \text{(r)}\end{matrix}\right\}\right]\text{[,R]}$

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 of the 8-character linker-assigned phase name in the format nnnnnnpp 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.

(Ø)

Specifies that register 0 has been preloaded with the workarea address.

Positional Parameter 3:

work-area-length

Specifies the number of bytes of the phase header that are to be placed in the workarea.

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 workarea.

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.

2.1.8.1. Program Phase Header

The format of the phase header is as follows:

Bytes	Contents
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

2.1.9. Load a Program Phase and Branch (FETCH)

Function:

The FETCH macroinstruction 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 macroinstruction, 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	$\left\{\begin{matrix}\text{phase-name}\\ \text{(1)}\end{matrix}\right\}\left[,\left\{\begin{matrix}\text{entry-point-name}\\ \text{(Ø)}\end{matrix}\right\}\right]$[,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 nnnnnnpp 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.

(Ø)

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.

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 macroinstruction. This read pointer is used to find the phase on the second and all subsequent executions of this macroinstruction.

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 considerably reduce 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 macroinstruction is executed, and the 8-byte phase name is not overwritten.

2.2. PROGRAM TERMINATION

The program termination macroinstructions 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 macroinstructions will also clear all I/O locks for the task.

The following macroinstructions are available:

- EOJ

  Causes normal job step termination.

- CANCEL

  Causes abnormal job termination and prints out the job main storage.

There are two other macroinstructions used for job and task termination:

- DETACH

  Causes normal termination of a task.

- DUMP

  Causes normal job step termination in addition to printing out the job main storage.

2.2.1. Normal Termination

Normal program termination is requested by means of the EOJ or DUMP macroinstructions, 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 detach all subtasks, delink outstanding I/O, and wait for all outstanding system functions to complete, 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 macroinstruction provides a printout of the contents of the job region if the appropriate //OPTION job control statement is specified (for example, DUMP, JOBDUMP, SYSDUMP, or ABRDUMP). After printing the region, the DUMP transient 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 macroinstruction.

2.2.2. Abnormal Termination

Abnormal job termination can be requested by you through the CANCEL macroinstruction, 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.

2.2.3. Printout

Both the CANCEL and DUMP macroinstructions 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, ABRDUMP, or SYSDUMP parameter was specified in the OPTION job control statement. Otherwise, both macroinstructions will execute normally; however, no printout will occur.

2.2.4. End-of-Job Step (EOJ)

Function:

The EOJ macroinstruction causes normal job step termination. It terminates a primary task or a subtask. If an EOJ macroinstruction is issued from a primary task with active subtasks, all subtasks are terminated. If an EOJ macroinstruction is issued from a subtask, only the subtask and any subtasks it created are terminated.

This macroinstruction also clears all locks for the task.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	EOJ	

There are no parameters for the EOJ macroinstruction.

The EOJ macroinstruction 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 macroinstruction 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 canceled (depending on the existence and function of an abnormal termination island code routine). An EOJ macroinstruction executed by a subtask is treated as a request for the DETACH macroinstruction function.

Error Conditions:

The job will be canceled if errors that 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).

2.2.5. Cancel a Job (CANCEL)

Function:

The CANCEL macroinstruction 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 macroinstruction also displays an abnormal termination message on the operator console indicating which job is being terminated and the error code defining the error. Unless NODUMP is entered as positional parameter 2, this macroinstruction provides a diagnostic storage dump of the job region similar to that produced by the DUMP macroinstruction.

This macroinstruction also clears all locks for the task.

Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	CANCEL	⎡⎧error-code⎫⎤[,NODUMP] ⎢⎨(Ø)⎬⎥ ⎣⎩Ø⎭⎦

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.

(Ø)

Indicates that register 0 has been preloaded with the error code.

If omitted, the error code is set to binary zero.

Positional Parameter 2:

NODUMP

Specifies no dump regardless of the dump options specified in the OPTION job control statement.

The CANCEL macroinstruction 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 macroinstruction, 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 macroinstruction, 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, ABRDUMP, or SYSDUMP option was specified via job control;
- the CANCEL macroinstruction does not specify NODUMP; and
- a printer was assigned to the job or is available.

2.3. 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 macroinstruction.

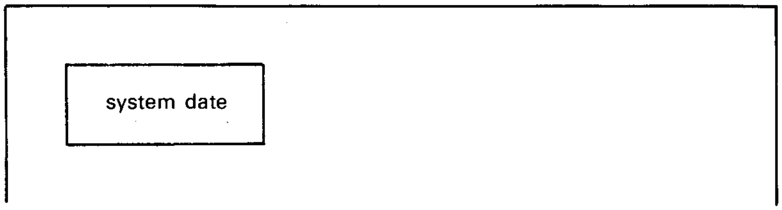
At times you may want to request an interrupt to your program after a specified interval. For example, you may wish to allow 30 seconds for a response from a terminal, and if no response if received within that time, branch to another subroutine or to another task. You can do this by using the SETIME macroinstruction.

2.3.1. Date and Time Facilities

2.3.1.1. Current Date

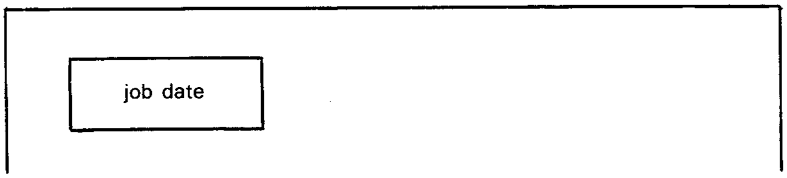
The current date is placed in the systems information block by the operator during initial program load. The date is automatically advanced each day at midnight unless the supervisor was configured at system generation time not to update. In that case, the operator must change the date through a console command. This date is referred to herein as the system date to distinguish it from the job date.

System Information Block



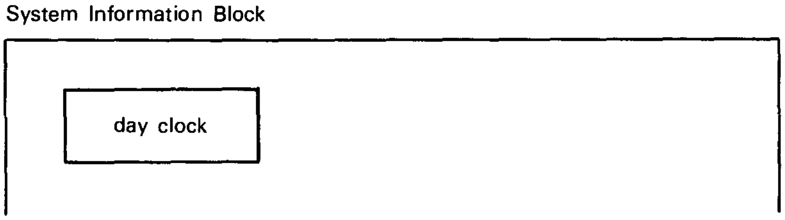
There is a date for each job, which is stored in the preamble for the job. This is the date you get when you use the GETIME macroinstruction. Normally, the job date is the same as the system date. However, you can change it using the SET job control statement which changes the date for your own job and 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.

Job Preamble



2.3.1.2. Time of Day

In addition to the current date, the GETIME macroinstruction gives you the time. The current time of day is maintained by a simulated day clock in the system information block. 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.



2.3.1.3. Get Current Date and Time (GETIME)

Function:

The GETIME macroinstruction 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.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	GETIME	$\left[\begin{Bmatrix}M\\S\end{Bmatrix}\right]$

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 macroinstruction with the S parameter or with no parameter, the time is expressed in packed decimal format in the form:

Ohhmmss+

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        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's assume you wish to print the date and exact time a job step is completed. The two subroutines shown in Figure 2-1 each get the date and time from the job preamble and the system information block, unpack and edit them into buffers, then print the contents of the buffers.

```
     1        10     16
 1.           GETIME S                        Returns date in register Ø and time in register 1.
 2.           ST     RØ,WS1                   Stores date from register Ø 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.           OI     BUFFER+9,X'FØ'           Changes contents of right hand byte of date from C1 to F1.
 7.           OI     BUFFER+17,X'FØ'          Changes contents of right hand byte of time from C3 to F3.
 8.           MVC    BUFFER(2),BUFFER+4       )
 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' '            > Inserts slashes, spaces, and periods in date and time.
14.           MVI    BUFFER+9,C' '            |
15.           MVC    BUFFER+1Ø(2),BUFFER+12   |
16.           MVI    BUFFER+12,C'.'           |
17.           MVC    BUFFER+13(2),BUFFER+14   |
18.           MVI    BUFFER+15,C'.'           )
19.           DMOUT  PRINT,BUFFER             Prints date and time from BUFFER.
2Ø.           EOJ                             Terminates the job step.
                .
                .
                .
                .
21.           GETIME S                        Returns date in RO and time in RI.
22.           ST     Ø,SAVE                   Stores date in full-word save.
23.           ED     DATMSK(1Ø),SAVE          Unpacks and edits date.
24.           MVC    BUFFER2(8),DATMSK+2      Moves edited date to buffer.
25.           ST     1,SAVE                   Stores time in save.
26.           ED     TIMMSK(1Ø),SAVE          Unpacks and edits time.
27.           MVC    BUFFER2+1Ø(8),TIMMSK+2   Moves edited time to buffer.
28.           DMOUT  PRINT,BUFFER2            Prints date and time.
                .
                .
                .
                .
29.  SAVE     DS     F                        Buffer initialized to blanks
3Ø.  BUFFER2  DC   CL18' '                    Date mask format;  99/99/99
31.  DATMSK   DC   X'4Ø212Ø2Ø612Ø2Ø612Ø2Ø'    Time mask format;  99.99.99
32.  TIMMSK   DC   X'4Ø212Ø2Ø4B2Ø2Ø4B2Ø2Ø'
```

Figure 2—1. Examples of GETIME Macroinstruction

Let's assume the GETIME macroinstruction 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:

Preamble	job date				System Information Block	day clock			
Register 0	07	71	02	4C	Register 1	00	93	01	3C

Following execution of line 7, BUFFER contains:

				7	7	1	0	2	4			0	9	3	0	1	3

Following execution of line 18, BUFFER contains the date (year/mon/day) and the time (hours.min.sec):

7	7	/	1	0	/	2	4			0	9	.	3	0	.	1	3

The date and time are printed:

77/10/24 09.30.13

If the subroutine in lines 21 through 32 is executed with the same original contents of registers 0 and 1, BUFFER2 will contain the following after execution of line 27

7	7	/	1	0	/	2	4			0	9	.	3	0	.	1	3

and will print the same date and time as the subroutine of lines 1 through 20.

If you write this macroinstruction 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 macroinstruction was executed at 10 seconds after midnight, September 26, 1979, registers 0 and 1 would contain:

```
1        10    16
         GETIME M
```

Register 0	07	90	92	6C

Register 1	00	00	27	10

2.3.2. Timer Interrupt Facilities

The timer services module also enables you to request a scheduled timer interrupt in the requesting task. Using the SETIME macroinstruction you may request an interrupt after any time period greater than 1 millisecond. You may:

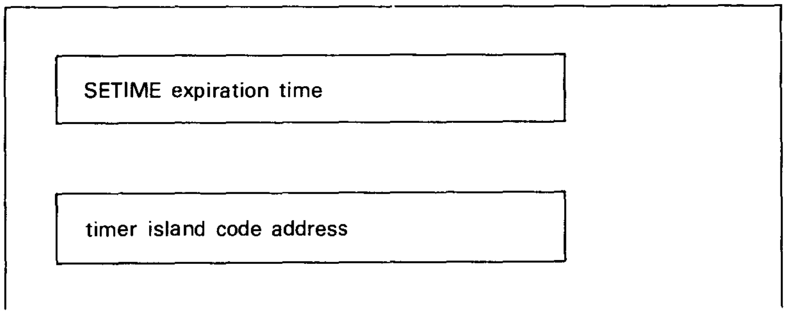
- continue processing the task until the interrupt, then transfer control to the task's timer island code;

- 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 macroinstruction 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.

Task Control Block



If timer island code is to be executed, a STXIT macroinstruction must have been previously issued to link the island code to this task. If no timer island code is present, or if the interrupt request was canceled, 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 macroinstruction.

2.3.2.1. Set Timer Interrupt (SETIME)

Function:

The SETIME macroinstruction 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 macroinstruction) is executed.

Note that, in this case, the STXIT macroinstruction 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 macroinstruction 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 macroinstruction.

This macroinstruction cancels any previous SETIME request. By using this macroinstruction with no parameters or with a time interval of zero, you can effectively eliminate any outstanding SETIME requests without having to set up a new one.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	SETIME	[{time-interval}] [,WAIT] [,{M}] [{(1)}] [,{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 4095_{10}. 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 canceled, 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 macroinstruction.

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.

2.3.2.2. 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 macroinstruction. When the time interval elapses, the timer island code for this task is executed. For example, the instruction:

```
1       10    16
-------------------------------------------------------------
        SETIME 30,,S
        next instruction
        .
        .
        .
```

or

```
        SETIME 30
        next instruction
        .
        .
        .
```

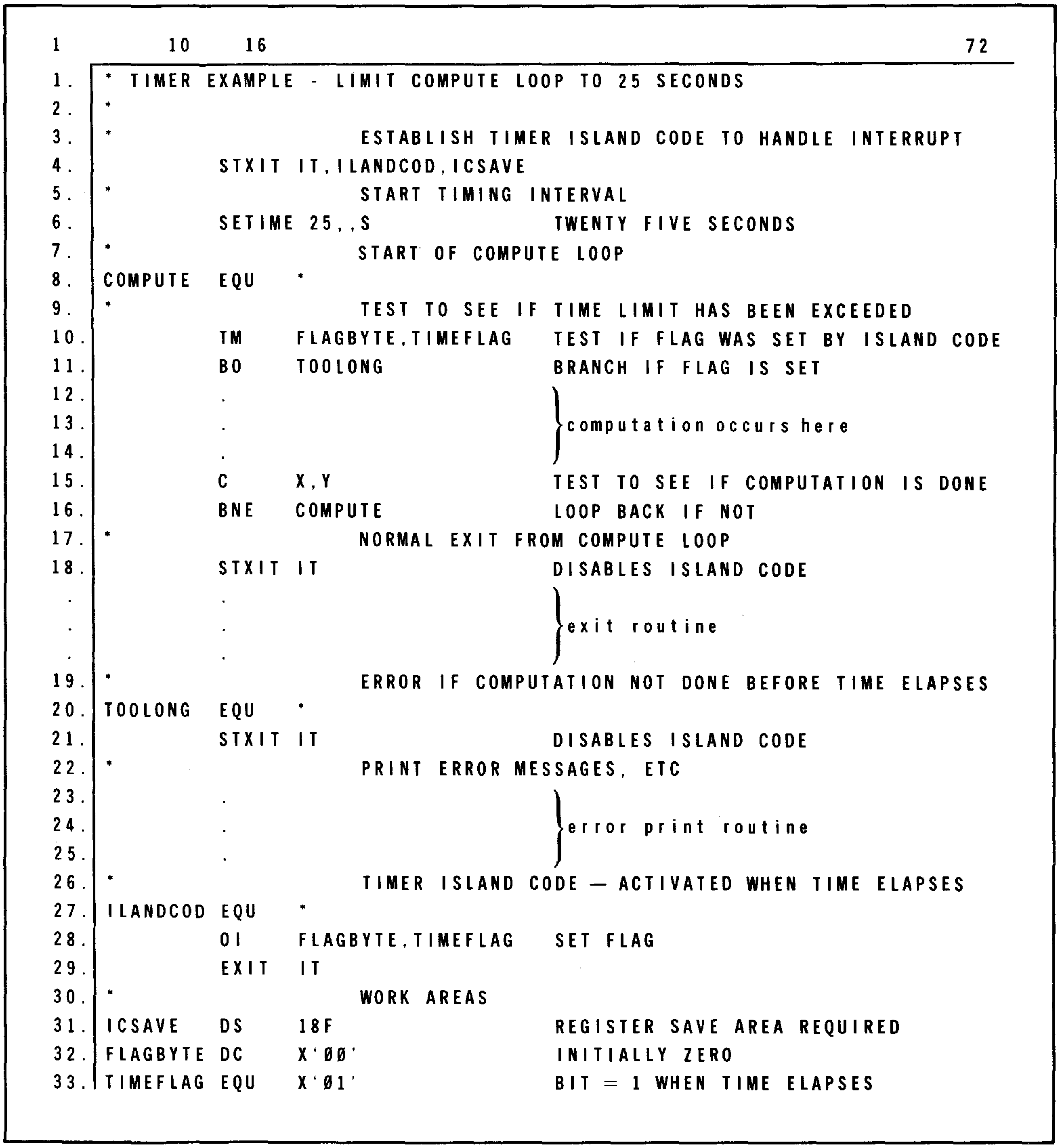
requests a timer interrupt in 30 seconds. The task continues processing until the 30-second time interval elapses; then the timer island code is executed.

If you want to specify an interval smaller than a second, the instruction:

```
        SETIME 200,,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 2-2 is an example of the use of the SETIME macroinstruction to request an interrupt in 25 seconds so that a time of 25 seconds can be placed on the computation that follows.



```
1      10     16                                                     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.          TM    FLAGBYTE,TIMEFLAG  TEST IF FLAG WAS SET BY ISLAND CODE
11.          BO    TOOLONG            BRANCH IF FLAG IS SET
12.          .                        \
13.          .                         >computation occurs here
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.          .                        \
24.          .                         >error print routine
25.          .                        /
26. *                  TIMER ISLAND CODE — ACTIVATED WHEN TIME ELAPSES
27. ILANDCOD EQU   *
28.          OI    FLAGBYTE,TIMEFLAG  SET FLAG
29.          EXIT  IT
30. *                  WORK AREAS
31. ICSAVE   DS    18F                REGISTER SAVE AREA REQUIRED
32. FLAGBYTE DC    X'ØØ'              INITIALLY ZERO
33. TIMEFLAG EQU   X'Ø1'              BIT = 1 WHEN TIME ELAPSES
```

Figure 2—2. Example of SETIME Macroinstruction

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.

2.3.2.3. 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 macroinstruction. For example, the instruction:

```
1       10     16
________________________________________________
        SETIME 3Ø,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.

2.3.2.4. Cancel a Previous Timer Interrupt Request

To cancel a previous timer interrupt request, simply use the SETIME macroinstruction without parameters. For example:

```
1.|     SETIME 3ØØ,,M
2.|     next instruction
3.|     .
4.|     .
5.|     .
6.|     SETIME
```

Line 1 requests activation of interval timer island code in 300 milliseconds. Line 6 cancels the request.

2.4. 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 macroinstructions 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 macroinstructions can also be used to pass parameters from the calling program to the called program.

2.4.1. 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.

- Registers 2 through 12 – Free registers

  These registers are never used or referenced by the direct linkage macroinstructions.

- 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.

2.4.2. Linkage Procedure

The calling program establishes direct linkage with another program by means of the CALL or VCALL macroinstruction. If registers are used in the called program (other than 0, 1, and 15), the SAVE macroinstruction 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 macroinstructions perform the linkage functions for you. Or, if you prefer, depending on how you code the parameters in the SAVE and RETURN macroinstructions, you can perform some of these functions yourself.

If you use the SA parameters in the SAVE and RETURN macroinstructions, 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 macroinstruction.

If you use the COVER and COVADR parameters in the SAVE macroinstruction, 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.

2.4.3. 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 macroinstructions. Note that this register save area is different from the save area used with island code linkage for register and PSW storage (described in 2.5).

The format of the register save area is shown in Figure 2-3 and further explained in Table 2-1.

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 2—3. Register Save Area Format

Table 2—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.

2.4.4. Call a Program (CALL/VCALL)

The CALL and VCALL macroinstructions 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 macroinstruction.

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 macroinstruction 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 macroinstruction 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 macroinstruction.

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 macroinstruction 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       10    16
-------------------------------------------------------
        CALL  TEST,(ADDR1,ADDR2,ADDR3)
        CALL  TEST,TSTADR
        CALL  SINE,(1)
        CALL  (15),(1)
```

2.4.5. Generate an Argument List (ARGLST)

The ARGLST macroinstruction generates an argument list (list of parameters) in the format required by the CALL/VCALL macroinstruction.

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

Format:

LABEL	△OPERATION△	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 macroinstruction 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:

```
TSTADR   ARGLST ADDR1,ADDR2,ADDR3
```

2.4.6. Save Register Contents (SAVE)

The SAVE macroinstruction 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] [,COVER={r (r1,r2,...,rn) 15}] [,COVADR={base-addr *}] [,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 macroinstruction, that is, the contents of the location counter at the time this macroinstruction 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        10    16
------------------------------------------------
SUB      SAVE  (14,12),,COVER=12
         SAVE  (14,12),,SA=SAVEAREA,COVER=1Ø
         SAVE  (14,12),,COVADR=SUB2,SA=AREA
```

2.4.7. Restore Registers and Return (RETURN)

The RETURN macroinstruction 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	△OPERATION△	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 macroinstruction.

SA=*

Specifies that you have defined a save area elsewhere in the routine.

If omitted, a save area is not created by this macroinstruction, and register 13 is unaltered.

Examples:

```
1       10      16
-----------------------------------------------
        RETURN  (14,12)
        RETURN  (14,12),T,SA=*
        RETURN  (14,12),,SA=SAVEAREA
```

2.5. ISLAND CODE LINKAGE

Your program may be interrupted at any time for a variety of reasons. Supervisor macroinstructions are available that let you handle four of these interrupts:

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 macroinstruction (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 or workstation.

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 macroinstructions that automatically generate the linkages between your island code routine and your program. The macroinstructions are:

- STXIT

  Attach and detach your island code routine.

- EXIT

  Exit from your island code routine.

You must use the STXIT macroinstruction in your program to attach your island code routines to your tasks. You use the EXIT macroinstruction in your program check, interval timer, and operator communication island code routines to return control to the interrupted task. Do not use the EXIT macroinstruction in the abnormal termination island code routine. Instead, use:

- a DETACH macroinstruction to detach the task;
- a DUMP or EOJ macroinstruction to terminate the job step; or
- a CANCEL macroinstruction to terminate the job.

2.5.1. Attaching Island Code to a Task (STXIT)

You use the STXIT macroinstruction 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 macroinstruction. One is for program check, abnormal termination, and interval timer island code routines; and one is for operator communication.

2.5.1.1. Attaching Program Check, Abnormal Termination, and Interval Timer Island Code

Function:

This form of the STXIT macroinstruction 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 macroinstruction when it is used for program check, abnormal termination, or interval timer island code linkage is:

LABEL	△OPERATION△	OPERAND
[symbol]	STXIT	$\left\{\begin{matrix}\text{PC}\\\text{AB}\\\text{IT}\end{matrix}\right\}\left[,\left\{\begin{matrix}\text{entry-point}\\\text{(1)}\end{matrix}\right\},\left\{\begin{matrix}\text{save-area}\\\text{(Ø)}\end{matrix}\right\}\right]$

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)

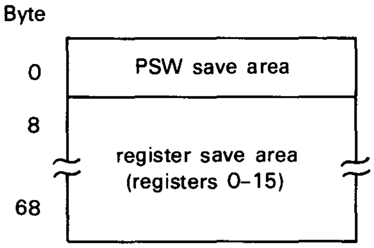
Indicates that register 1 has been preloaded with the address of the entry point.

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:



(Ø)

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 macroinstruction to detach an island code routine (2.5.2). 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 macroinstruction for program check, abnormal termination, and interval timer are shown in 2.5.5, 2.5.6, and 2.5.7.

2.5.1.2. Attaching Operator Communication Island Code

Function:

This form of the STXIT macroinstruction 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 and the operator is notified.

Format:

The format for the STXIT macroinstruction when it is used for unsolicited operator communication linkage is:

LABEL	△OPERATION△	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.

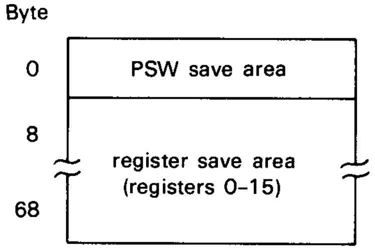
(1)
Indicates that register 1 has been preloaded with the address of a 4-word table containing parameters 2, 3, 4, and 5 in the following format:

Byte	
0	save area address
4	entry point address
8	message area address
12	message area length

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.

2.5.2. Detaching Island Code from a Task (STXIT)

Function:

This form of the STXIT macroinstruction terminates linkage between your task and the user island code routine specified by the parameter.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	STXIT	$\left\{\begin{matrix}\text{PC}\\\text{AB}\\\text{IT}\\\text{OC}\end{matrix}\right\}$

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 macroinstruction 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.

2.5.3. Island Code Entrance

As we have described earlier, you attach your island code routine with the STXIT macroinstruction, 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 macroinstruction, 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 is 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.

For abnormal termination, interval timer, or operator communication island code processing, registers saved by the system in an island code save area should never be depended upon or changed by your job. Usually it makes no difference; however, if two or more interrupts for different types of island code are received at about the same time, an island code save area may not have your PSW and registers, but rather another island code's PSW and registers. These rules do not apply to the PC island code. It is always safe to look at the PC island code PSW and register save area.

Your island code routines are given control by the task switcher even though the associated task is in a wait state. This override wait (e.g., wait for I/O synchronization, wait for interval timer, task pause, PAUSE console commands) is referred to as island code override and remains in effect during island code execution.

For example, assume that you have written an island code routine to handle operator communications, your task has issued a wait for an I/O operation, and the operator enters an unsolicited message at the system console. The island code routine is entered immediately, regardless of whether the wait has been 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.

As you know, the purpose of an island code routine is to handle an interrupt immediately. Consequently, functions that cause a task to be put in a wait state (WAIT macroinstruction and certain data management imperative macroinstructions such as GET or PUT, TYIELD macroinstruction, SETIME macroinstruction) should not be placed in an island code routine. If they are, they can defeat the purpose of the island code routine; that is, they prevent an interrupt from being handled immediately. To illustrate this, assume that you included a wait in an island code routine and it was followed by an EXIT macroinstruction. Control would be immediately transferred to the interrupted task even though the wait in the island code was still set. If another interrupt occurred, control is not transferred from your task to your island code routine until the wait within the island code routine is completed.

2.5.4. Island Code Exit (EXIT)

At the close of your island code routine, you can:

- use the EXIT macroinstruction to return control to the interrupted task; or
- use the DETACH, EOJ, DUMP, or CANCEL macroinstruction to terminate the task.

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 macroinstruction as the last executable instruction of the island code routine.

2.5.4.1. Exiting from Program Check and Operator Communication Island Code

Function:

This format of the EXIT macroinstruction is used for program check and operator communications island code. The macroinstruction terminates the 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 macroinstruction must be the last executable instruction within the island code routine.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	EXIT	$\left\{\begin{matrix}\mathtt{PC}\\\mathtt{OC}\end{matrix}\right\}$

Positional Parameter 1:

PC

Specifies that exit is from the program check island code routine.

OC

Specifies that exit is from the operator communications island code routine.

2.5.4.2. Exiting from Interval Timer Island Code

Function:

This format of the EXIT macroinstruction is used for interval timer island code. The macroinstruction terminates the user island code routine, restores the contents of the registers and the PSW, and returns program control to the point immediately following the interrupt. If positional parameter 2 is specified, the interval timer is set to the specified value before program control is returned to the task. When the specified time interval elapses, the timer island code is again executed. This macroinstruction must be the last executable instruction within the island code routine.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	EXIT	$\mathtt{IT}\left[,\left\{\begin{matrix}\mathtt{timer\text{-}interval}\\\mathtt{(1)}\end{matrix}\right\}\right]\ \left[,\left\{\begin{matrix}\mathtt{M}\\\mathtt{S}\end{matrix}\right\}\right]$

Positional Parameter 1:

IT

Specifies that exit is from the interval timer island code routine.

Positional Parameter 2:

timer-interval

Specifies the interval of time that must expire beofre the timer island code is again activated. 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 4095_{10}. To specify a value greater than 4095_{10}, enter (1) as positional parameter 1 and preload register 1 with the required time interval value.

The effect of this parameter is the same as if you had issued your own SETIME macroinstruction immediately before the EXIT. If you had done that, however, an interrupt occurring between the SETIME and EXIT macroinstructions could possibly take control away from you long enough for your SETIME to expire and cause an error (referenced island code in busy state). To avoid this possibility, you should only reset the interval timer either when the EXIT macroinstruction is issued or by a SETIME macroinstruction issued outside the timer island code.

(1)

Indicates that register 1 has been preloaded with the time interval value.

If omitted, the interval timer is reset by this macroinstruction.

Positional Parameter 3:

M

Specifies that the time interval entered as positional parameter 2 is expressed in milliseconds.

S

Specifies that the time interval entered as positional parameter 2 is expressed in seconds.

If omitted, parameter S is assumed.

2.5.4.3. 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 macroinstructions. You may use the DETACH, EOJ, DUMP, or CANCEL macroinstruction. The use of these macroinstructions to terminate abnormal termination island code is described in 2.5.6.

2.5.5. 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 macroinstruction in your program that linked the island code to the task. At this time, the least significant eight bits of register 0 contain 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 macroinstruction.

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 macroinstruction 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.

Let us look at how you would use the STXIT macroinstruction with symbolic addresses. Figure 2-4 illustrates this.

```
1        10   16
         LR    R3,R7
         .
         .
         .
         MH    R3,TRAJ
PCOVFL   STXIT PC,AROVFL,PC1
         ST    R3,MAXTRAJ
         .
         .
         .
         L     R6,MAXWGT
         .
         .
         .
AROVFL   C     R1,TESTA   \
         .                 |
         .                 >island code routine
         .                 |
         EXIT  PC         /
PC1      DS    18F
```

Figure 2—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 macroinstruction 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 2-5.

```
      1         10     16
 1.             LR     R3,R7
 2.             .
 3.             .
 4.             .
 5.             MH     R3,TRAJ
 6.             LA     R1,AROVFL
 7.             LA     RØ,PC1
 8.   PCOVFL    STXIT  PC,(1),(Ø)
 9.             ST     R3,MAXTRAJ
10.             .
11.             .
12.             .
13.             L      R6,MAXWGT
 .              .
 .              .
 .              .
59.   AROVFL    C      R1,TESTA  \
60.             .                 |
61.             .                 >island code routine
62.             .                 |
63.             EXIT   PC        /
64.   PC1       DS     18F
```

Figure 2—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 macroinstruction, 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 macroinstruction in line 8, you simply write the register numbers as shown in the format.

When the STXIT macroinstruction 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 macroinstruction 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.

2.5.6. 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 macroinstruction) 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 macroinstruction 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 macroinstruction to exit from abnormal termination island code. Instead, you may use any of the following macroinstructions 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

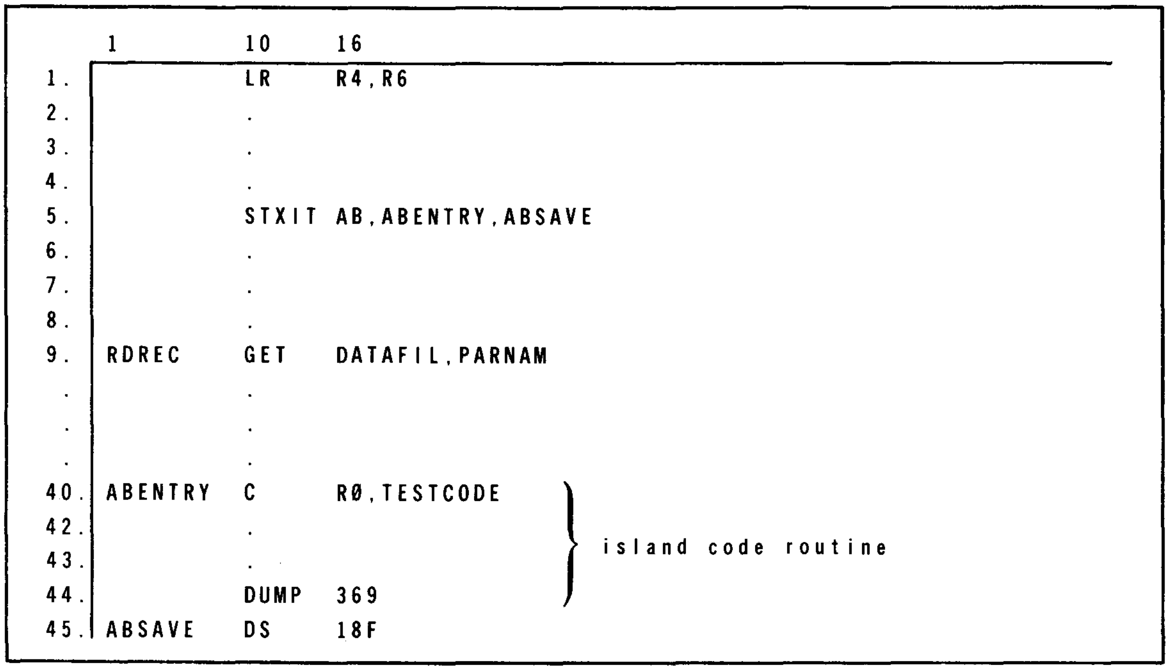
  Print out the job region contents and terminate the job abnormally.

The EOJ macroinstruction is described in 2.2.4, CANCEL in 2.2.5, DUMP in 7.1.2, and DETACH in 5.3.3.

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 2-6 is an example of how you use the STXIT macroinstruction to attach the abnormal termination island code routine to your task. Note that in this case we have chosen to use the DUMP macroinstruction to exit from the island code routine.



```
      1          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      )
42.              .                       } island code routine
43.              .                       }
44.              DUMP   369              )
45.   ABSAVE     DS     18F
```

Figure 2—6. Example of Abnormal Termination Island Code Linkage Using Symbolic Addresses

In this example, we've coded an abnormal termination STXIT macroinstruction in line 5. The entry point address is ABENTRY and the save area address is ABSAVE. As we mentioned earlier, the STXIT macroinstruction 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.

2.5.7. Interval Timer

Your interval timer island code routine receives control as the result of a timer interrupt requested by a SETIME macroinstruction in your program written without the WAIT parameter.

When the time interval specified in the SETIME macroinstruction elapses, your interval timer island code routine gains control at the entry point specified in the STXIT macroinstruction 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 macroinstruction to return to the interrupted task.

Remember, there must be a SETIME macroinstruction without the WAIT parameter to request an interval timer interrupt, and a STXIT macroinstruction 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 2-7 is an example of the use of the SETIME, STXIT, and EXIT macroinstructions with an interval timer island code routine.

```
1        10    16
* TIMER EXAMPLE — LIMIT COMPUTE LOOP TO 25 SECONDS
*
*                  ESTABLISH TIMER ISLAND CODE TO HANDLE INTERRUPT
         STXIT IT,ILANDCOD,ICSAVE
*                  START TIMING INTERVAL
         SETIME 25,,S              TWENTY FIVE SECONDS
*                  START OF COMPUTE LOOP
COMPUTE  EQU   *
*                  TEST TO SEE IF TIME LIMIT HAS BEEN EXCEEDED
         TM    FLAGBYTE,TIMEFLAG   TEST IF FLAG WAS SET BY ISLAND CODE
         BO    TOOLONG             BRANCH IF FLAG IS SET
         .                         ⎫
         .                         ⎬computation occurs here
         .                         ⎭
         C     X,Y                 TEST TO SEE IF COMPUTATION IS DONE
         BNE   COMPUTE             LOOP BACK IF NOT
*                  NORMAL EXIT FROM COMPUTE LOOP
         STXIT IT                  DETACH ISLAND CODE
         .                         ⎫
         .                         ⎬exit routine
         .                         ⎭
```

Figure 2—7. Example of Interval Timer Island Code Linkage Using Symbolic Addresses (Part 1 of 2)

```
     1        10    16
19. *                   ERROR IF COMPUTATION NOT DONE BEFORE TIME ELAPSES
20. TOOLONG  EQU   *
21.          STXIT IT                          DETACH ISLAND CODE
22. *                   PRINT ERROR MESSAGES, ETO
23.                                         )
24.                                         }error print routine
25.                                         )
26. *                  TIMER ISLAND CODE — ACTIVATED WHEN TIME ELAPSES
27. ILANDCOD EQU   *
28.          OI    FLAGBYTE,TIMEFLAG  SET FLAG
29.          EXIT  IT
30. *                   WORK AREAS
31. ICSAVE   DS    18F                REGISTER SAVE AREA REQUIRED
32. FLAGBYTE DC    X'ØØ'              INITIALLY ZERO
33. TIMEFLAG EQU   X'Ø1'              BIT = 1 WHEN TIME ELAPSES
```

Figure 2—7. Example of Interval Timer Island Code Linkage Using Symbolic Addresses (Part 2 of 2)

In this example, the SETIME macroinstruction (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 macroinstruction (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 macroinstruction is used again (lines 18 and 21) to detach the island code routine. The EXIT macroinstruction (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.

2.5.8. Operator Communication

Your operator communication island code routine receives control when the operator enters an unsolicited message at the system console or workstation. 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 or workstation, refer to the appropriate operations handbook for your system.

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 macroinstruction that linked the island code routine to the job step. At this time, register 0 contains the length (including the character under the cursor) of the message entered by the operator. Register 1 contains either a 0, indicating that the operator communication was initiated at the console, or a negative sign (–), indicating that the operator communication was initiated at the workstation. (Register 1 would not contain an ECB address because operator communication island code routines always execute under the primary task TCB.)

To exit from the operator communication island code, use the EXIT macroinstruction 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 2–8 is an example of the use of the STXIT and EXIT macroinstructions for operator communication island code routine using symbolic addresses. The general operation is similar to that described for program check. However, you will note that, in addition to the entry point and save area, the STXIT macroinstruction also specifies a message area and message length.

Following the format, the STXIT macroinstruction 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         10      16
 1.  |          LR      R3,R7
  .  |          .
  .  |          .
  .  |          .
21.  |          STXIT   OC,SYSCON,OC1,OPRMSG,6Ø
  .  |          .
  .  |          .
  .  |          .
75.  |SYSCON    LR      R5,OPRMSG+3  \
  .  |          .                     |
  .  |          .                     } island code routine
  .  |          .                     |
89.  |          EXIT    OC           /
90.  |OC1       DS      18F
91.  |OPRMSG    DS      15F
```

Figure 2—8. Example of Operator Communication Island Code Linkage Using Symbolic Addresses

Figure 2-9 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:

Byte	
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 macroinstruction if you are using symbolic addresses. Remember also that the time the operator communication interrupt occurs, this 16-byte field (OCSETUP) must contain the appropriate addresses and message area length.

```
     1        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
```

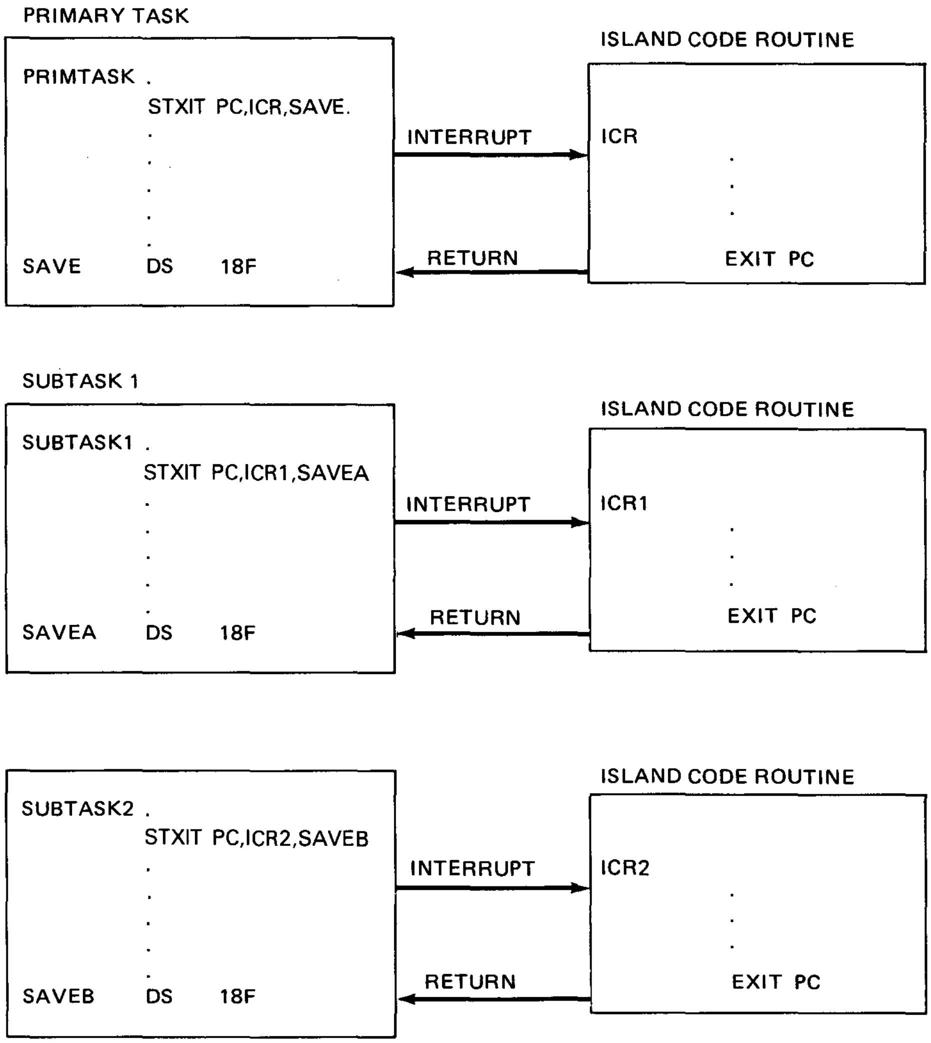
Figure 2—9. Example of Operator Communication Island Code Linkage Using Register Addresses

2.5.9. Use of Island Code with Multitasking

2.5.9.1. 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 macroinstruction in each task to link the task to its island code routine.

Figure 2-10 depicts a job step with three tasks (the primary task and two subtasks). The STXIT macroinstruction 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.



```
PRIMTASK .
         STXIT PC,ICR,SAVE.
         .
         .
         .
         .
SAVE     DS    18F
```

```
ICR
         .
         .
         .
         EXIT PC
```

```
SUBTASK1 .
         STXIT PC,ICR1,SAVEA
         .
         .
         .
         .
         .
SAVEA    DS    18F
```

```
ICR1
         .
         .
         .
         EXIT PC
```

```
SUBTASK2 .
         STXIT PC,ICR2,SAVEB
         .
         .
         .
         .
         .
SAVEB    DS    18F
```

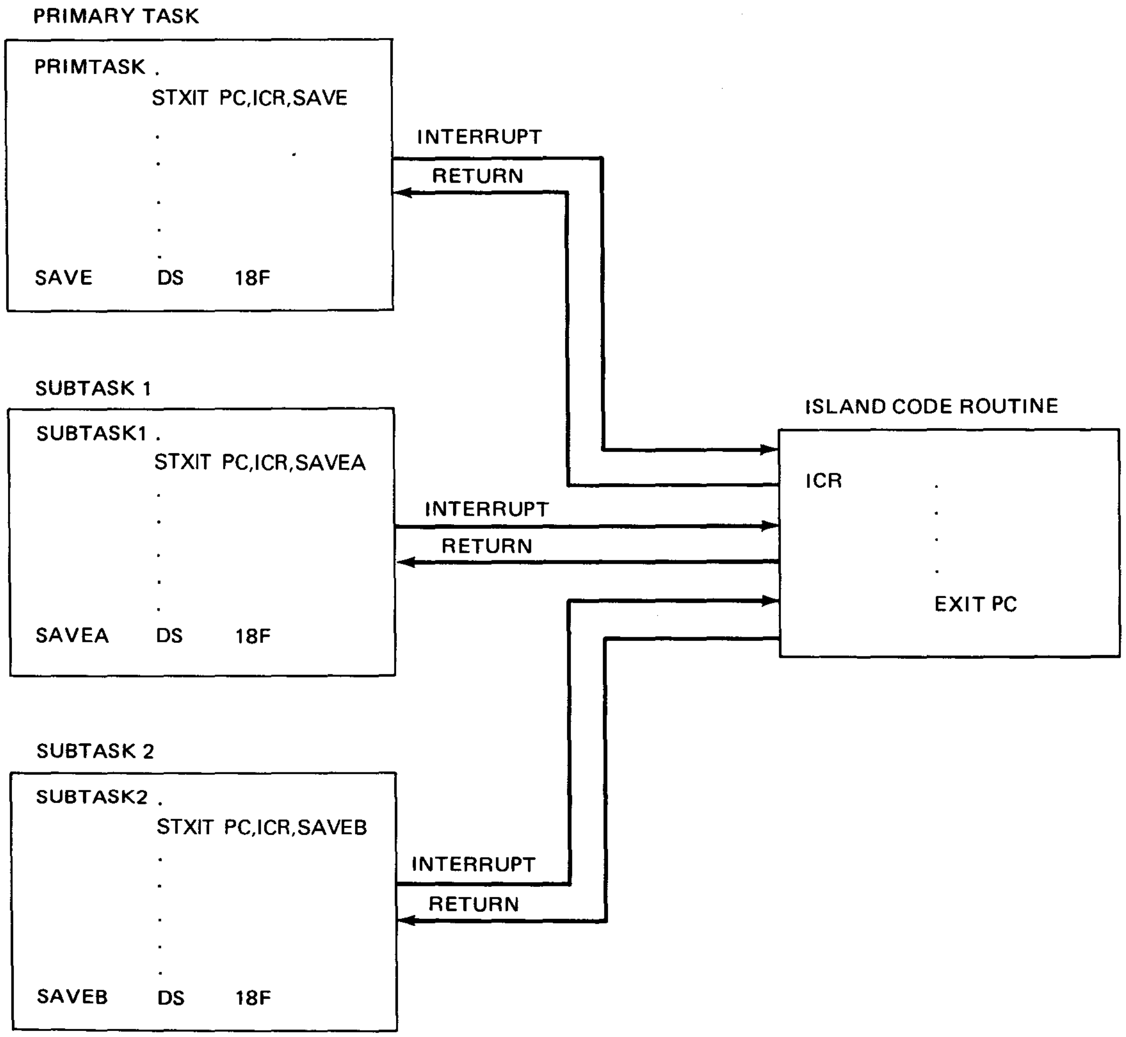
```
ICR2
         .
         .
         .
         EXIT PC
```

Figure 2—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 macroinstruction 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 macroinstruction. 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 2-11 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 macroinstruction 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.



```
PRIMTASK .
         STXIT PC,ICR,SAVE
         .
         .
         .
         .
         .
SAVE     DS    18F
```

```
SUBTASK1 .
         STXIT PC,ICR,SAVEA
         .
         .
         .
         .
         .
SAVEA    DS    18F
```

```
SUBTASK2 .
         STXIT PC,ICR,SAVEB
         .
         .
         .
         .
         .
SAVEB    DS    18F
```

```
ICR      .
         .
         .
         .
         EXIT PC
```

Figure 2—11. Example of Common Program Check Island Code for All Tasks in a Job Step

2.5.9.2. 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 macroinstruction 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 macroinstruction is effective when an interrupt occurs. In this case, each succeeding STXIT macroinstruction supersedes the previous one, and you do not have to issue a STXIT macroinstruction to detach each previous island code routine.

2.5.9.3. 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 macroinstruction 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 macroinstruction is effective when an interrupt occurs.

2.6. 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 macroinstruction or set using the PUTCOM macroinstruction. 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 macroinstructions 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.

2.6.1. Get Data from Communication Region (GETCOM)

Function:

The GETCOM macroinstruction 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.

2.6.2. Put Data into Communication Region (PUTCOM)

Function:

The PUTCOM macroinstruction 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.

2.6.3. Get Data from System Control Tables (GETINF)

Function:

The GETINF macroinstruction 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 workarea in main storage specified in positional parameter 2.

NOTE:

If you use the GETINF macroinstruction in your program, you must reassemble your program upon every major release of the system software.

Format:

LABEL	△OPERATION△	OPERAND

[symbol]	GETINF	{TCB},{work-area},number-of-bytes,displacement
		{SIB} {(1)}
		{PRE}
		{PUB}

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 must specify work-area, not (1).

Positional Parameter 2:

work-area
: Specifies the symbolic address of the workarea 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 workarea.

Not valid if positional parameter 1 is PUB because register 1 already contains the address of the PUB or 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.

2.6.4. Move Data from LDA (GETLDA)

Function:

The GETLDA macroinstruction allows you to read data from the local data area (LDA) (set up by job control or OS/3 interactive (OS/3I) LOGON processing) into the user supplied buffer.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	GETLDA	⎧address⎫[,disp][,length] ⎩(1)⎭

Positional Parameter 1:

`address`
Specifies the address in the user program to which the data is transferred.

`(1)`
Indicates register 1 is preloaded with the address of an 8-byte parameter area as follows:

Bytes 0 – 3 = address of user buffer
Bytes 4 – 5 = displacement into LDA
Bytes 6 – 7 = length of transfer

Positional Parameter 2:

`disp`
Specifies the local data area displacement after data transfer.

Positional Parameter 3:

`length`
Specifies the number of bytes to be transferred. The maximum is 256 bytes. The default is 1.

2.6.5. Move Data to LDA (PUTLDA)

Function:

The PUTLDA macroinstruction allows you to transfer data to LDA from the user supplied buffer.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	PUTLDA	{address}[,disp][,length] {(1)}

Positional Parameter 1:

address

Specifies the address in the user program from which the data is transferred.

(1)

Indicates that register 1 is preloaded with the address of an 8-byte parameter area as follows:

Bytes 0 – 3 = address of user buffer
Bytes 4 – 5 = displacement into LDA
Bytes 6 – 7 = length of transfer

Positional Parameter 2:

disp

Specifies the LDA displacement for data transfer.

Positional Parameter 3:

length

Specifies the number of bytes to be transferred. The maximum is 256 bytes The default is 1.

2.7. 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 or diskette file, nor is a device assignment set required for the card reader or diskette subsystems.

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 run library file. During the execution of the job step, this file is read into main storage and may be accessed by GETCS macroinstructions 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 macroinstruction.

Except for PARAM statements, each retrieved record is an exact image of the source statement, which may be from 1 to 128 bytes. Thus, you can read 80-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.

2.7.1. 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             .
              .
           /*
           //PARAM
           /$
Data          .
Set           .
2             .
              .
           /*
           /$
Data          .
Set           .
3             .
              .
           /*
           /$
              .
              .
              .
Data       /$       (Embedded /$ image)
Set           .
4             .
              .
           /*       (Embedded /* image)
              .
              .
              .
           /*       (Real /* image)
```

2.7.2. Reading Embedded Data

If you are reading one record at a time from this embedded data file, the first GETCS macroinstruction 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 macroinstruction, program control is returned to the issuing program at the point immediately following the GETCS macroinstruction. Registers 0 and 1 will contain:

RO - The binary count of records retrieved.

- 0_{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.

- 80000nnn if an error occurred while processing GETCS. (nnn is the error code; refer to the system messages manual, UP-8076 (current version), for a list of possible error codes).

R1 - The reread pointer (2.7.4). When passed to the SETCS macroinstruction (2.7.5), it allows you to reread embedded data at this pointer.

If two or more records are requested by a single GETCS macroinstruction, the first occurrence of a real /* image terminates the control stream reader function. The /* is not returned until the next GETCS macroinstruction call, at which time register 0 is set to zero and register 1 contains the reread pointer. On subsequent GETCS macroinstruction calls, the /* image is always returned; however, control is not returned to the error address specified because it is not an error.

Because control streams may themselves be embedded data, the GETCS macroinstruction indicates the end of a data set by signaling 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.

2.7.3. Get File from Control Stream (GETCS)

Function:

The GETCS macroinstruction 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. Except for PARAM statements, each retrieved record is an exact image of the source statement. PARAM statements appear according to standard JCL conventions.

Images are read sequentially from the start of the entire data file. You can alter the sequence or reread data by using the SETCS macroinstruction.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	GETCS	{input-area | (1)} [{number-of-records | (Ø) | 1}] [,{error-addr | (r)}] [,{n | 80}]

Positional Parameter 1:

input-area

Specifies the symbolic address of an input area in main storage that is to receive the record 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.

(Ø)

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.

2.7.4. Rereading Embedded Data

Following execution of a GETCS macroinstruction, 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 macroinstruction. A succeeding GETCS macroinstruction 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 macroinstruction), 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 macroinstruction has just returned five images, the reread pointer would point to the first image in the data file, not the fifth.

2.7.5. Reset Control Stream Reader (SETCS)

Function:

The SETCS macroinstruction alters the sequence in which a subsequent GETCS macroinstruction 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	$\left\{\begin{array}{l}\text{pointer}\\ \text{data-set-no}\\ (1)\\ \text{NEXT}\end{array}\right\}\left[,\left\{\begin{array}{l}\text{R}\\ \text{S}\end{array}\right\}\right]\left[,\left\{\begin{array}{l}\text{error-addr}\\ (\text{r})\end{array}\right\}\right]$

Positional Parameter 1:

pointer

Specifies the symbolic address of a full word embedded data file pointer provided by a previous GETCS macroinstruction.

Upon successful completion of a GETCS macroinstruction, control is returned to the program at the point immediately following the GETCS macroinstruction, 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 macroinstruction, 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 macroinstructions 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 macroinstructions 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 macroinstruction.

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.

2.7.6. Minimizing Disk Accesses

The job control stream embedded data reader operates as a transient within the supervisor and is called by the GETCS macroinstruction. 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.

3. Disk and Diskette Space Management

3.1. GENERAL

Space management comprises a set of routines that provide an efficient and completely automatic disk and diskette space accounting capability. Job control, by using these routines, can:

- allocate space to a new disk (format label) or diskette (format or data set label) file with the EXT job control statement;
- extend a disk or diskette format label file with the EXT job control statement;
- scratch a disk or diskette file with the SCR statement; and
- rename a disk or diskette (format label only) file using the REN statement.

More information on these capabilities can be found in the current version of the job control user guide, UP-8065.

In addition to these functions, space management provides you with the OBTAIN macroinstruction for the retrieval of volume and file information from a disk or diskette volume. For disk and format label diskette volumes, this information is contained in the volume table of contents (VTOC), a permanently allocated, unmovable file that exists on every volume. The VTOC is addressed by the standard volume label and is created by the 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 consolidated data management concepts and facilities, UP-8825 (current version) for the formats and description of the VTOC and disk/diskette format file labels.

For data set label diskette volumes, volume and file information is contained in the index track.

3.2. ACCESS DISK/DISKETTE FORMAT LABEL FILE VTOC USER BLOCK (OBTAIN)

Function:

The OBTAIN macroinstruction 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	△OPERATION△	OPERAND
[symbol]	OBTAIN	$\left\{\begin{matrix}\text{param-list}\\(1)\end{matrix}\right\}\left[,\left\{\begin{matrix}\text{error-addr}\\(r)\end{matrix}\right\}\right]\left[,\left\{\begin{matrix}\text{vol-seq-no}\\1\end{matrix}\right\}\right]$ [,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:

Code	Interpretation
00	VOL1 address in form Occchhrr
01	Format 1 address in form Occchhrr
02	Format 2 address in form Occchhrr
03	Format 3 address in form Occchhrr
04	Format 4 address in form Occchhrr
05	Format 5 address in form Occchhrr
06	Format 6 address in form Occchhrr
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 format 1-6 label record at the disk address that is in the first word of the buffer in the form Occchhrr

NOTE:

Addresses in the form Occchhrr 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 8 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 0ccchhrr) 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        10    16
_______________________________________
         OBTAIN (1),(4),2
         OBTAIN RECOVER1,ERRRTN
```

3.3. OBTAIN DISKETTE DATA SET LABEL INFORMATION (OBTAIN)

Function:

The OBTAIN macroinstruction 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.

3.4. SPACE MANAGEMENT ERROR CODES

Errors that occur during processing of the OBTAIN macroinstruction 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 macroinstructions). 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.

4. System Access Technique Macroinstructions

4.1. GENERAL

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 the SAT operation or to construct your own handler modules.

Whenever *disk file* is used in the following discussion, it refers to both disk and format label diskette files. A format label diskette file is treated exactly the same as any other disk file.

SAT techniques and macroinstructions that define and control disk files are described in 4.2; SAT techniques and macroinstructions that define and control tape files are described in 4.5.

4.2. 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 macroinstruction, which generates a partition control appendage to the DTF file table. Up to seven partitions may be defined within a single file.

4.2.1. 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 4-1) 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 macroinstruction, 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 macroinstructions 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.

<table>
<tr><th>BYTE</th><th>0</th><th>1</th><th>2</th><th>3</th></tr>
<tr><td>0</td><td colspan="4">current ID</td></tr>
<tr><td>4</td><td colspan="4">max relative block</td></tr>
<tr><td>8</td><td colspan="4">logical blocks/track</td></tr>
<tr><td>12</td><td>PCA ID</td><td colspan="3">EOD ID</td></tr>
<tr><td>16</td><td>I/O count</td><td colspan="3">IOAREA/address</td></tr>
<tr><td>20</td><td colspan="2">block size</td><td>reserved</td><td>sectors/block</td></tr>
<tr><td>24</td><td colspan="2">lace factor/key length</td><td colspan="2">unit of store</td></tr>
<tr><td>28</td><td colspan="4">DTF address</td></tr>
<tr><td>32</td><td>PCA flags</td><td colspan="3">EOD address</td></tr>
</table>

PCA FLAGS

Bit		Bit	
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 4—1. Partition Control Appendage (PCA) Table Format

When searching by key (READE and READH macroinstructions), you must know the relative address of at least one block on the track you want 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 want 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 macroinstruction, 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 macroinstruction, the address contained in the current ID field will be the block just read plus 1.

When using the SEEK macroinstruction, 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.

4.2.2. Block Addressing by Key

Blocks are addressed either by key or relative ID. You create a partition using keyed data blocks (Figure 4-2) by specifying the KEYLEN keyword parameter of the PCA declarative macroinstruction. The key is placed in the first part of the I/O buffer area and is left-justified; when the PUT macroinstruction 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 macroinstruction, 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 macroinstruction 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.

4.2.3. 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 want to access, then issue a GET macroinstruction to read the block or a PUT macroinstruction to write the block.

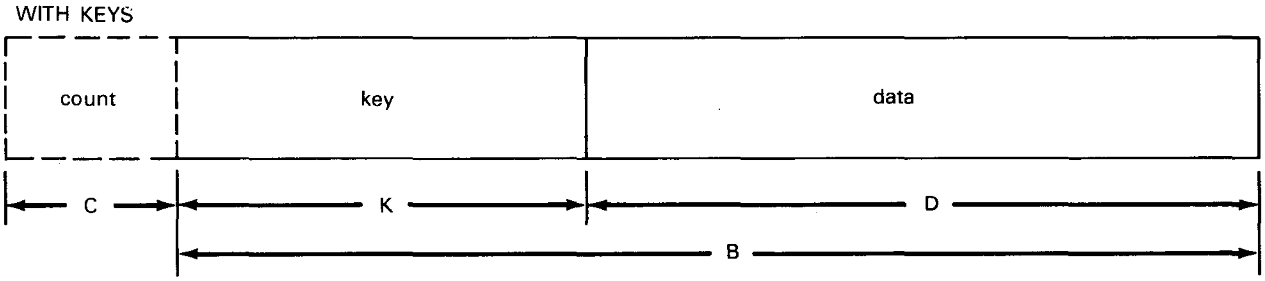
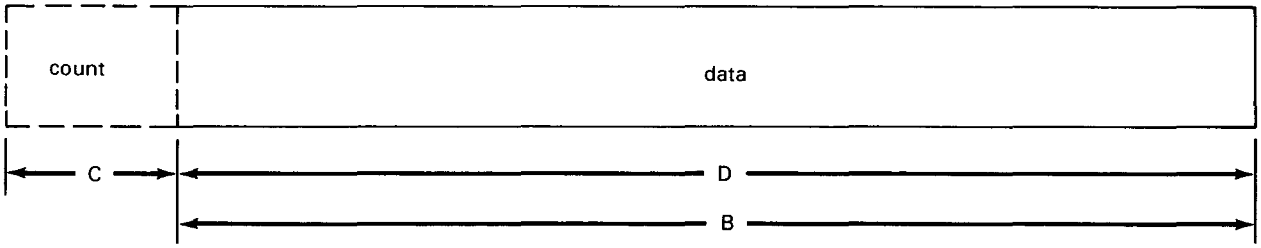


Figure 4—2. Record Formats for Disk Devices (Part 1 of 2)

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 4—2. Record Formats for Disk Devices (Part 2 of 2)

4.2.4. 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 macroinstruction. This is represented as a percentage value of the overall file.

To calculate the SIZE entry, use the following formula:

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

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 macroinstruction.

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 macroinstruction for the partition.

Partition No.	BLKSIZE	Percentage	Result	SIZE
1	1024	40	40960	67
2	256	50	12800	21
3	768	10	7680	12
		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 macroinstruction 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 specify 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 macroinstruction references a block beyond the current maximum block address for the partition. If the new allocation cannot satisfy the current PUT macroinstruction demands, an error will be indicated. However, partitions will not be extended beyond the volume on which the file resides.

4.2.5. 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 macroinstruction. When the file is opened by the OPEN macroinstruction, 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 4.2.5.2.

Figure 4-3 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)

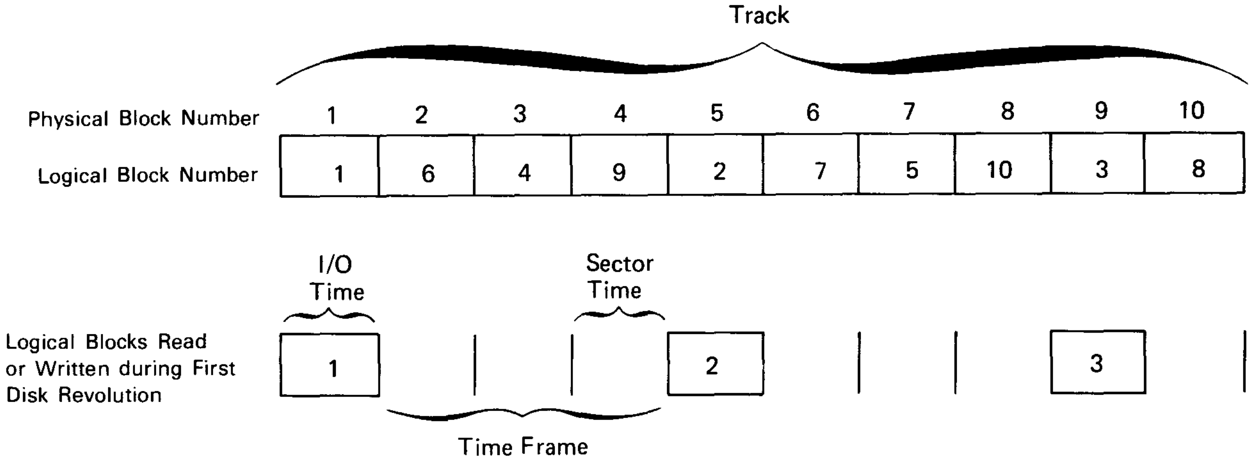
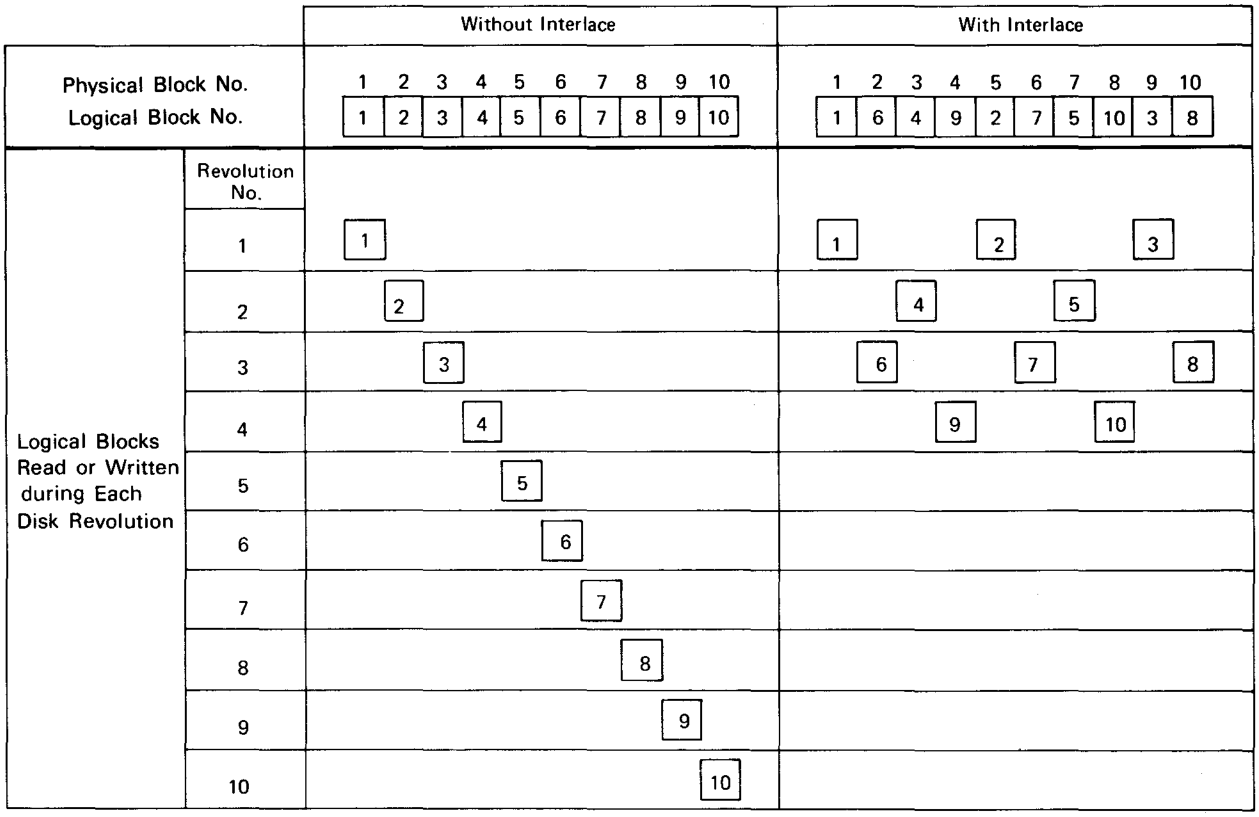


Figure 4—3. Definition of Interlace Variables

4.2.5.1. Interlace Operation

Figure 4-4 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 Interlace										With Interlace									
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 4—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 fourth 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 4.2.5.2. Although the formula is based on the use of a typical 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 macroinstruction, 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.

4.2.5.2. 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 macroinstruction 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.

4.2.6. Accessing Multiple Blocks

When you are engaged in sequential processing (SEQ=YES specified in PCA macroinstruction), you can read or write more than one block with each SAT imperative macroinstruction 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 macroinstruction. 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 4-5) will contain the number of buffers not acted upon.

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

Figure 4—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 4—5. Define the File (DTF) Table Format (Part 2 of 2)

Normally, SAT makes a single reference to the physical input/output control system (PIOCS) 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 PIOCS to access the next track. For interlaced files, SAT makes one reference to PIOCS 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.

4.3. DISK SAT FILE INTERFACE

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

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

4.3.1. Define a New File (DTFPF)

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 (DTFPF) macroinstruction which creates a table in main storage (Figure 4-5) that can be referenced by the system.

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

Format:

LABEL	△OPERATION△	OPERAND
filename	DTFPF	PCA1=partition-name [. . . ,PCA7=partition-name] [,ACCESS={ EXC EXCR SRDO SRD }] [,ALINE=YES] [,ERROR=symbol] [,EXTENTS=n] [,FCB=YES] [,LIBUP=YES] [,WAIT=YES]

The DTFPF macroinstruction 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 DTFPF macroinstruction contains only the required partition names (one for each PCA macroinstruction) 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 DTFPF macroinstruction, assume a predetermined value or condition (default). These keyword parameter defaults are as follows:

Keyword	Default
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 DTFPF macroinstruction.
FCB	The file control block (FCB), which controls file I/O, is placed into the transient area of main storage during the 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 macroinstruction 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 DTFPF macroinstruction.

- 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 DTFPF macroinstruction 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 added the number of extents allocated (to the file) to the number of partitions in the file.

  When standard system libraries are being accessed, 19 extents are recommended to be specified as EXTENTS=19.

- 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 macroinstruction is issued. However, you may place an FCB in the I/O area specified in the PCA macroinstruction for the first or only partition of the file. This area address is specified by the IOAREA1 keyword parameter of the PCA macroinstruction.

- 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. ·

- Normally, you must issue a WAITF macroinstruction after each I/O function to assure completion of the input or output operation and to set particular status bytes in the DTFPF 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 macroinstruction.

4.3.1.1. 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-8839 (current version)) specifies the type of filelocks available and the types of files affected. The LIBUP or ACCESS parameter in your DTFPF macroinstruction 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 DTFPF.

For SAT disk files, if the LIBUP or ACCESS DTF keyword parameters are not used, the file is locked as read only. Therefore, any attempt to output to the file results in a DM14 error message (invalid imperative macro/macro sequence issued). To avoid this situation and to prevent reassembling the DTF, you can place a //DD ACCESS=EXC job control statement anywhere between the DVC and LFD statements for the file in question. This statement, at open time, causes the file to be marked as exclusively dedicated to the requesting DTF and thereby removes any restrictions as to the type of operation you can perform while preventing concurrent use of the file by other jobs.

4.3.1.2. 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 DTFPF, the ACCESS parameter supersedes LIBUP. The ACCESS options can only be used if FILELOCK=SHARE 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.

4.3.2. 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 macroinstruction, the characteristics of each partition appendage must be described. This is done by using the partition control appendage (PCA) macroinstruction.

This is a declarative macroinstruction 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 macroinstruction is the same as that assigned by the PCAn keyword parameter in the DTFPF macroinstruction. 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:

Keyword Default

EODADDR When the GET macroinstruction accesses the block with the relative block number equal to the end of data ID for that partition, SAT assumes that 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 8430 and 8433 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.

Keyword	Default
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 macroinstructions).
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 macroinstruction 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, such as the FORMAT and LACE keyword parameters, because you cannot use format write (WRITE ID) and interlace simultaneously.

- Some are required together, such as the SEQ and EODADDR keyword parameters.

- The LACE keyword parameter must be specified for interlace files. The lace factor is adjusted by SAT for all 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 macroinstruction keywords are as follows:

- When specifying blocksize (BLKSIZE keyword parameter), also specify the size of the I/O area. When using sectorized disk subsystems, specify this value in multiples of 256 because this is the size of the fixed sectors. 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 macroinstruction 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.

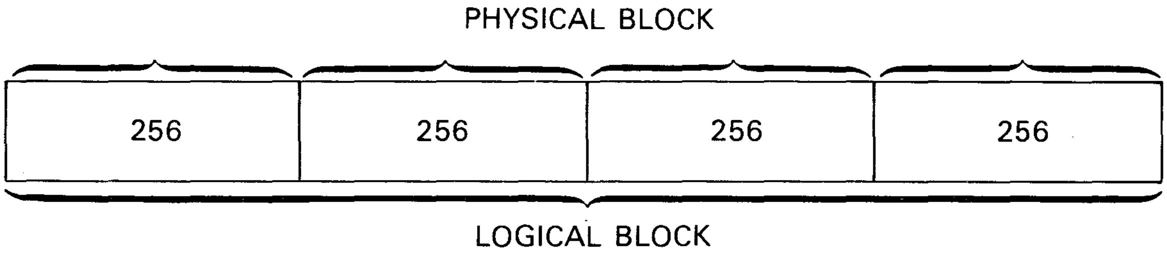
- If loading your file on a device where the space allocated is not preformatted (FORMAT=NO), a format write command is issued by SAT for each PUT macroinstruction that references a relative block number equal to the end-of-data address of the partition being accessed. A data write command is issued by SAT for each PUT macroinstruction 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 by SAT to make all disk subsystems conform to a similar access pattern. A thorough discussion of interlace operation and computation is provided in 4.2.5.

- 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 macroinstruction is issued, the open transient routine sets the current ID field to relative block 1 of the partition. Each subsequent GET or PUT macroinstruction 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 macroinstruction 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 macroinstruction 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 want 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 parameter to specify the length (3 to 255 bytes) of the key field in formatted records.

4.3.3. Processing Partitioned SAT Files

Once you have established your file on disk (that is, you have issued DTFPF and PCA macroinstructions to describe and name your file), use the imperative macroinstructions to open, control, and close your file processing. These macroinstructions 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 subsections give a brief functional description of these imperative macroinstructions. This description is followed by listing these macroinstructions in 4.4 and includes a detailed description of their parameters and characteristics.

4.3.3.1. Processing Blocks by Key

Macroinstruction	Function
OPEN	Initiates the open transient routine and identifies the file (as listed in the DTFPF macroinstruction) to be processed.
PUT	Identifies the file and partition to be accessed. Issues the write for the indicated block.

Macroinstruction Function

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.

4.3.3.2. Processing by Relative Block Number

Macroinstruction Function

OPEN Initiates the open transient routine and identifies the file (as listed in the DTFPF macroinstruction) 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.

4.4. CONTROLLING YOUR DISK FILE PROCESSING

After you have specified the details of the file and partition you wish to access through the declarative macroinstructions, the imperative macroinstructions described in the following subsections actually control your file accessing. The sequence of these macroinstructions for a particular type of processing is listed in 4.3.3.1 and 4.3.3.2, together with a brief description of their function.

4.4.1. Open a Disk File (OPEN)

Function:

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

Format:

LABEL	△OPERATION△	OPERAND

[symbol]	OPEN	{filename-1[,...,filename-n]}
		{(1)}

Positional Parameter 1:

filename-1

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

(1)

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

Positional Parameter n:

filename-n

Successive entries specify the symbolic addresses of the DTFPF macroinstructions 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.

Multiple open should be used to open more than one file when filelock is involved to prevent a lockout between two programs contending for the same file. If any one file on an OPEN macroinstruction cannot be opened because of lock, then none of the files will be opened. In such a case, if an error address had been specified in the DTF of the first file that failed, control returns to that error address. An 88 (lock failure) occurs in the DTF error code (byte 56 of the DTF file table). If no error address was specified, all files specified by the OPEN macroinstruction are deactivated pending the closing of those files by the locking program. This also produces a DM88 (writing for lock) console message.

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

The transient routine called by the OPEN macroinstruction 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.

4.4.2. Retrieve Next Logical Block (GET)

Function:

The GET macroinstruction 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 macroinstruction by the keyword parameter IOAREA1.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	GET	$\left\{\begin{matrix}\text{filename}\\ \text{(1)}\end{matrix}\right\},\left\{\begin{matrix}\text{PCA-name}\\ \text{(Ø)}\end{matrix}\right\}$

Positional Parameter 1:

filename

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

(1)

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

Positional Parameter 2:

PCA-name

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

(Ø)

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

PCA Table Content:

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

When a GET macroinstruction is issued for a SAT file, the contents of registers 14, 13, and 12 are saved in three consecutive full words whose address is in register 13.

4.4.3. Output a Logical Block (PUT)

Function:

The PUT macroinstruction 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 macroinstruction by the keyword parameter IOAREA1.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	PUT	$\left\{\begin{array}{l}\text{filename}\\(1)\end{array}\right\},\left\{\begin{array}{l}\text{PCA-name}\\(Ø)\end{array}\right\}$

Positional Parameter 1:

filename

Specifies the symbolic address of the DTFPF macroinstruction in the program corresponding to the file being written.

(1)

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

Positional Parameter 2:

PCA-name

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

(Ø)

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

PCA Table Content:

The OPEN macroinstruction 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 macroinstruction is used, the current ID field will be updated after each PUT macroinstruction 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.

When a PUT macroinstruction is issued for a SAT file, the contents of registers 14, 13, and 12 are saved in three consecutive full words whose address is in register 13.

4.4.4. Wait for Block Transfer (WAITF)

Function:

The WAITF macroinstruction ensures that a command initiated by a preceding GET, PUT, READE, or READH macroinstruction 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 macroinstruction, the WAITF macroinstruction must be issued after a GET, PUT, READE, or READH macroinstruction and before another imperative macroinstruction is issued for that file.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	WAITF	$\left\{\begin{matrix}\text{filename}\\ \text{(1)}\end{matrix}\right\}$

Positional Parameter 1:

filename

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

(1)

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

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

Function:

The READE and READH macroinstructions initiate a search by key for a block having a key equal to the key specified (READE) or equal to or greater than the key specified (READH).

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	$\begin{Bmatrix}\text{READE}\\\text{READH}\end{Bmatrix}$	$\begin{Bmatrix}\text{filename}\\(1)\end{Bmatrix},\begin{Bmatrix}\text{PCA-name}\\(Ø)\end{Bmatrix}$

Positional Parameter 1:

filename
Specifies the symbolic address of the DTFPF macroinstruction in the program corresponding to the file being processed.

(1)
Indicates that register 1 has been preloaded with the address of the DTFPF macroinstruction.

Positional Parameter 2:

PCA-name
Specifies the symbolic address of the PCA macroinstruction associated with the partition to be accessed.

(Ø)
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 macroinstruction, the current ID field in the PCA table will be the relative block number plus 1.

4.4.6. Access a Physical Block (SEEK)

Function:

The SEEK macroinstruction 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	△OPERATION△	OPERAND
[symbol]	SEEK	$\begin{Bmatrix}\text{filename}\\ \text{(1)}\end{Bmatrix},\begin{Bmatrix}\text{PCA-name}\\ \text{(Ø)}\end{Bmatrix}$

Positional Parameter 1:

filename
: Specifies the symbolic address of the DTFPF macroinstruction in the program corresponding to the file being accessed.

(1)
: Indicates that register 1 has been preloaded with the address of the DTFPF macroinstruction.

Positional Parameter 2:

PCA-name
: Specifies the symbolic address of the PCA macroinstruction associated with the partition to be accessed.

(0)
: Indicates that register 0 has been preloaded with the address of the PCA macroinstruction.

4.4.7. Close a Disk File (CLOSE)

Function:

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

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CLOSE	filename-1[,...,filename-n] (1) *ALL

Positional Parameter 1:

filename-1

Specifies the symbolic address of the DTFPF macroinstruction in the program corresponding to the file to be closed.

(1)

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

*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 DTFPF macroinstructions in the program corresponding to the additional files to be closed.

4.5. 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 PIOCS 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 macroinstructions. You use the SAT and TCA declarative macroinstructions to define the characteristics of the file and the data management technique to be used to process the file. The SAT macroinstruction creates the DTF table for the file, and the TCA macroinstruction creates the appendage to the table. These macroinstructions are described in 4.8. You use the OPEN, GET, PUT, CNTRL, WAITF, and CLOSE imperative macroinstructions to control file processing. These are described in 4.9.

All files processed by TSAT are written in a forward direction, and can be read forward and backward. The CNTRL macroinstruction 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 4.6) and tape volume and file organization conventions (described in 4.7).

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 4.10.

4.6. SYSTEM STANDARD TAPE LABELS

Magnetic tapes may be labeled or unlabeled, and a labeled tape may contain either standard or nonstandard labels. You indicate this using the FILABL parameter in the TCA macroinstruction. TSAT assumes that tapes have standard labels. If nonstandard labels exist on input files, TSAT bypasses them.

All standard tape labels are in blocks of 80 bytes and are always recorded at the same density as the data. The first three bytes of each label identify the type, and the fourth byte indicates its position within the group. For example, VOL1 indicates this is the first volume label for this file.

For block numbered tapes, each label includes a 3-byte block number field as the first three bytes of the label, making the label 83 bytes long.

There are five tape label groups; three are required and two are optional. The tape label groups are:

■ Volume label group	VOL
■ File header label group	HDR
■ User header label group (optional)	UHL
■ File trailer label group	EOF or EOV
■ User trailer label group (optional)	UTL

TSAT does not process user header (UHL) or user trailer (UTL) labels. No provision is made for creating these labels on output files; if they exist on input files, TSAT bypasses them.

TSAT label processing is limited to one volume label (VOL1), two file header labels (HDR1 and HDR2), and two file trailer labels (EOF1 and EOF2 or EOV1 and EOV2). No provision is made for creating additional labels on output files; if they exist on input files, TSAT bypasses them.

Tape label formats for block numbered files are shown in Figures 4–17 through 4–21. Tape label formats for files without block numbers are shown in Figures 4–6 through 4–10 and are described in the following subsections.

4.6.1. Volume Label Group

A volume label group consists of a single volume label (VOL1). The VOL1 label identifies the tape reel, 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 4-6, 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 labeled file, INIT 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 macroinstruction 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.

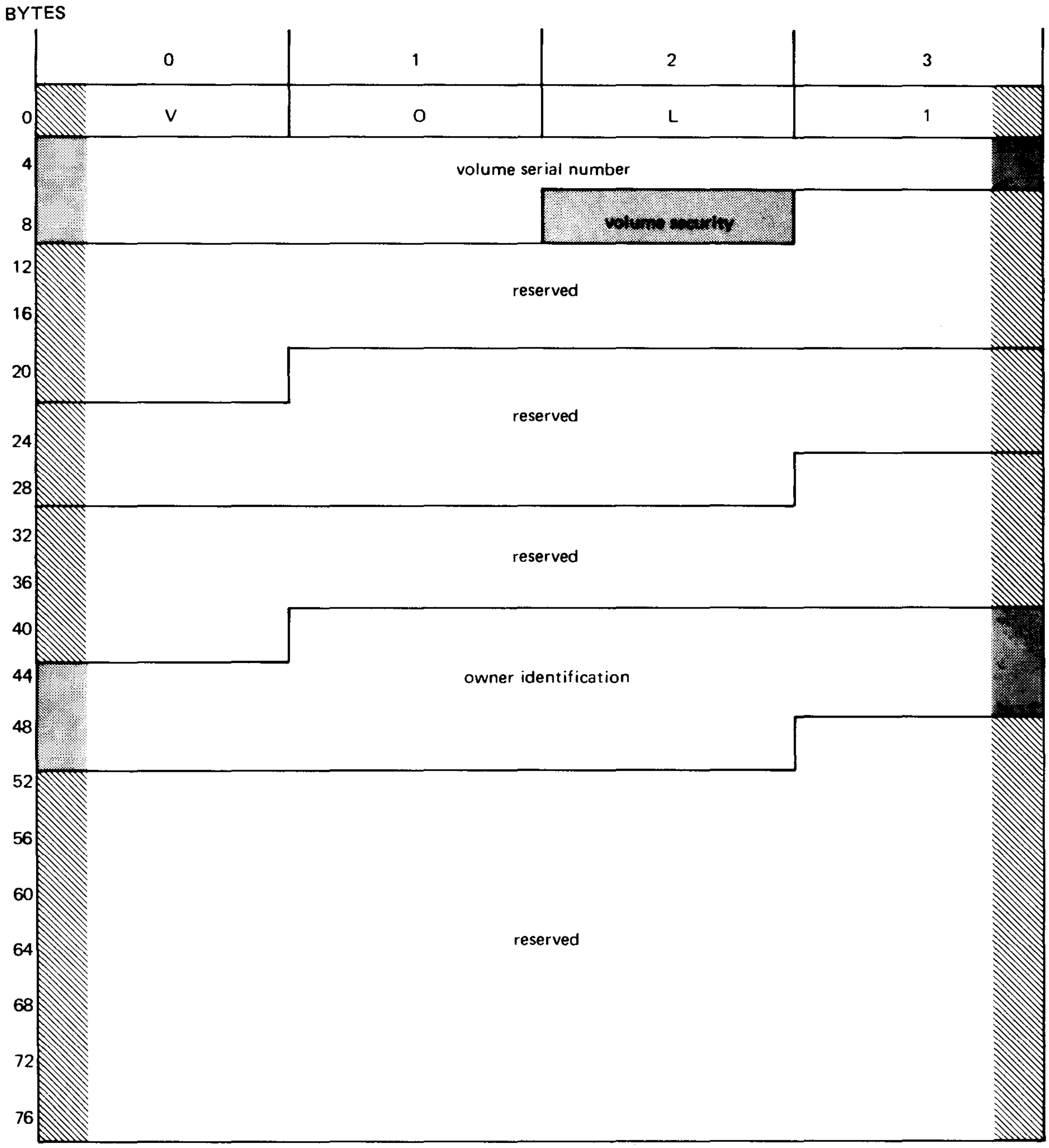
If the output tape is not at the load point after the open-and-rewind options are performed, TSAT assumes that the tape is positioned between the two ending tape marks of the previous file or just prior to the HDR1 label of an existing file. In either case, no volume label checking or creation is performed.

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 macroinstruction for an input tape and you have specified READ=BACK in the TCA macroinstruction, the label is bypassed without processing.

The format of the volume label is shown in Figure 4-6. The fields are described in Table 4-1.



LEGEND:

▨ Generated by TSAT or reserved for system expansion.

░ Written by TSAT from user-supplied data.

Figure 4—6. Tape Volume 1 (VOL1) Label Format for an EBCDIC Volume

Table 4—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	-----	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.

4.6.2. File Header Label Group

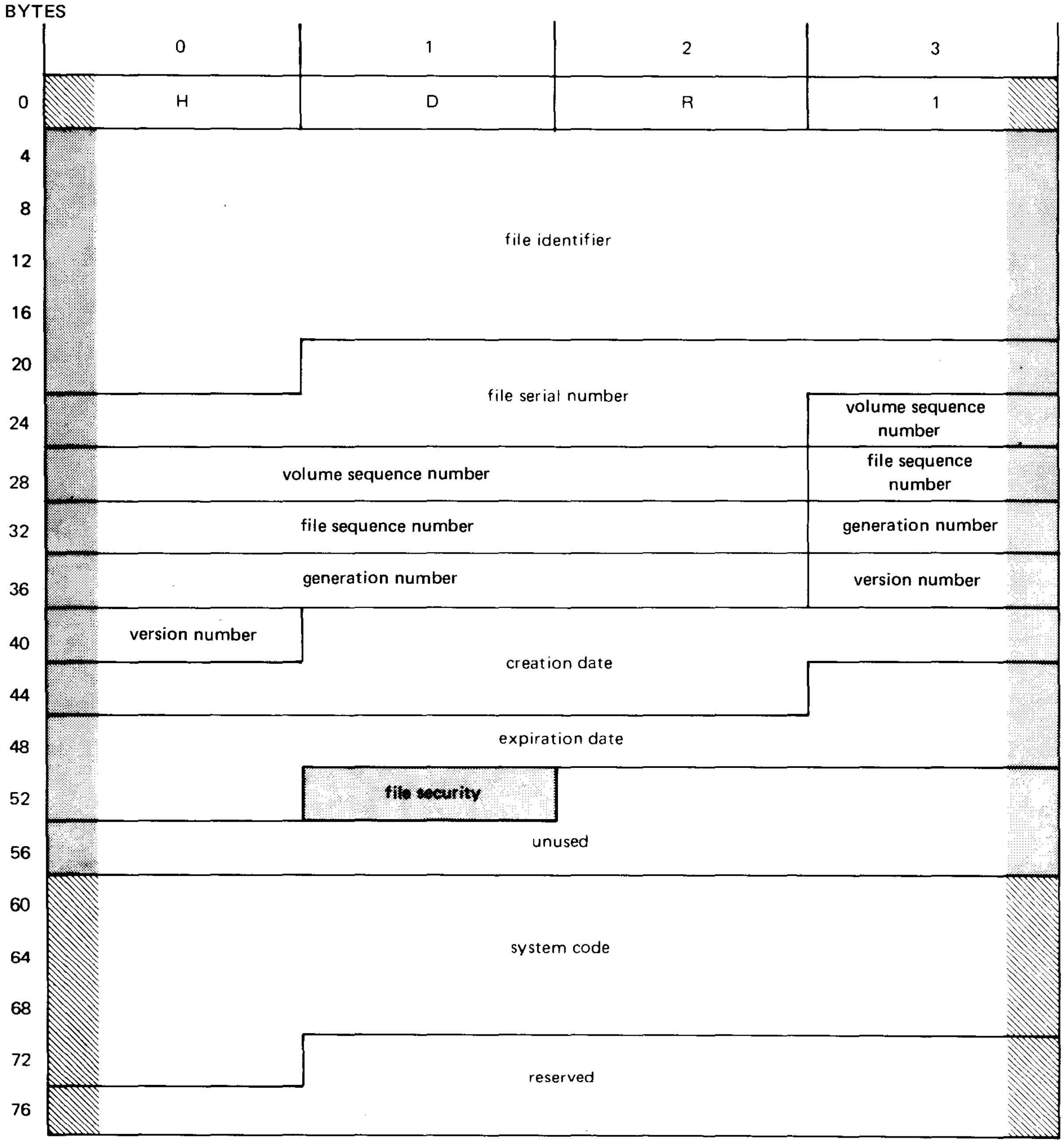
The file header label group consists of two labels: the file header 1 label (HDR1) and the file header 2 label (HDR2).

4.6.2.1. 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 4-7; its fields are described in Table 4-2. 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 macroinstruction, the HDR1 label is bypassed without processing. For multifile 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.

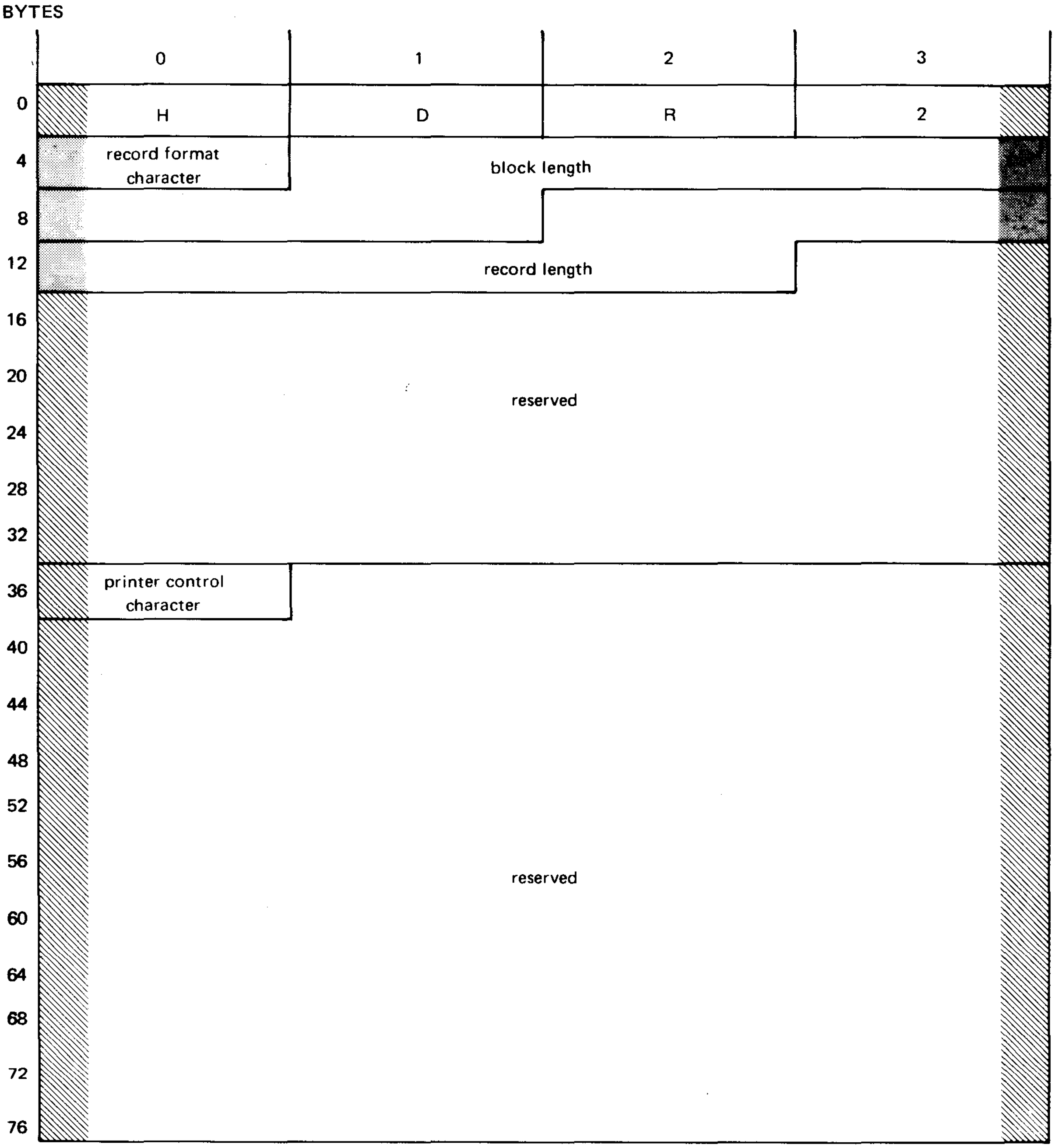
Figure 4—7. First File Header Label (HDR1) Format for an EBCDIC Tape Volume

Table 4—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}).

4.6.2.2. 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 operating system as indicated in the system code field of the HDR1 label, the HDR2 label is ignored by TSAT. Figure 4–8 shows the format of the HDR2 label; Table 4–3 describes its fields.



LEGEND:

▨ Generated by TSAT or reserved for system expansion.

░ Written by TSAT from user-supplied data.

Figure 4—8. Second File Header Label (HDR2) Format for an EBCDIC Tape Volume

Table 4—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	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

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.

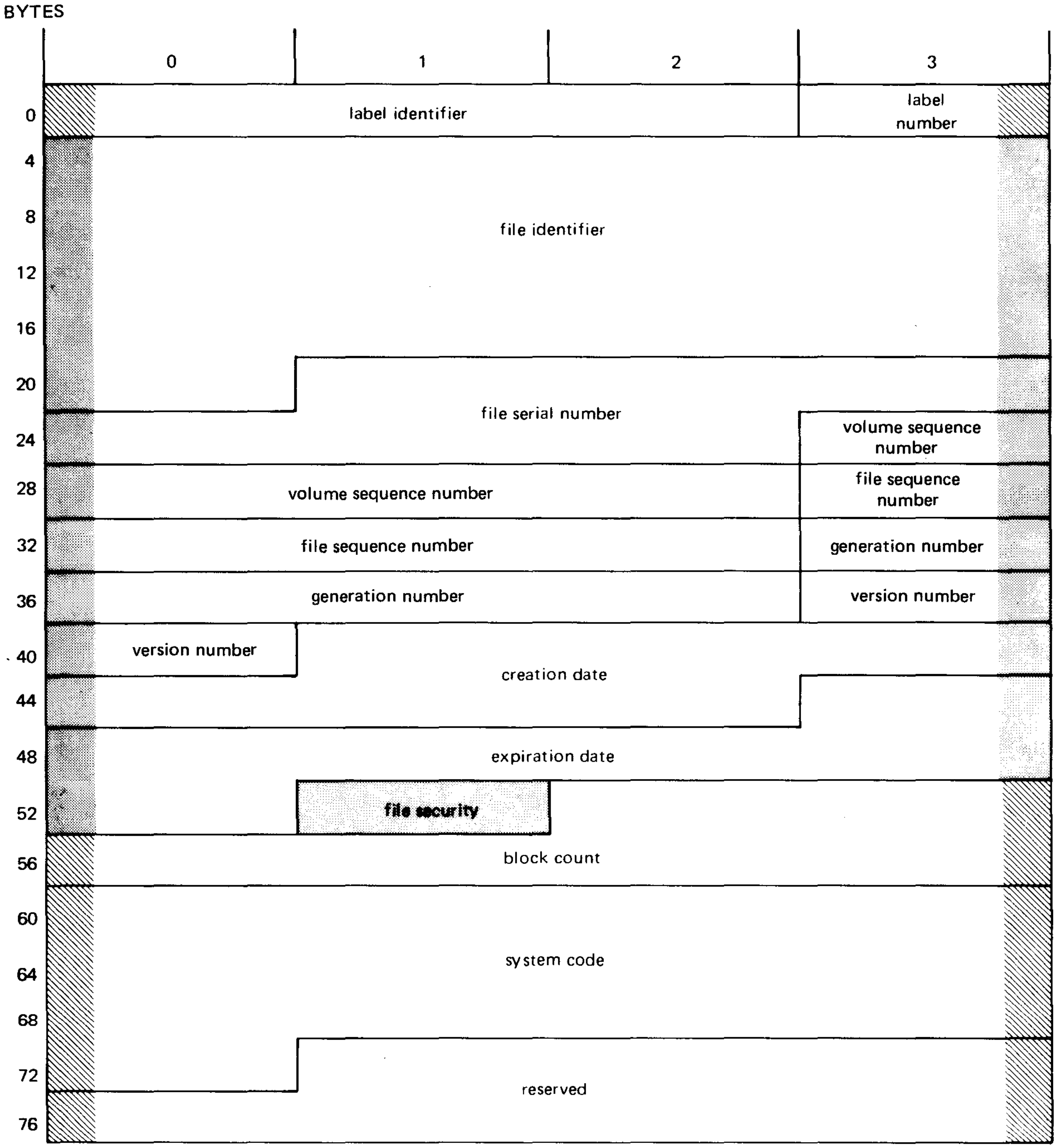
4.6.3. File Trailer Label Group

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 macroinstruction to an input tape file, with READ=BACK specified in the TCA macroinstruction, 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 4-9 illustrates the format of the EOF1 or EOV1 label; Table 4-4 summarizes the contents of its fields. Figure 4-10 illustrates the format of the EOF2 or EOV2 label; Table 4-5 presents the contents of its fields.



LEGEND:

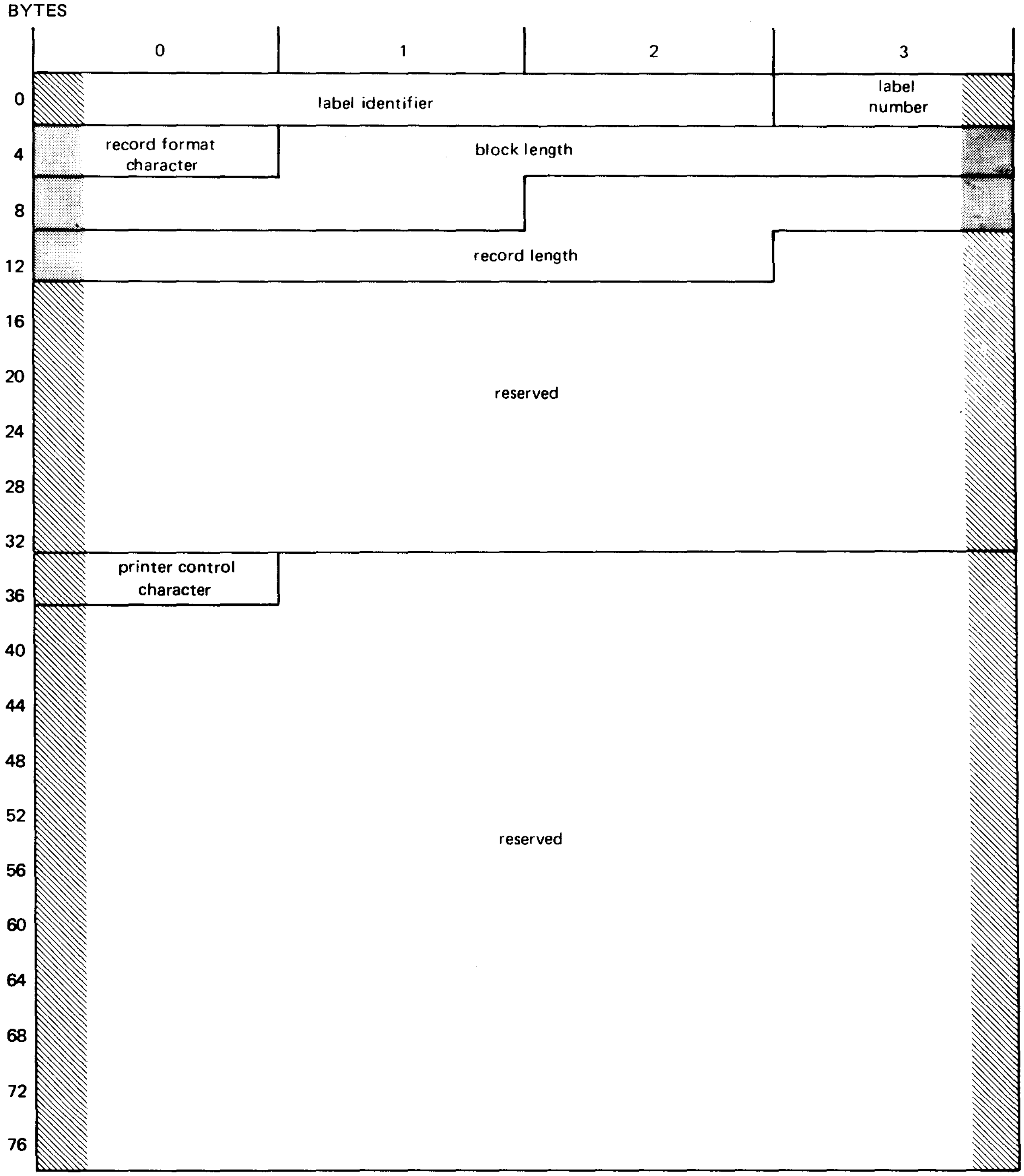
▨ Generated by TSAT or reserved for system expansion.

░ Written by TSAT from user-supplied data.

Figure 4—9. Tape File EOF1 and EOV1 Label Formats for EBCDIC Tapes

Table 4—4. Tape File EOF1 and EOV1 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 EOV 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 4—10. Tape File EOF2 and EOV2 Label Formats for EBCDIC Tapes

Table 4—5. Tape File EOF2 and EOV2 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 EOV for an end-of-volume label
Label number	3	Always 2
Record format character	4	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 zeros.
(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

4.7. TAPE VOLUME AND FILE ORGANIZATION

As was stated earlier, magnetic tape files processed by TSAT must observe certain label conventions. These were described in 4.6. Magnetic tape files must also observe conventions as to volume and file organization. The following subsections 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.

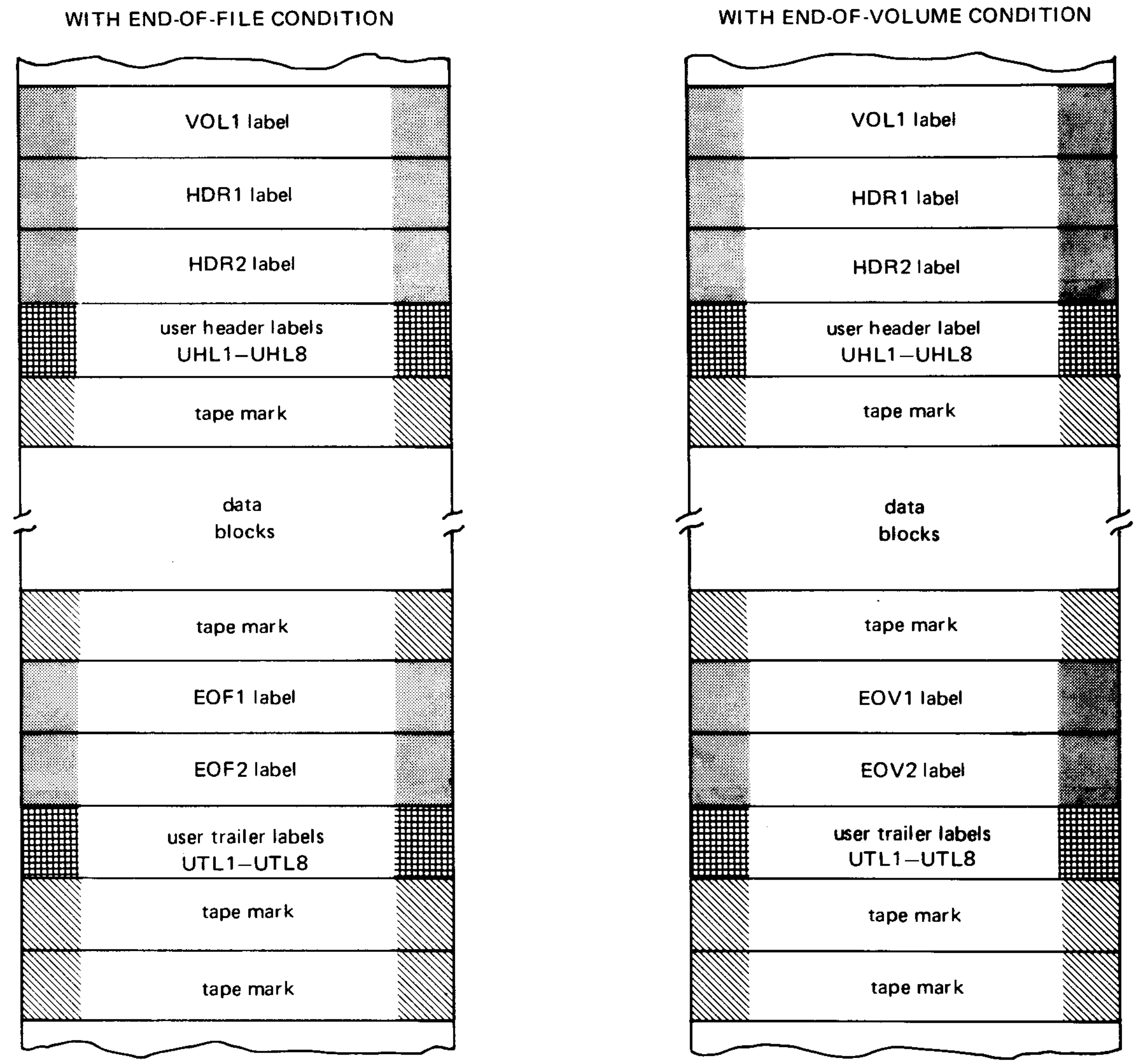
Remember that TSAT assumes only standard labeled files. TSAT bypasses user header labels, user trailer labels, and nonstandard labels. These labels are included in the following figures and descriptions only to show their relative location within the various volume organizations.

4.7.1. Standard Tape Volume Organization

A standard volume has standard labels, required tape marks, and is capable of being processed by the logical IOCS. Figures 4-11, 4-12, and 4-13 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:

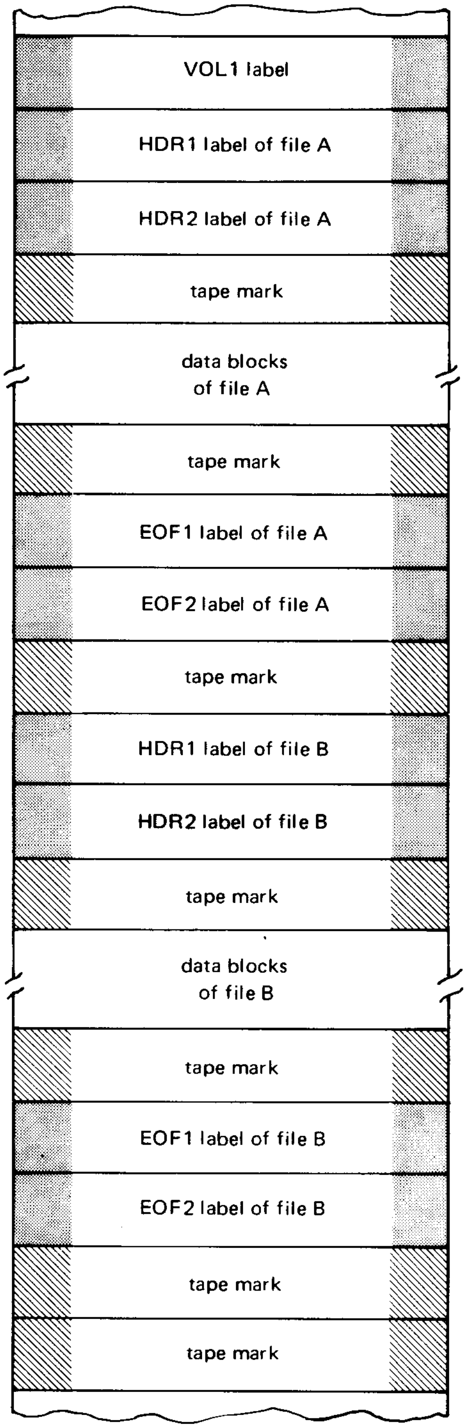
□ Contents supplied by user.

▨ 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 UHLs and eight UTLs. Bypassed by TSAT.

Figure 4—11. Reel Organization for EBCDIC Standard Labeled Tape Volumes Containing a Single File



LEGEND:

□ Content supplied by user.

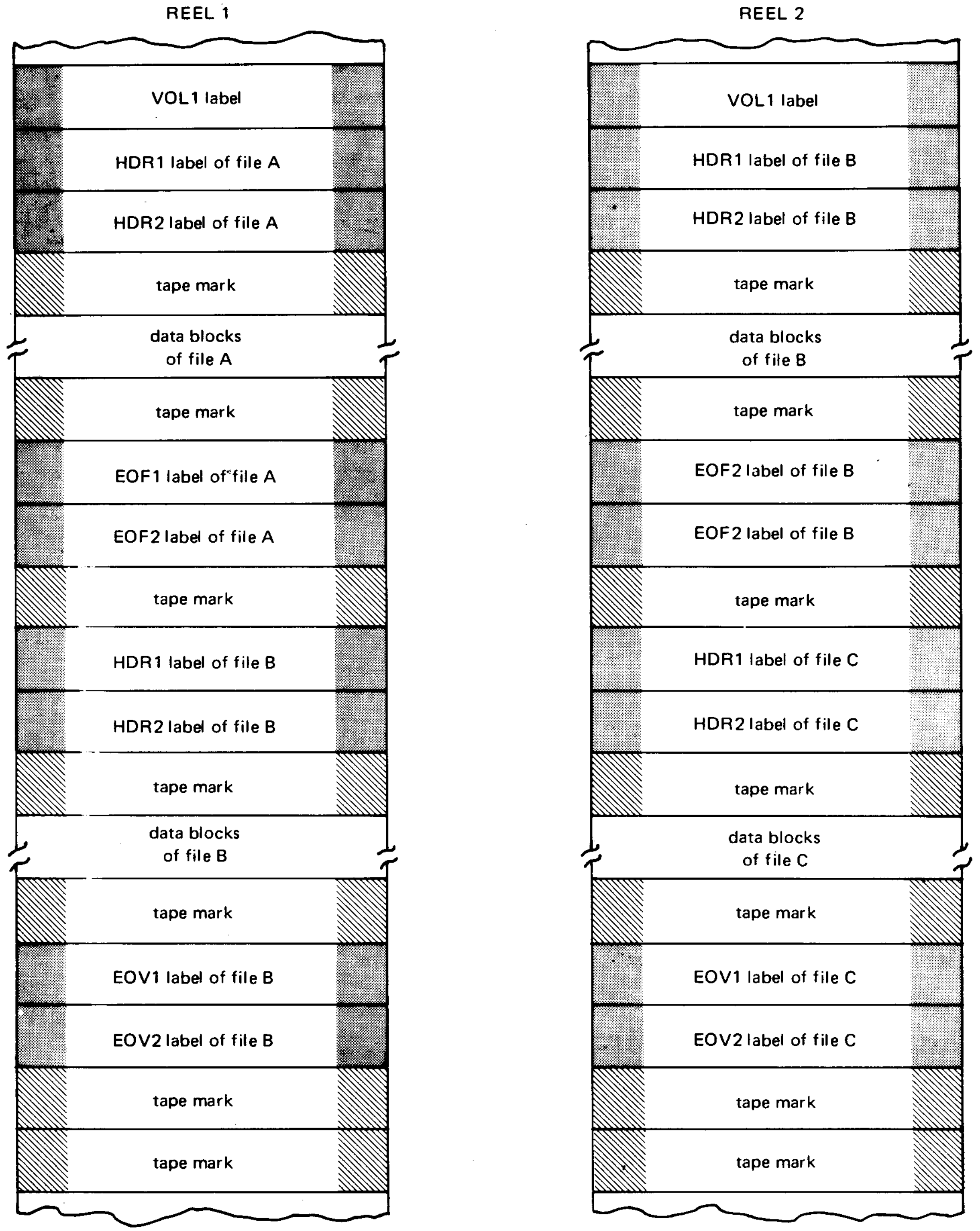
▨ Required and generated by TSAT.

▒ Generated by TSAT; user supplies content for certain fields.

NOTE:

Assume that file B completes on this volume.

Figure 4—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 EOV1 and EOV2 labels shown here would be replaced with EOF1 and EOF2 labels.

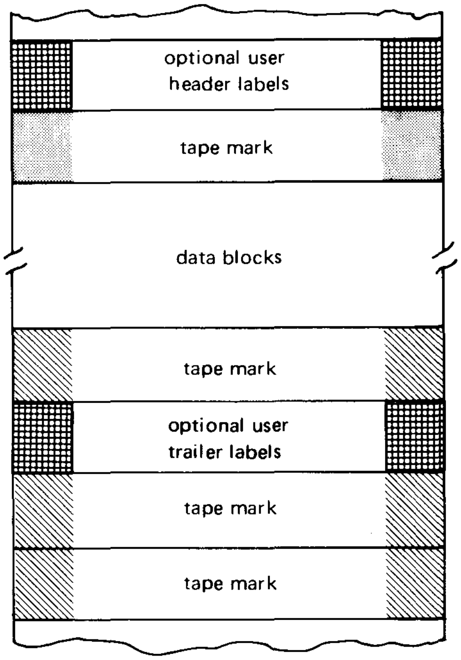
Figure 4—13. Reel Organization for EBCDIC Standard Labeled Tape Volumes: Multifile Volumes with End-of-Volume Condition

4.7.2. 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 4-14 and 4-15 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:

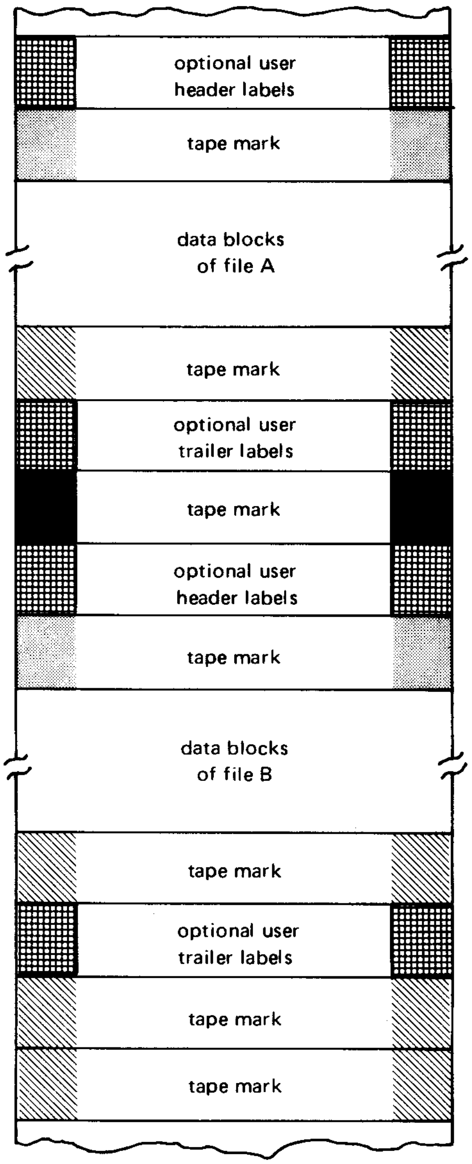
□ Contents supplied by user.

▨ Required and generated by TSAT; only two tape marks follow data blocks if UTLs 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 4—14. Reel Organization for EBCDIC Nonstandard Volume Containing a Single File


optional user header labels
tape mark
data blocks of file A
tape mark
optional user trailer labels
tape mark
optional user header labels
tape mark
data blocks of file B
tape mark
optional user trailer labels
tape mark
tape mark

LEGEND:

□ Content supplied by user.

▧ Required and generated by TSAT; only two tape marks follow data blocks of last file on volume if UTLs 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.

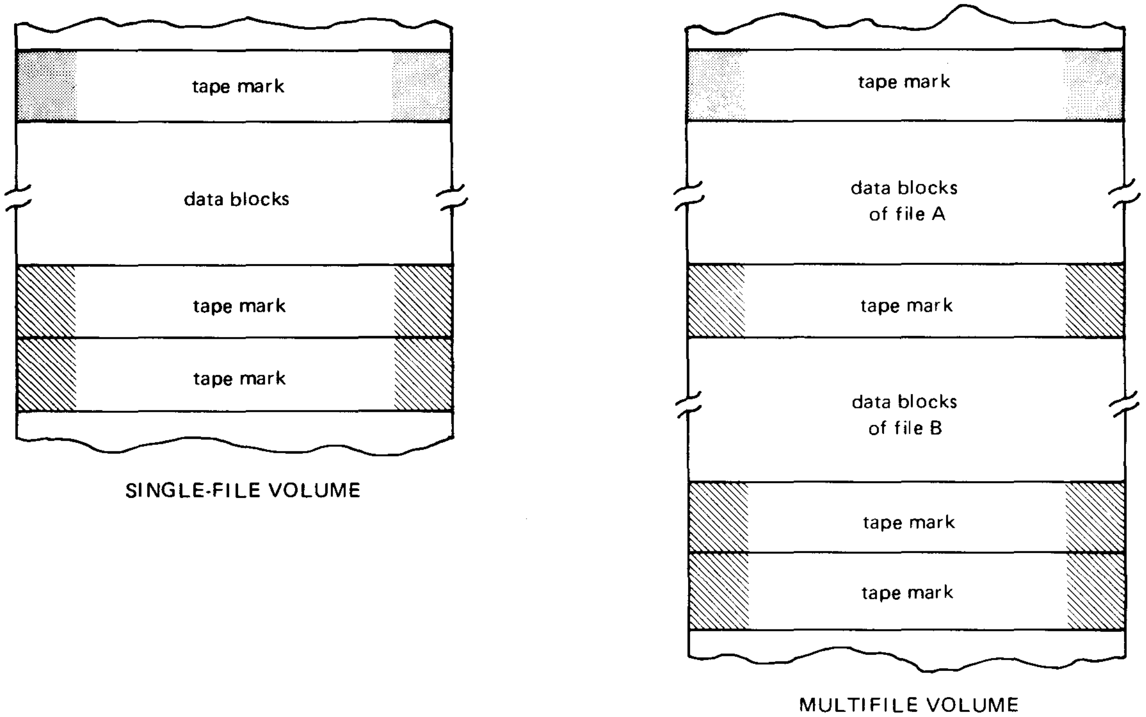
■ Always present; written by TSAT

Figure 4—15. Reel Organization for EBCDIC Nonstandard Multifile Volume

4.7.3. 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 macroinstruction, 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 macroinstruction.

Figure 4-16 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.



LEGEND:

□ Content supplied by user.

▨ Required and generated by TSAT; two tape marks follow data blocks of last file on volume.

░ Generated by TSAT unless user specifies TPMARK=NO; required only when user specifies READ=BACK.

Figure 4—16. Reel Organization for Unlabeled EBCDIC Volumes

4.8. TAPE SAT FILE INTERFACE

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

- SAT

  Defines a TSAT magnetic tape file.

- TCA

  Defines a tape control appendage.

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

4.8.1. Define a Magnetic Tape File (SAT)

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

Function:

The SAT macroinstruction 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 macroinstruction and must not appear in a sequence of executable codes.

Format:

LABEL	△OPERATION△	OPERAND
filename	SAT	TCA=TCA-name [,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 macroinstruction describing the tape control appendage.

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 macroinstruction, you have placed the FCB for this file in the I/O area specified by the IOAREA1 keyword parameter of the TCA macroinstruction 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 macroinstruction 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 macroinstruction after each I/O operation.

4.8.2. Define a Tape Control Appendage (TCA)

Function:

The TCA macroinstruction 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 macroinstruction and must not appear in a sequence of executable code.

Format:

LABEL	△OPERATION△	OPERAND
TCA-name	TCA	IOAREA1=area-name
		,BLKSIZE=n
		[,BKNO=YES]
		[,CKPTREC=YES]
		$\left[,CLRW=\begin{Bmatrix}NORWD\\RWD\end{Bmatrix}\right]$
		[,EOFADDR=end-of-data-addr]

(continued)

LABEL	△OPERATION△	OPERAND
TCA-name (cont)		$\left[,\text{FILABL=}\begin{Bmatrix}\text{STD}\\ \text{NSTD}\\ \text{NO}\end{Bmatrix}\right]$ [,LBLK=n] [,OPRW=NORWD] $\left[,\text{READ=}\begin{Bmatrix}\text{FORWARD}\\ \text{BACK}\end{Bmatrix}\right]$ $\left[,\text{REWIND=}\begin{Bmatrix}\text{UNLOAD}\\ \text{NORWD}\end{Bmatrix}\right]$ [,TPMARK=NO] [,TYPEFLE=OUTPUT]

Label:

TCA-name

Specifies the symbolic address of the TCA table generated by this macroinstruction. This must be the same name specified in the TCA parameter of the SAT macroinstruction 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 4.10.

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 macroinstruction 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:

//△CHKPT△//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 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.

- PUT

  Outputs the next logical block.

- WAITF

  Waits for block transfer.

- CNTRL

  Controls tape unit functions.

- CLOSE

  Closes a tape file.

4.9.1. Open a Tape File (OPEN)

Function:

After the file has been defined by the SAT and TCA declarative macroinstructions, the OPEN macroinstruction must be issued to initialize the file before any other access can be made. This macroinstruction 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 macroinstruction in the program corresponding to the file to be opened.

(1)
: Indicates that register 1 has been preloaded with the address of the SAT macroinstruction.

Positional Parameter n:

filename-n
: Successive entries specify the symbolic addresses of the SAT macroinstructions in the program corresponding to the additional files to be opened.

4.9.2. Get Next Logical Block (GET)

Function:

The GET macroinstruction 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 macroinstruction by the keyword parameter IOAREA1.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	GET	$\left\{\begin{array}{l}\text{filename}\\ \text{(1)}\end{array}\right\},\left\{\begin{array}{l}\text{TCA-name}\\ \text{(Ø)}\end{array}\right\}$

Positional Parameter 1:

filename

Specifies the symbolic address of the SAT macroinstruction in the program corresponding to the file being read.

(1)

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

Positional Parameter 2:

TCA-name

Specifies the symbolic address of the TCA macroinstruction associated with the partition to be accessed.

(Ø)

Indicates that register 0 has been preloaded with the address of the TCA macroinstruction.

When a GET macroinstruction is issued for a SAT file, the contents of registers 14, 13, and 12 are saved in three consecutive full words whose address is in register 13.

4.9.3. Output Next Logical Block (PUT)

Function:

The PUT macroinstruction 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 macroinstruction by the keyword parameter IOAREA1.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	PUT	$\left\{\begin{array}{l}\texttt{filename}\\\texttt{(1)}\end{array}\right\},\left\{\begin{array}{l}\texttt{TCA-name}\\\texttt{(Ø)}\end{array}\right\}$

Positional Parameter 1:

`filename`
Specifies the symbolic address of the SAT macroinstruction in the program corresponding to the file being written.

`(1)`
Indicates that register 1 has been preloaded with the address of the SAT macroinstruction.

Positional Parameter 2:

`TCA-name`
Specifies the symbolic address of the TCA macroinstruction associated with the partition to be written.

`(Ø)`
Indicates that register 0 has been preloaded with the address of the TCA macroinstruction.

When a PUT macroinstruction is issued for a SAT file, the contents of registers 14, 13, and 12 are saved in three consecutive full words whose address is in register 13.

4.9.4. Wait for Block Transfer (WAITF)

Function:

The WAITF macroinstruction ensures that a command initiated by a preceding GET or PUT macroinstruction 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 macroinstruction, the WAITF macroinstruction must be issued after a GET or PUT macroinstruction and before another imperative macroinstruction is issued for that file.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	WAITF	$\left\{\begin{array}{l}\texttt{filename}\\\texttt{(1)}\end{array}\right\}$

Positional Parameter 1:

filename
Specifies the symbolic address of the SAT macroinstruction in the program corresponding to the file being accessed.

(1)
Indicates that register 1 has been preloaded with the address of the SAT macroinstruction.

4.9.5. Control Tape Unit Functions (CNTRL)

Function:

This macroinstruction 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 macroinstruction following a CNTRL macroinstruction.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CNTRL	$\left\{\begin{matrix}\text{filename}\\ \text{(1)}\end{matrix}\right\}$,code

Positional Parameter 1:

filename
Specifies the symbolic address of the corresponding SAT macroinstruction in the program.

(1)
Indicates that register 1 has been preloaded with the address of the SAT macroinstruction.

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*

**Applies only to input files.*

REW	Rewind tape
RUN	Rewind tape with interlock (unloads tape)
WTM	Write tape mark

4.9.6. Close a Tape File (CLOSE)

Function:

The CLOSE macroinstruction performs the required termination operations for a file, for example, construction of the EOF label group. Once the CLOSE macroinstruction has been issued for a file, only the OPEN macroinstruction may reference that file.

Format;

LABEL	△OPERATION△	OPERAND
[symbol]	CLOSE	{filename-1[,....,filename-n] (1)}

Positional Parameter 1:

filename-1
Specifies the symbolic address of the SAT macroinstruction in the program corresponding to the file to be closed.

(1)
Indicates that register 1 has been preloaded with the address of the SAT macroinstruction.

Positional Parameter n:

filename-n
Successive entries specify the symbolic addresses of the SAT macroinstructions in the program corresponding to the additional files to be closed.

4.10. 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 is executed by PIOCS. Some of the general requirements 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 3 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. PIOCS uses both of these counters when reading and writing block numbered tapes.
- 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 4-17 through 4-21.
- 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, PIOCS 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.

4.10.1. 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 macroinstruction).

3. You must indicate to TSAT that you have reserved the 4-byte block number area by specifying BKNO=YES in the TCA macroinstruction (4.8.2).

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.

4.10.2. 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 in the following subsections.

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.

4.10.2.1. Initialized Processing

Initialized processing includes:

- TPREP utility routine processing, described in the system service programs user guide, UP-8841 (current version);

- processing output files with standard labels (FILABL=STD specified in the TCA macroinstruction) 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 macroinstruction).

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

4.10.2.2. Noninitialized Processing

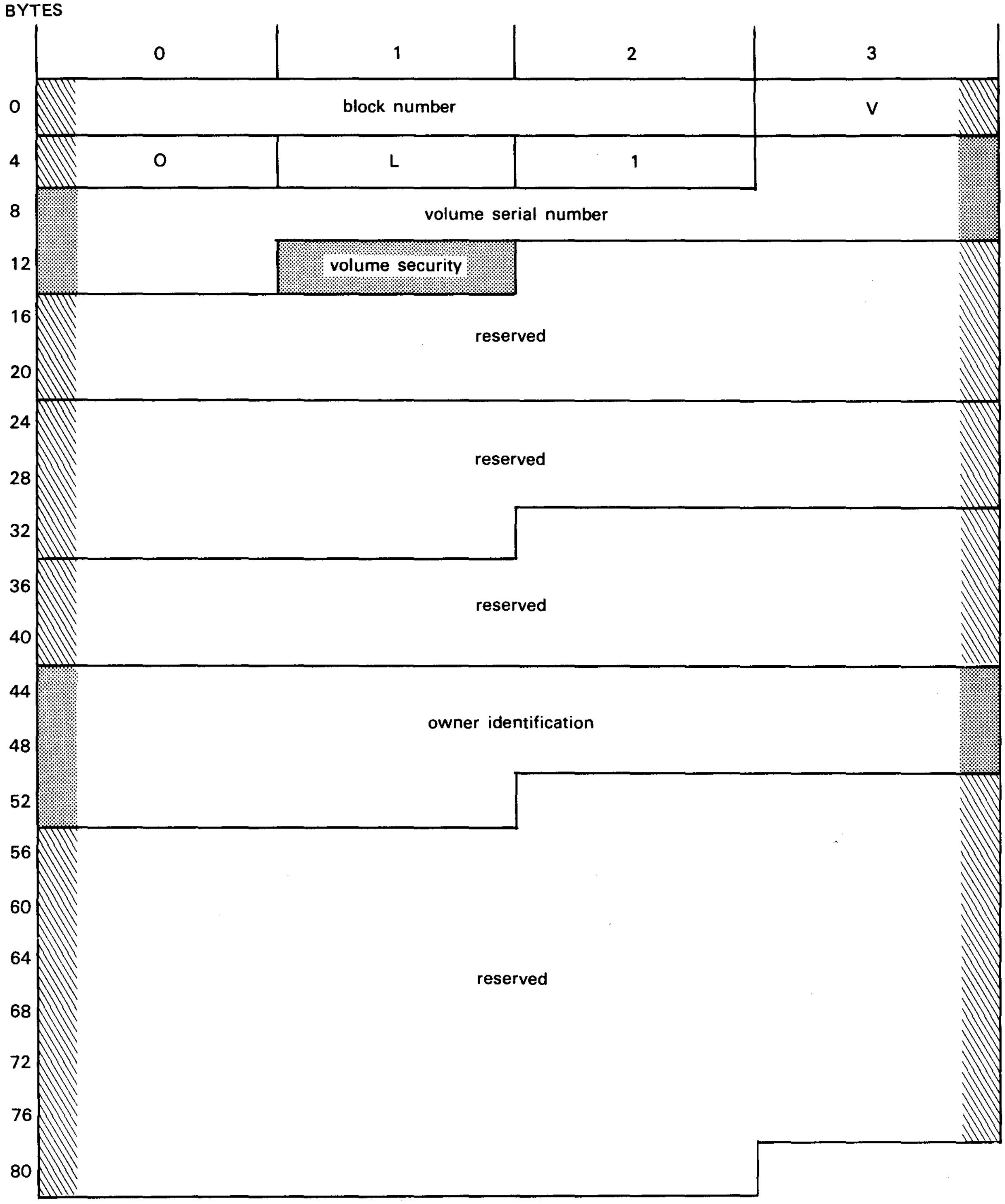
Noninitialized processing includes:

- processing output files with standard labels (FILABL=STD specified in the TCA macroinstruction), but without PREP specified in the VOL job control statement; or
- processing input files with standard labels (FILABL=STD specified in the TCA macroinstruction).

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:

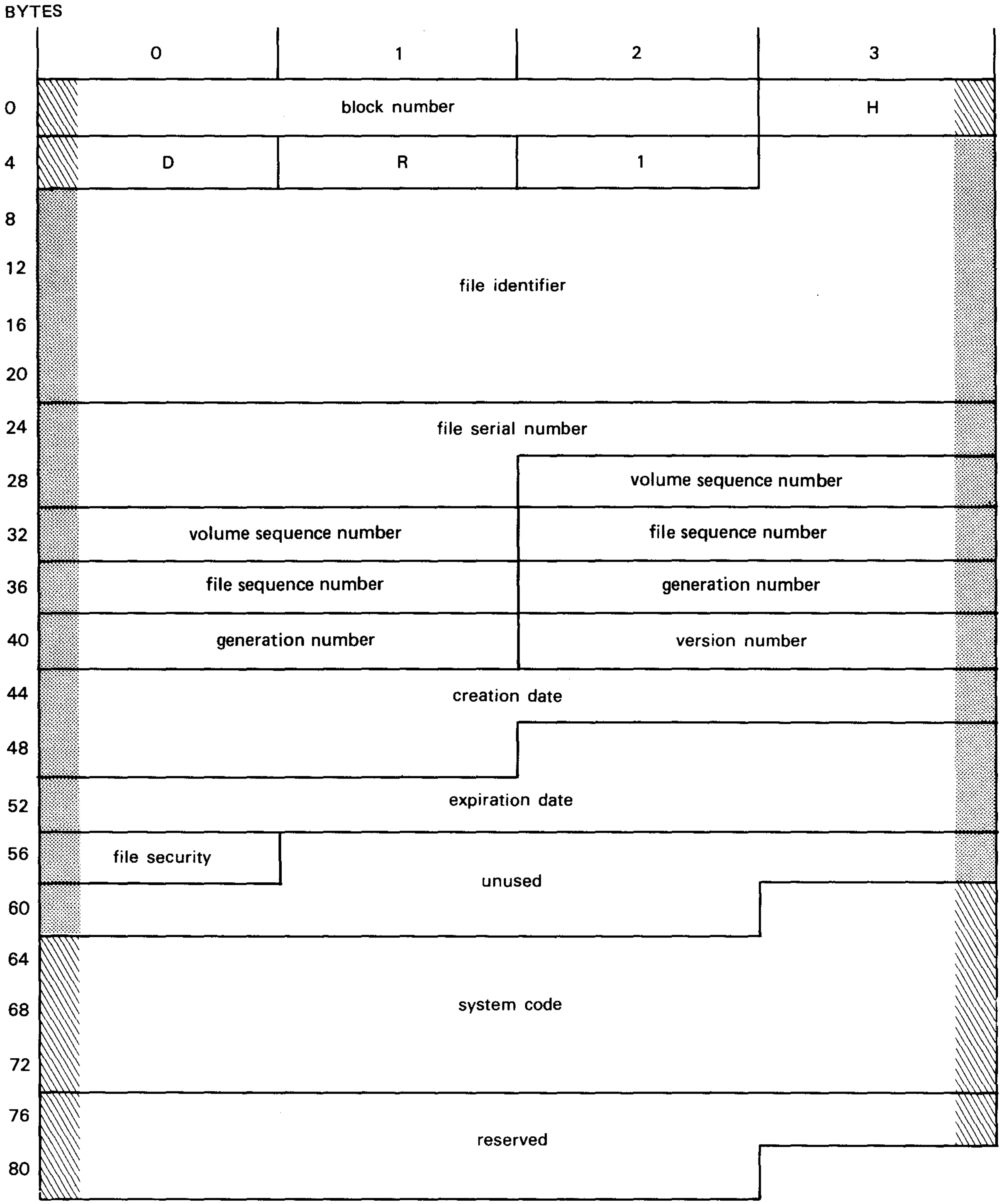
▨ Generated by TSAT or reserved for system expansion.

▒ 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 4–1, except that each field is offset three bytes.

Figure 4—17. Tape Volume 1 (VOL1) Label Format for an EBCDIC Volume with Block Numbers



BYTES	0	1	2	3
0	block number	block number	block number	H
4	D	R	1	file identifier
8	file identifier	file identifier	file identifier	file identifier
12	file identifier	file identifier	file identifier	file identifier
16	file identifier	file identifier	file identifier	file identifier
20	file identifier	file identifier	file identifier	file identifier
24	file serial number	file serial number	file serial number	file serial number
28	file serial number	file serial number	volume sequence number	volume sequence number
32	volume sequence number	volume sequence number	file sequence number	file sequence number
36	file sequence number	file sequence number	generation number	generation number
40	generation number	generation number	version number	version number
44	creation date	creation date	creation date	creation date
48	creation date	creation date	expiration date	expiration date
52	expiration date	expiration date	expiration date	expiration date
56	file security	unused	unused	unused
60	unused	unused	unused	system code
64	system code	system code	system code	system code
68	system code	system code	system code	system code
72	system code	system code	system code	system code
76	reserved	reserved	reserved	reserved
80	reserved	reserved	reserved	

LEGEND:

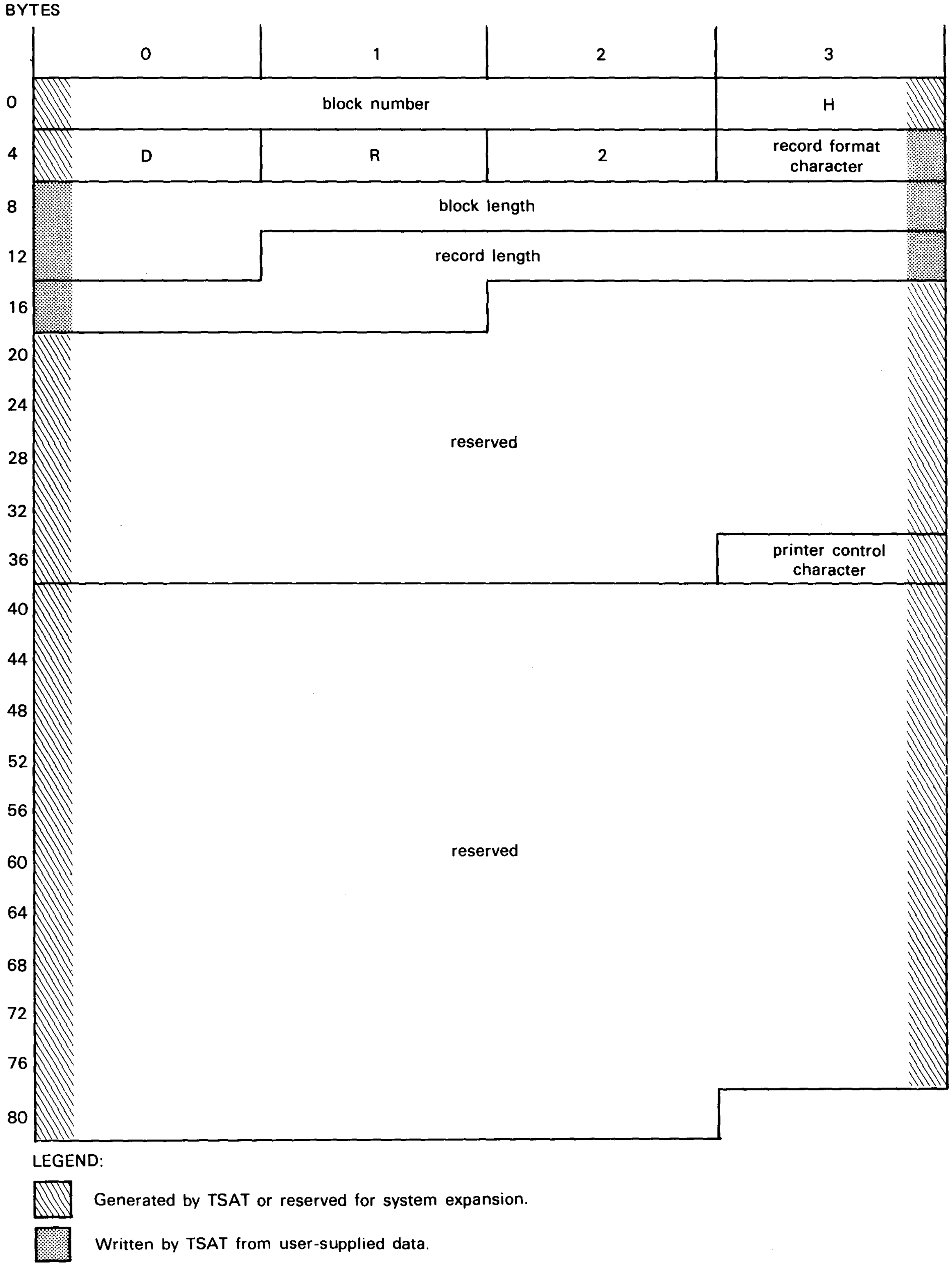
▨ Generated by TSAT or reserved for system expansion.

░ 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 4-2, except that each field is offset three bytes.

Figure 4—18. First File Header Label (HDR1) Format for an EBCDIC Tape Volume with Block Numbers



LEGEND:

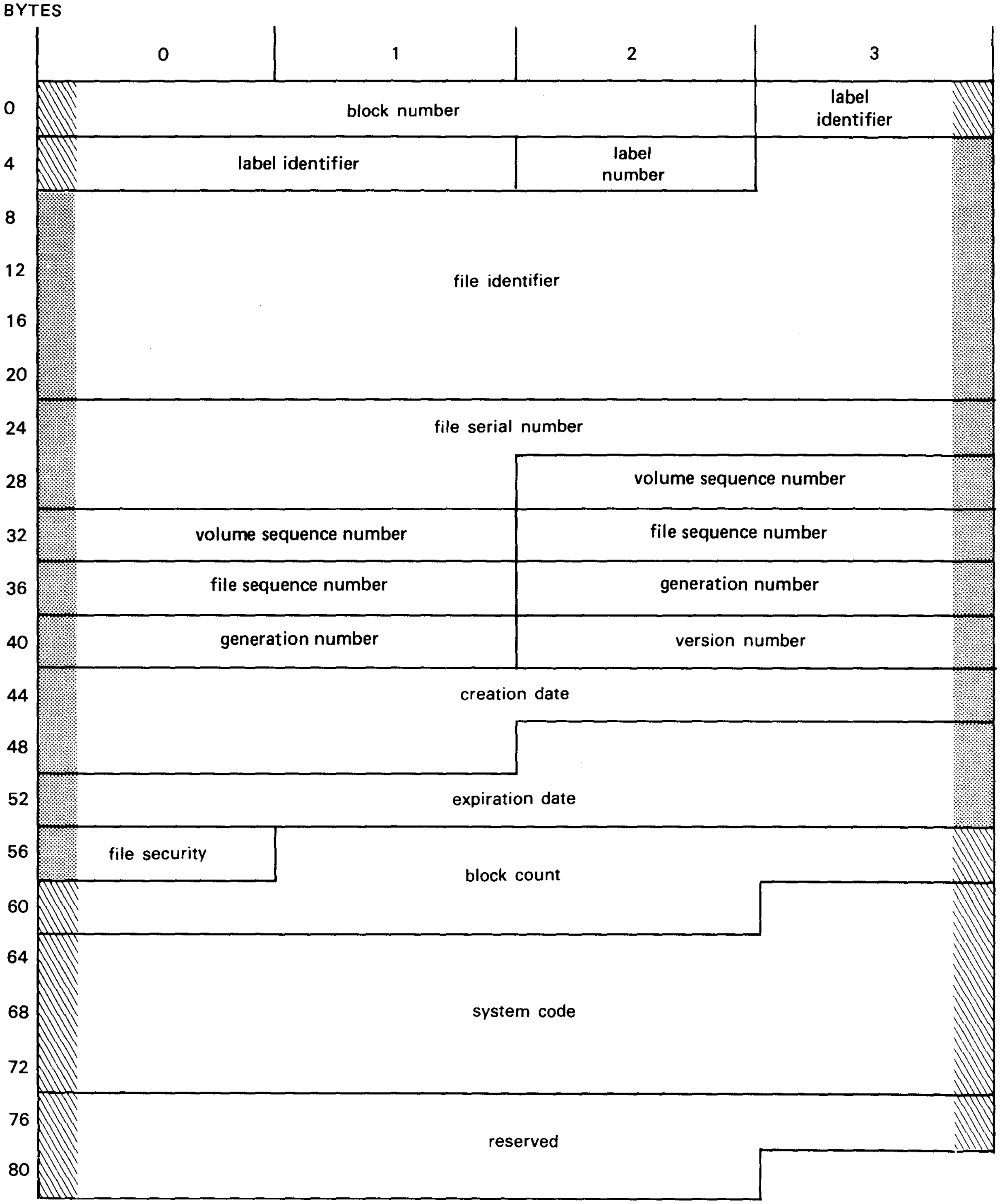
▨ Generated by TSAT or reserved for system expansion.

░ 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 4-3, except that each field is offset three bytes.

Figure 4—19. Second File Header Label (HDR2) Format for an EBCDIC Tape Volume with Block Numbers



LEGEND:

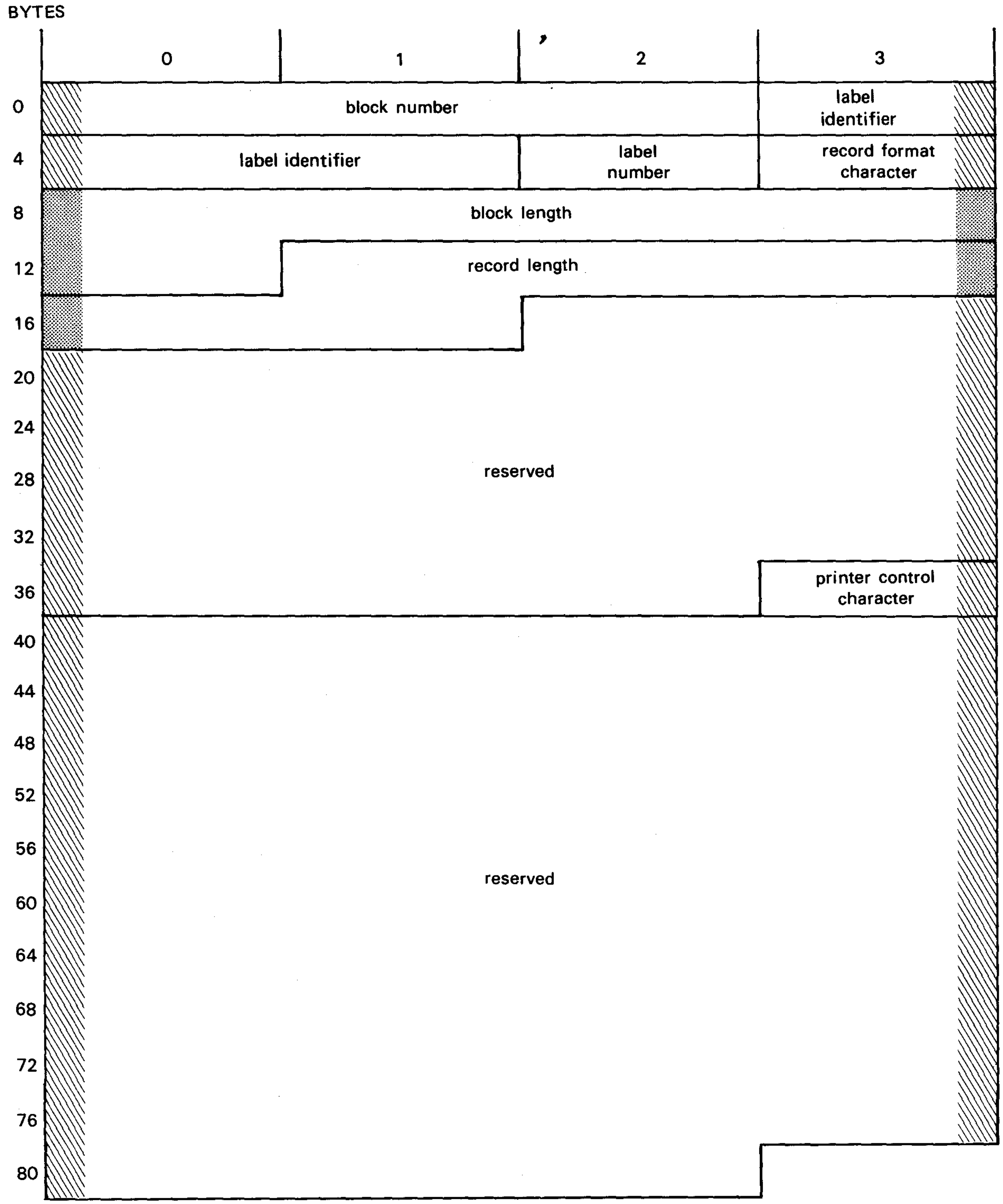
▨ Generated by TSAT or reserved for system expansion.

░ 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 EOV1 labels are the same as described in Table 4–4, except that each field is offset three bytes.

Figure 4—20. Tape File EOF1 and EOV1 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 EOV2 labels are the same as described in Table 4–5, except that each field is offset three bytes.

Figure 4—21. Tape File EOF2 and EOV2 Label Formats for Block Numbered EBCDIC Files

5. Multitasking Macros

5.1. GENERAL

5.1.1. Multijobbing and Multitasking

The OS/3 data processing system can concurrently process from 1 to 14 jobs, each job consisting of one or more job steps (programs) that 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 14) for concurrent execution. The allocation of processor time to these jobs is based on a system switch list that 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.

5.1.1.1. 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.

5.1.1.2. 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.

5.2. TASK MANAGEMENT

5.2.1. General

The supervisor is designed with multitasking capability which is utilized by the supervisor and extended to the user through macroinstructions. 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 (subtasks, other than primary) are created by the ATTACH macroinstruction 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 macroinstruction is executed for that task or an error occurs that prevents the task from successfully completing its work. A DETACH in behalf of the primary task is interpreted as an end-of-job step. When a subtask terminates, the task's event control block (ECB) is reset to the idle/unused state and any task waiting for that task is activated.

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.

5.2.2. Task Creation

Task creation is performed by the ATTACH macroinstruction, which causes entry into the attach function to create a subtask. The code to be executed by the task specified on the ATTACH macroinstruction call must be in main storage, within the user region, when the ATTACH macroinstruction 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.

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 because a TCB cannot be addressed by the user programs.

5.2.3. 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.

5.2.4. Task Termination

A task executes a DETACH macroinstruction 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 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 on behalf of this task.

5.2.5. 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 macroinstruction to mark itself nondispatchable until further queue entries have been made. Each time an AWAKE macroinstruction 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.

5.2.6. 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 internal subtask naming convention consists of a concatenation of the physical TCB (within the job) and the attaching task's name. The numbering of the subtasks should be random rather than sequential. For example, if a task named X'0102' attaches two subtasks, these might be named X'010205' and X'010209', or X'010203' and X'010207'.

5.3. TASK MANAGEMENT MACROINSTRUCTIONS

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 that 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 macroinstructions 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 the relative priority of a task.

5.3.1. Generate an Event Control Block (ECB)

Function:

The ECB macroinstruction 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 5-1).

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

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	ECB	

There are no parameters for the ECB macroinstruction.

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.

3. A primary task does not have an ECB associated with it; therefore, the primary task cannot be waited. 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        10    16
   | PRIMTASK START
   |          .
   |          .
   |          .
1. |          ATTACH ECB1,START,,2
   |          .
   |          .
   |          .
2. |          WAIT  ECB1
   |          .
   |          .
   |          .
   |          EOJ
3. | ECB1     ECB
   |          .
   |          .
   |          .
4. | START    EQU  *                          SUBTASK EXECUTION STARTS HERE
   |          .
   |          .
   |          .
5. |          DETACH ECB1                     SUBTASK EXECUTION ENDS HERE
```

Explanation:

1. 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.

2. At line 2, the primary task gives up control until the subtask is completed.

3. 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 waited.
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 that is awaiting completion of the task with which this ECB is associated.

Figure 5—1. Event Control Block (ECB) Format

5.3.2. Create an Additional Task (ATTACH)

Function:

The ATTACH macroinstruction 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	△OPERATION△	OPERAND
[symbol]	ATTACH	$\begin{Bmatrix}\text{ECB-name}\\(1)\end{Bmatrix},\begin{Bmatrix}\text{entry-point-name}\\(Ø)\end{Bmatrix}$ $\left[,\begin{Bmatrix}\text{error-addr}\\(r)\end{Bmatrix}\right]$ [,n]

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 macroinstruction is issued.

`(Ø)`
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 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        10    16
   ---------------------------------------------------------------------
1.          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 macroinstruction, 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.

5.3.3. Terminate a Task (DETACH)

Function:

The DETACH macroinstruction terminates a task by delinking the TCB from the switch list and returning the TCB to the job's free TCB queue. If this macroinstruction 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 macroinstruction also clears all locks for the task.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	DETACH	$\left[\begin{Bmatrix}\text{ECB-name}\\(1)\end{Bmatrix}\right]\left[,\begin{Bmatrix}\text{error-addr}\\(r)\end{Bmatrix}\right]$

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       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 macroinstruction, transfer control to error processing routine labeled ERROR.

5.3.4. Yield Until Task Completion (TYIELD)

Function:

The TYIELD macroinstruction 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 macrinstruction is used in combination with the AWAKE macroinstruction which reactivates a task made dormant by the TYIELD macroinstruction.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	TYIELD	

There are no parameters for the TYIELD macroinstruction.

5.3.5. Reactivate a Task (AWAKE)

Function:

The AWAKE macroinstruction reactivates a task made dormant by a TYIELD macroinstruction. 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	△OPERATION△	OPERAND
[symbol]	AWAKE	$\left[\begin{Bmatrix}\text{ECB-name}\\ (1)\end{Bmatrix}\right]$

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 macroinstruction is executed with a zero address in register 1, the primary task will be taken out of a TYIELD condition.

Examples:

```
    1       10     16
1.          AWAKE                  AWAKE PRIMARY TASK
2.          SR     R1,R1
            AWAKE  (1)             AWAKE PRIMARY TASK
3.          AWAKE  ECB1            AWAKE SUBTASK
             .
             .
    ECB1    ECB
4.          LA     R1,ECB1
            AWAKE  (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.

5.3.6. Change a Priority (CHAP)

Function:

The CHAP macroinstruction changes the dispatching priority of the task issuing the instruction. The number (either positive or negative) entered as the operand is added to or subtracted from the current dispatching priority of the task (specified by the switch-priority parameter in the EXEC job control statement). This changes the dispatching priority value, resulting in a lesser or greater priority for the task. A positive value will lower the priority; a negative value will raise the priority. This macroinstruction does not change the priority to a specific level; instead, it adjusts the priority relative to the level under which it is executed.

The highest priority level to which you can change is to the original priority of the job step.

The lowest priority level to which you can change depends upon the number of priority levels specified by the SUPGEN keyword parameter PRIORITY. See the supervisor concepts and facilities manual, UP-8831 (current version).

If you try to raise or lower the priority beyond the specified boundaries, the system will automatically stop at the highest (or lowest) priority level with no error.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	CHAP	$\left\{\begin{matrix}\texttt{n}\\\texttt{(1)}\end{matrix}\right\}$

Positional Parameter 1:

n

Specifies a value to be added to or subtracted from the dispatching priority for the task in order to change its priority. To lower the priority, use a positive number; to raise the priority, use a negative number.

(1)

Indicates register 1 has been preloaded with either a positive or negative increment.

Examples:

To lower priority:

```
     1       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.

To raise priority:

```
     1         10      16
1.            CHAP    —3

2.            L       R1,=A(—3)
              CHAP    (1)
```

Change the dispatching priority of the task by three. This will lower the dispatching priority value in the negative direction by three, which will result in a priority three more than the current priority. Both examples perform the same function.

5.4. TASK SYNCHRONIZATION

5.4.1. 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 macroinstruction.

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 macroinstructions 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 macroinstruction. The ECB is posted with a completion code when a task terminates or executes the TYIELD macroinstruction. The ECB is specified on a WAIT and WAITM instruction in order to hold processing of an issuing task until the awaited task either terminates or issues a POST macroinstruction.

Several macroinstructions 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.

- TPAUSE

  Deactivate a task.

- TGO

  Reactive a task.

A WAIT or WAITM macroinstruction suspends execution of the issuing task. A POST macroinstruction reactivates the suspended task.

A TPAUSE macroinstruction deactivates a task other than the issuing task. A TGO macroinstruction reactivates a task (other than the issuing task) deactivated by a TPAUSE macroinstruction.

The WAIT and WAITM macroinstructions can also be used (with different parameters) to synchronize a task with its I/O. For task synchronization, the macroinstruction references an event control block; for I/O synchronization, the macroinstruction references a command control block.

5.4.2. Wait for Task Completion (WAIT)

Function:

The WAIT macroinstruction temporarily suspends program execution until the specified task is completed or executes a POST macroinstruction in behalf of the waiting task. If the related task is completed, control is returned to the point immediately following the WAIT macroinstruction. 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 macroinstruction is issued, the issuing task relinquishes control until the ECB is marked complete or until a POST macroinstruction is executed by the awaited task in behalf of the waiting task.

Format:

LABEL	△OPERATION△	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.

Examples:

```
1       10    16
-----------------------------------------------
        WAIT  ECB1
        WAIT  (1)
```

5.4.3. Multiple Task Wait (WAITM)

Function:

The WAITM macroinstruction temporarily suspends program execution until any one of several tasks specified by the instruction is completed or executes a POST macroinstruction on 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 macroinstruction, with register 1 containing the address of the event control block associated with the completed task.

Format:

LABEL	△OPERATION△	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 macroinstruction may also specify a combination of ECB and CCB addresses as parameters.

When this macroinstruction 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 macroinstruction always requires more than one event to be tested. If only one event is to be tested, use the WAIT macroinstruction.

5.4.4. Activate the Waiting Task (POST)

Function:

The POST macroinstruction activates the waiting task without requiring the awaited task to terminate. When the POST macroinstruction 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 macroinstruction.

Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	POST	

The task being activated by the POST macroinstruction is the one waiting for the task executing the POST macroinstruction; therefore, there are no parameters for this macroinstruction.

Examples:

```
1        10    16
TASK     EQU   *
         AWAKE TASK2ECB
         WAIT  TASK2ECB
         .
         .
         .
TASK1ECB ECB
TASK2ECB ECB
TASK2    EQU   *
         EXCP  INPUTCCB
         WAIT  INPUTCCB
         POST
         .
         .
         .
         TYIELD
```

5.4.5. Deactivate a Task (TPAUSE)

Function:

The TPAUSE macroinstruction causes a specified task to be deactivated until a subsequent TGO macroinstruction is issued to reactivate that task. This permits better control of task synchronization within a job. The TPAUSE macroinstruction can also be used to deactivate all tasks of a job step other than the issuing task. When used, no other task, even if it had a higher priority or pending island code activation, can possibly interrupt the task and take control.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	TPAUSE	$\begin{Bmatrix}\text{ECB-name}\\ \text{ALL}\\ \text{(1)}\end{Bmatrix}\left[,\begin{Bmatrix}\text{error-addr}\\ \text{(r)}\end{Bmatrix}\right]$

Positional Parameter 1:

ECB-name

Specifies the symbolic address of the event control block of the task to be activated.

ALL

Specifies that all tasks of the job step are to be deactivated. Note that the calling task is not acted upon in this case.

(1)

Specifies that register 1 has been preloaded with the address of one or more addresses pointing to the ECBs controlling the task to be deactivated. The last address in the list must have the X'80' bit set in the high order byte to indicate the termination of the list.

Positional Parameter 2:

error-addr

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

(r)

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

An error is returned to the user if the address or addresses passed do not point to a valid ECB, the ECB does not point to an attached TCB, or the ECB points to the calling task.

5.4.6. Reactivate a Task (TGO)

Function:

The TGO macroinstruction reactivates a specified task or tasks deactivated by a previous TPAUSE macroinstruction. The TGO macroinstruction can also be used to reactivate all tasks of a job step (other than the issuing task) previously deactivated by TPAUSE. If the TYIELD parameter is specified, the issuing task also relinquishes control of the processor.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	TGO	⎧ECB-name⎫ ⎡,⎧error-addr⎫⎤ ⎡,TYIELD⎤ ⎨ALL⎬ ⎣ ⎩(r)⎭⎦ ⎩(1)⎭

Positional Parameter 1:

ECB-name
: Specifies the symbolic address of the event control block of the task to be activated.

ALL
: Specifies that all tasks of the job step are to be reactivated. Note that the calling task is not acted upon in this case.

(1)
: Specifies that register 1 has been preloaded with the address of one or more addresses pointing to the ECBs controlling the task to be activated. The last address in the list must have the X'80' bit set in the high order byte to indicate the termination of the list.

Positional Parameter 2:

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

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

An error is returned to the user if the address or addresses passed do not point to a valid ECB, the ECB does not point to an attached TCB, or the ECB points to the calling task.

Positional Parameter 3:

TYIELD

Specifies that the TYIELD function is to be performed on the issuing task at the end of the TGO processing. If you use this parameter, the effect is exactly as if you had coded a TGO followed immediately by a TYIELD. If you had done that, however, your task could have been interrupted between the TGO and TYIELD macroinstructions and possibly lost control to another task. This parameter eliminates that possibility and the TYIELD takes effect immediately.

6. Spooling - Breakpoint Macroinstruction

6.1. 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 consolidated data management concepts and facilities, UP-8825 (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 in the following subsections. Figure 6-1 gives a simplified picture of the relationship between the input/output devices and the software components of the spooling function.

6.1.1. 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.

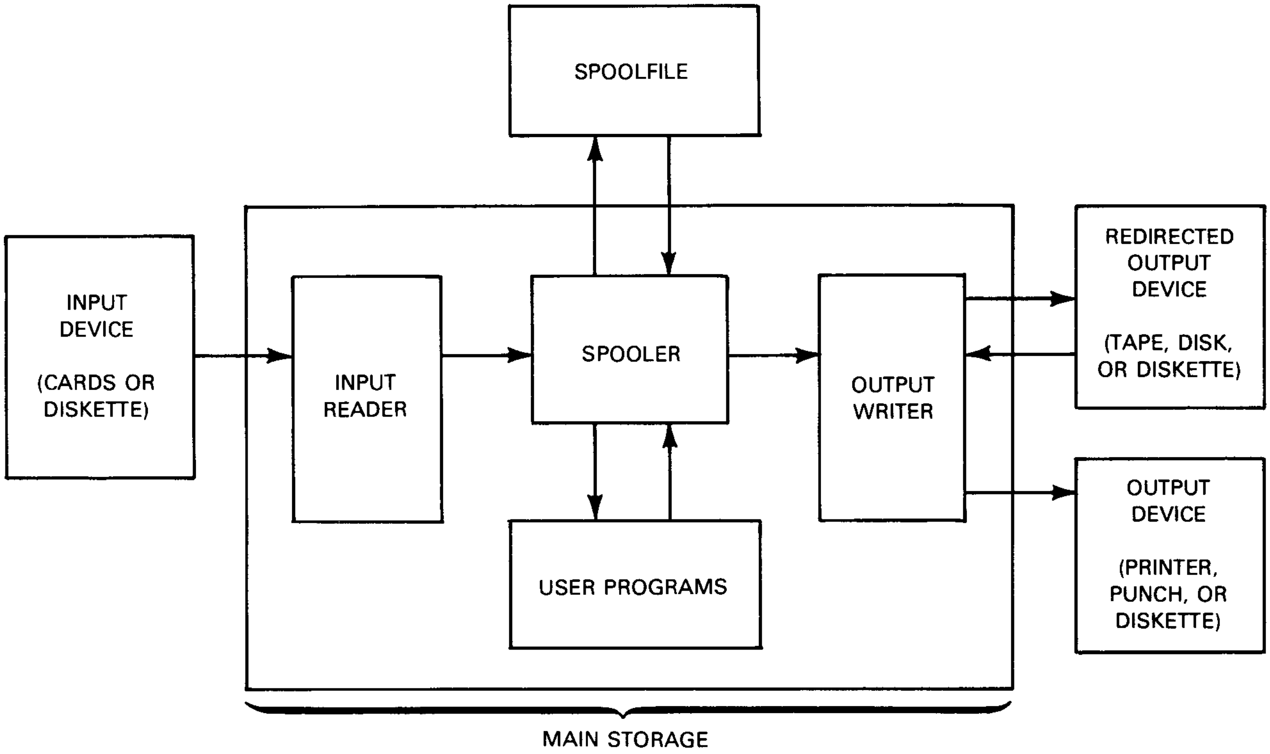

SPOOLFILE
INPUT DEVICE (CARDS OR DISKETTE)
INPUT READER
SPOOLER
OUTPUT WRITER
REDIRECTED OUTPUT DEVICE (TAPE, DISK, OR DISKETTE)
OUTPUT DEVICE (PRINTER, PUNCH, OR DISKETTE)
USER PROGRAMS
MAIN STORAGE

Figure 6—1. Relationship of Spooling Devices and User Programs

6.1.2. Input Reader

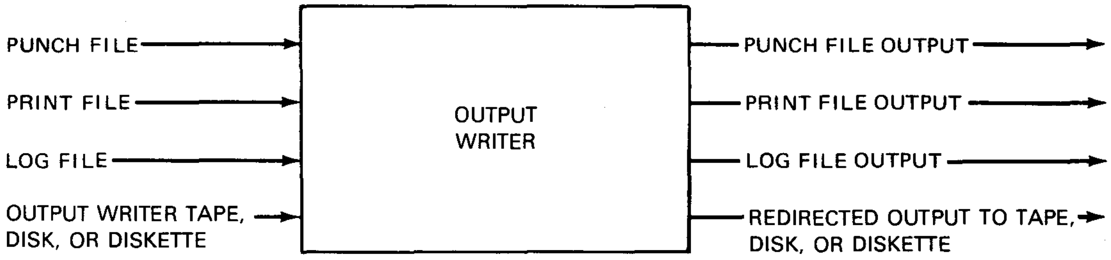
The input reader reads cards from a real card reader or records from a diskette and writes these images to the spoolfile via a virtual card reader and the spooler. It closes the previous subfile if one exists and opens a new subfile. A given input reader can handle only one card reader or diskette at a time; however, any number of input readers can be active.

6.1.3. 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 disk when necessary using the system access technique (SAT) for accesses to the spoolfile. All input/output requests addressing virtual devices are trapped and routed to the spooler for processing. The spooler supports both reads and writes to virtual devices while simulating the action of PIOCS 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.

6.1.4. 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, disk, or diskette so that it 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 define burst mode by selecting a subcriterion.
- 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, disk, or diskette.

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, 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.

6.1.5. Breakpoint

There are a number of special functions, such as open, close, find, and 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 DMBRK macroinstruction.

6.2. 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 data management macroinstructions. A job that runs on a nonspooling system will also run on a spooling system, and vice versa. If you use the DMBRK macroinstruction in your program, it will be ignored if your job is run on a nonspooling system.

6.3. CREATE A BREAKPOINT IN A SPOOL OUTPUT FILE (DMBRK)

Function:

The DMBRK macroinstruction 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 macroinstruction is included in a program executing in a system that does not have the spooling capability, the macroinstruction is ignored.

Format:

LABEL	△OPERATION△	OPERAND
[symbol]	DMBRK	{filename (1)}

Positional Parameter 1:

filename

Specifies the symbolic address of the CDIB macroinstruction in the program which defines the file in which a breakpoint is to be created.

(1)

Specifies that register 1 contains the address of the CDIB macroinstruction defining the file in which a breakpoint is to be created.

7. Diagnostic and Debugging Aids

7.1. 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 macroinstructions: CANCEL, 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.

7.1.1. Snapshot Dumps (SNAP/SNAPF)

A snapshot dump is, by definition, a *selective dynamic dump* performed at various times in a run. The SNAP macroinstruction 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 macroinstructions for obtaining snapshot dumps: SNAP and SNAPF. Each macroinstruction performs the same function, except that the SNAPF macroinstruction is used in the spooling environment. To simplify this discussion, whenever we mention the SNAP macro, we also mean the SNAPF macroinstruction.

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 macroinstruction 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 macroinstruction is bypassed.

A hexadecimal printout of the general registers and the job's main storage area is always given when the SNAP macroinstruction executes. Program-relative addresses are listed on the left, and absolute addresses are listed on the right. After the SNAP macroinstruction executes, normal processing of the program continues. Control is returned to the instruction immediately following the SNAP macroinstruction.

The SNAPF macroinstruction works the same way as the SNAP macroinstruction, 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 macroinstructions are:

LABEL	△OPERATION△	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 macroinstruction may be used.

If you use the SNAPF macroinstruction, 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 a data management OPEN macroinstruction. or RDFCB

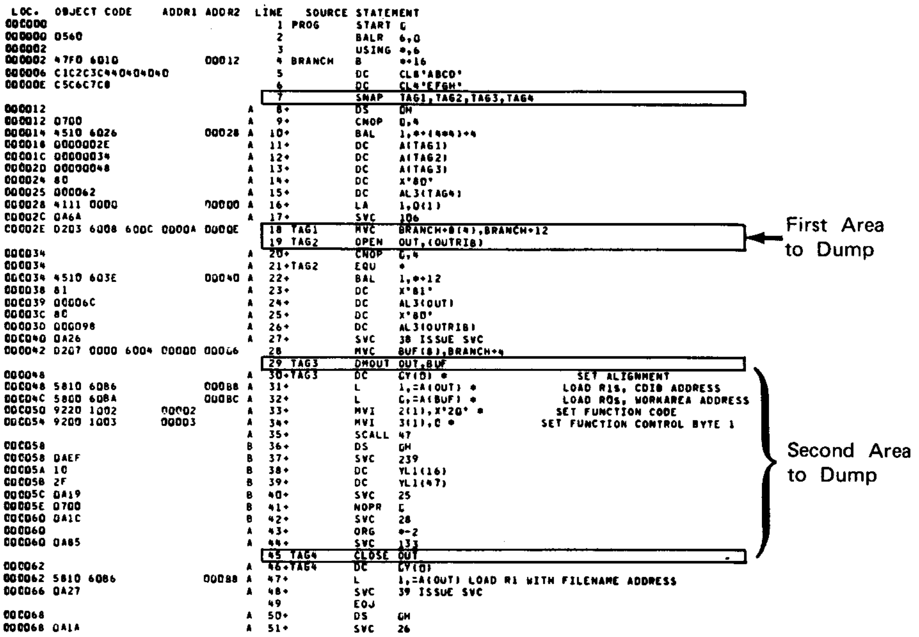
Remember, when using symbolic addresses, an even number of parameters must be used (start and end).

For example, if you coded the SNAP macroinstruction like this:

```
1        10     16
         SNAP   TAG1,TAG2,TAG3,TAG4
```

$$\texttt{SNAP}\quad \underbrace{\texttt{TAG1,TAG2}}_{\text{First Area}}\texttt{,}\underbrace{\texttt{TAG3,TAG4}}_{\text{Second Area}}$$

and placed it in your program like this:

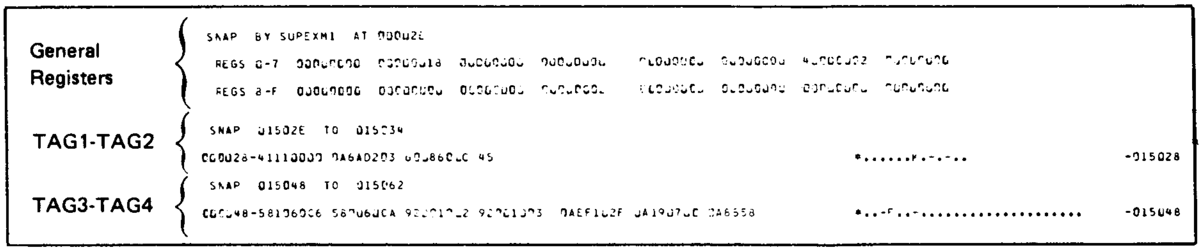


```
LOC.   OBJECT CODE     ADDR1 ADDR2  LINE   SOURCE STATEMENT
000000                                 1 PROG     START 0
000000 0560                            2          BALR  6,0
000002                                 3          USING *,6
000002 47F0 6010             00012     4 BRANCH   B     *+16
000006 C1C2C3C440404040                5          DC    CL8'ABCD'
00000E C5C6C7C8                        6          DC    CL4'EFGH'
                                       7          SNAP  TAG1,TAG2,TAG3,TAG4
000012                             A   8+         DS    0H
000012 0700                        A   9+         CNOP  0,4
000014 4510 6026             00028 A  10+         BAL   1,*+(4*4)+4
000018 0000002E                    A  11+         DC    A(TAG1)
00001C 00000034                    A  12+         DC    A(TAG2)
000020 00000048                    A  13+         DC    A(TAG3)
000024 80                          A  14+         DC    X'80'
000025 000062                      A  15+         DC    AL3(TAG4)
000028 4111 0000             00000 A  16+         LA    1,0(1)
00002C 0A6A                        A  17+         SVC   106
00002E D203 6008 600C 0000A 0000E     18 TAG1     MVC   BRANCH+8(4),BRANCH+12
                                      19 TAG2     OPEN  OUT,(OUTRIB)
000034                             A  20+         CNOP  0,4
000034                             A  21+TAG2     EQU   *
000034 4510 603E             00040 A  22+         BAL   1,*+12
000038 81                          A  23+         DC    X'81'
000039 00006C                      A  24+         DC    AL3(OUT)
00003C 80                          A  25+         DC    X'80'
00003D 000098                      A  26+         DC    AL3(OUTRIB)
000040 0A26                        A  27+         SVC   38 ISSUE SVC
000042 D207 0000 6004 00000 00066     28          MVC   BUF(8),BRANCH+4
                                      29 TAG3     DMOUT OUT,BUF
000048                             A  30+TAG3     DC    0Y(0) *                SET ALIGNMENT
000048 5810 6086             000B8 A  31+         L     1,=A(OUT) *          LOAD R1$, CDIB ADDRESS
00004C 5800 608A             000BC A  32+         L     0,=A(BUF) *          LOAD R0$, WORKAREA ADDRESS
000050 9220 1002       00002       A  33+         MVI   2(1),X'20' *       SET FUNCTION CODE
000054 9200 1003       00003       A  34+         MVI   3(1),0 *         SET FUNCTION CONTROL BYTE 1
                                   A  35+         SCALL 47
000058                             B  36+         DS    0H
000058 0AEF                        B  37+         SVC   239
00005A 10                          B  38+         DC    YL1(16)
00005B 2F                          B  39+         DC    YL1(47)
00005C 0A19                        B  40+         SVC   25
00005E 0700                        B  41+         NOPR  0
000060 0A1C                        B  42+         SVC   28
000060                             A  43+         ORG   *-2
000060 0A85                        A  44+         SVC   133
                                      45 TAG4     CLOSE OUT
000062                             A  46+TAG4     DC    0Y(0)
000062 5810 6086             000B8 A  47+         L     1,=A(OUT) LOAD R1 WITH FILENAME ADDRESS
000066 0A27                        A  48+         SVC   39 ISSUE SVC
                                      49          EOJ
000068                             A  50+         DS    0H
000068 0A1A                        A  51+         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 SUPEXM1  AT 00002E
 REGS 0-7  00000000  00000018  00000000  00000000   00000000  00000000  40000002  00000000
 REGS 8-F  00000000  00000000  00000000  00000000   00000000  00000000  00000000  00000000
```

TAG1-TAG2

```
SNAP  01502E  TO  015034
000028-41110000 0A6AD203 6008600C 45                                   *......K.-.-..          -015028
```

TAG3-TAG4

```
SNAP  015048  TO  015062
000048-58106086 5800608A 92201002 92001003  0AEF102F 0A190700 0A6558   *..-F..-....................   -015048
```

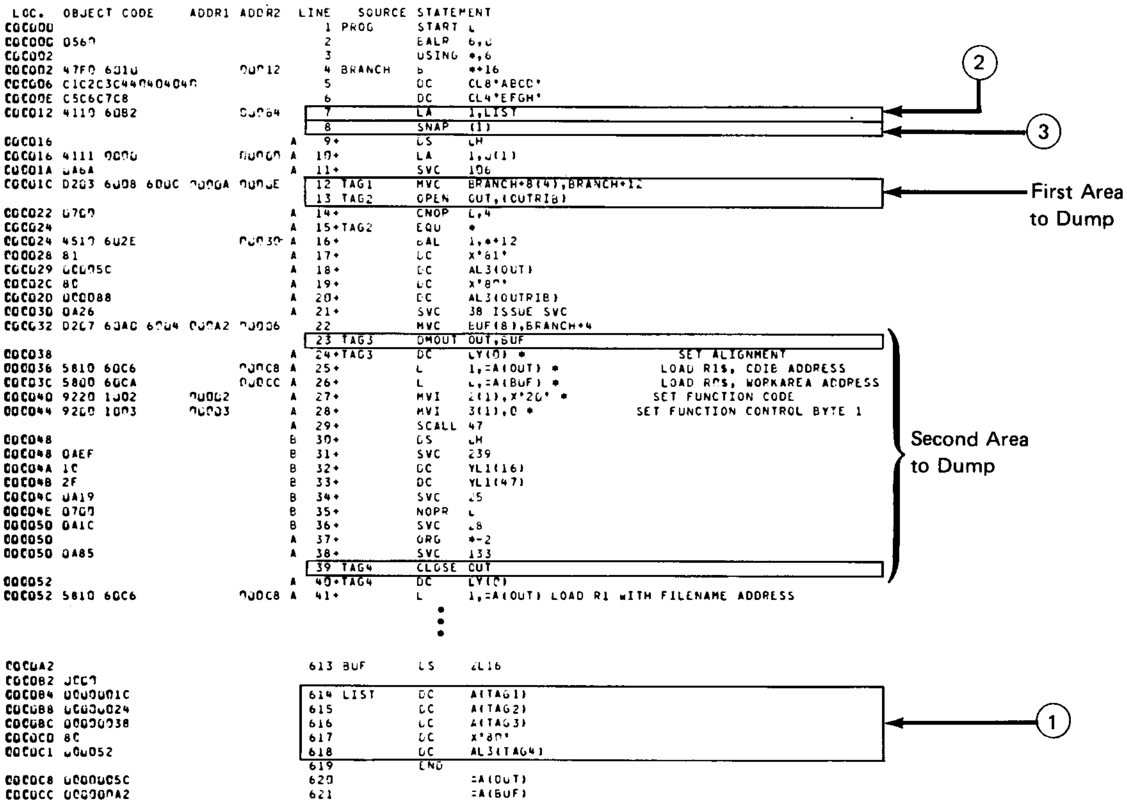
Notice that the SNAP macroinstruction is placed *before* the instruction areas to be dumped.

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 macroinstruction.

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 macroinstruction that this register contains the address by coding:

```
1        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           BALR  6,0
000002                                  3           USING *,6
000002 47F0 6010              00012     4 BRANCH    B     *+16
000006 C1C2C3C440404040                 5           DC    CL8'ABCD'
00000E C5C6C7C8                         6           DC    CL4'EFGH'
000012 4110 60B2              000B4     7           LA    1,LIST
                                        8           SNAP  (1)
000016                              A   9+          DS    0H
000016 4111 0000              00000 A  10+          LA    1,0(1)
00001A 0A6A                         A  11+          SVC   106
00001C D203 6008 600C 0000A 0000E      12 TAG1      MVC   BRANCH+8(4),BRANCH+12
                                       13 TAG2      OPEN  OUT,(OUTRIB)
000022 0700                         A  14+          CNOP  0,4
000024                              A  15+TAG2      EQU   *
000024 4510 602E              00030 A  16+          BAL   1,*+12
000028 81                           A  17+          DC    X'81'
000029 00005C                       A  18+          DC    AL3(OUT)
00002C 80                           A  19+          DC    X'80'
00002D 000088                       A  20+          DC    AL3(OUTRIB)
000030 0A26                         A  21+          SVC   38 ISSUE SVC
000032 D207 60A0 6004 000A2 00006      22           MVC   BUF(8),BRANCH+4
                                       23 TAG3      DMOUT OUT,BUF
000038                              A  24+TAG3      DC    0Y(0) *              SET ALIGNMENT
000036 5810 60C6              000C8 A  25+          L     1,=A(OUT) *        LOAD R1$, CDIB ADDRESS
00003C 5800 60CA              000CC A  26+          L     0,=A(BUF) *        LOAD R0$, WORKAREA ADDRESS
000040 9220 1002        00002       A  27+          MVI   2(1),X'20' *      SET FUNCTION CODE
000044 9200 1003        00003       A  28+          MVI   3(1),0 *        SET FUNCTION CONTROL BYTE 1
                                    A  29+          SCALL 47
000048                              B  30+          DS    0H
000048 0AEF                         B  31+          SVC   239
00004A 10                           B  32+          DC    YL1(16)
00004B 2F                           B  33+          DC    YL1(47)
00004C 0A19                         B  34+          SVC   25
00004E 0700                         B  35+          NOPR  0
000050 0A1C                         B  36+          SVC   28
000050                              A  37+          ORG   *-2
000050 0A85                         A  38+          SVC   133
                                       39 TAG4      CLOSE OUT
000052                              A  40+TAG4      DC    0Y(0)
000052 5810 60C6              000C8 A  41+          L     1,=A(OUT) LOAD R1 WITH FILENAME ADDRESS
                                                          •
                                                          •
                                                          •

0000A2                                613 BUF       DS    2L16
0000B2 0000
0000B4 0000001C                       614 LIST      DC    A(TAG1)
0000B8 00000024                       615           DC    A(TAG2)
0000BC 00000038                       616           DC    A(TAG3)
0000C0 80                             617           DC    X'80'
0000C1 000052                         618           DC    AL3(TAG4)
                                      619           END
0000C8 0000005C                       620                 =A(OUT)
0000CC 000000A2                       621                 =A(BUF)
```

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 bytes of TAG4 are specified by AL3.

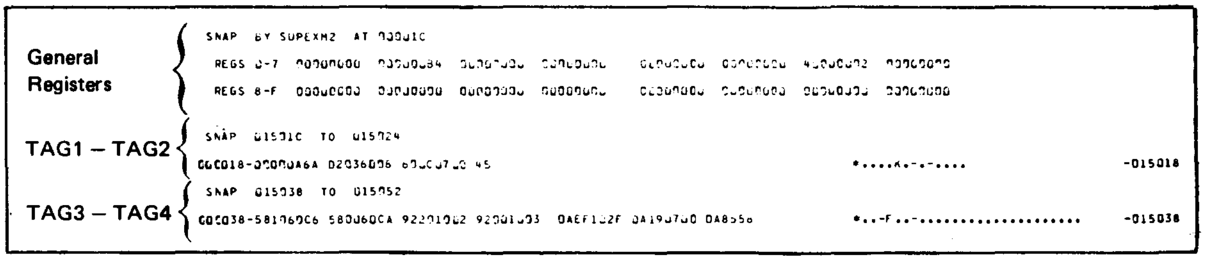
(2) Loads, the address of LIST into general register 1.

(3) Is the SNAP macroinstruction, 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 macroinstruction, 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
TAG1 – TAG2
TAG3 – TAG4

```
 SNAP  BY SUPEXM2  AT 00001C
  REGS 0-7  00000000  00000034  00000000  00000000   00000000  00000020  40000002  00000000
  REGS 8-F  00000000  00000000  00000000  00000000   00000000  00000000  00000000  00000000

 SNAP  01501C  TO  015024
00C018-00000A6A D2036006 60C070 45                                   *....K.-.-....          -015018
 SNAP  015038  TO  015052
00C038-581060C6 580060CA 92201002 92001003  0AEF102F 0A190700 0A8556    *..-F..-.....................   -015038
```

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 macroinstruction without any parameters,

```
1        10    16
_________________________
         SNAP
```

only the contents of the 16 general registers are printed.

NOTE:

The contents of general register 1 are destroyed by the SNAP or SNAPF macroinstruction. 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.

7.1.2. 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 macroinstruction causes this, and it is inserted in place of and acts as an EOJ macroinstruction. This means your job step runs to normal completion. Therefore, a DUMP macroinstruction terminates a job step without canceling it, unless, of course, something is wrong with your program.

Just as with the SNAP macroinstruction, you can initiate or suppress the dump at run time through the OPTION job control statement. However, with the DUMP macroinstruction, 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 dump analysis user guide/programmer reference, UP-8837 (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
_____________________________
// DVC 2Ø
// 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 macroinstruction for the job's print file.

If an OPTION job control statement is not present in the control stream, the DUMP macroinstruction acts as an EOJ macroinstruction. 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 macroinstruction is:

LABEL	△OPERATION△	OPERAND
[symbol]	DUMP	$\left[\begin{Bmatrix}\text{identification-code}\\ (\emptyset)\\ \emptyset\end{Bmatrix}\right]$

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, because each byte can hold one alphabetic character or two numeric characters, any combination that equals four bytes. Examples of this are:

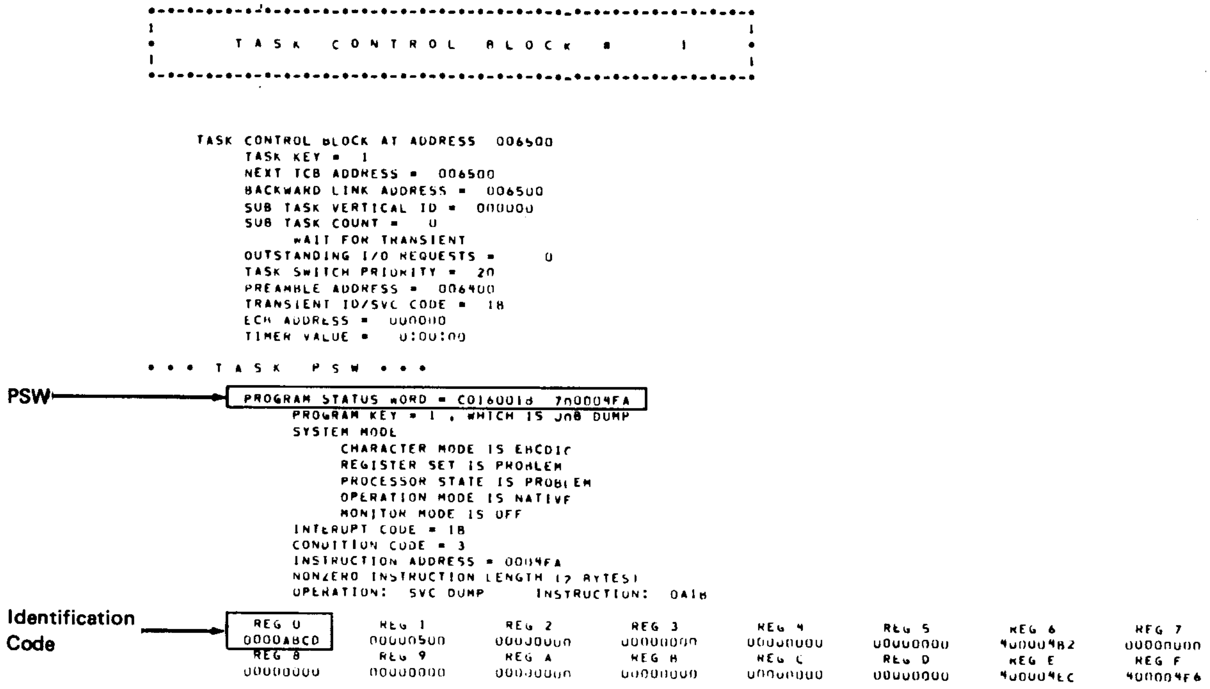
- 12345678
- A123456
- AB1234
- ABC12
- ABCD

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 macroinstruction, like this:

```
1            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):

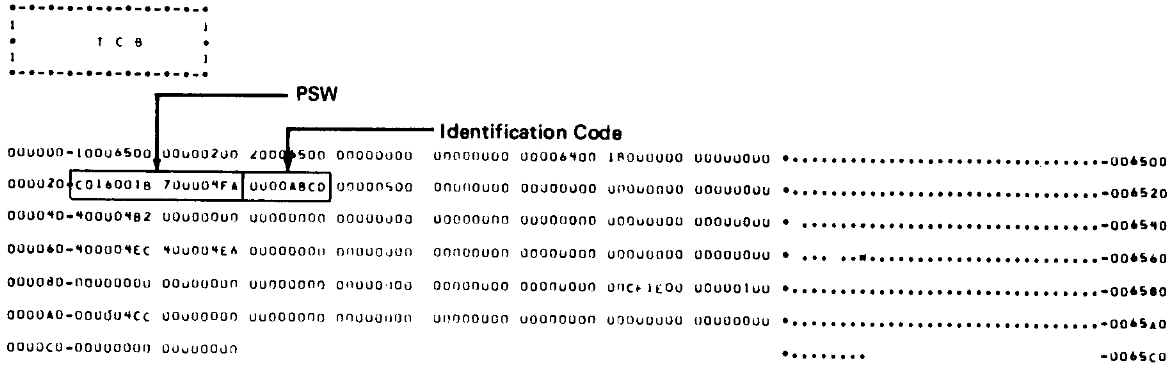


PSW

Identification Code

```
•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•
I                                                                 I
•       T A S K   C O N T R O L   B L O C K   *     1             •
I                                                                 I
•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•

      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 =  18
           ECB ADDRESS =  000010
           TIMER VALUE =   0:00:00

 * * *  T A S K   P S W  * * *

         PROGRAM STATUS WORD = C016001B  700004FA
                PROGRAM KEY = 1 , WHICH IS JOB DUMP
                SYSTEM MODE
                     CHARACTER MODE IS EBCDIC
                     REGISTER SET IS PROBLEM
                     PROCESSOR STATE IS PROBLEM
                     OPERATION MODE IS NATIVE
                     MONITOR MODE IS OFF
                INTERUPT CODE = 1B
                CONDITION CODE = 3
                INSTRUCTION ADDRESS = 0004FA
                NONZERO INSTRUCTION LENGTH (2 BYTES)
                OPERATION:  SVC DUMP      INSTRUCTION:  0A18

          REG 0      REG 1      REG 2      REG 3      REG 4      REG 5      REG 6      REG 7
        0000ABCD   00000500   00000000   00000000   00000000   00000000   400004B2   00000000
          REG 8      REG 9      REG A      REG B      REG C      REG D      REG E      REG F
        00000000   00000000   00000000   00000000   00000000   00000000   400004EC   400004F6
```

here:

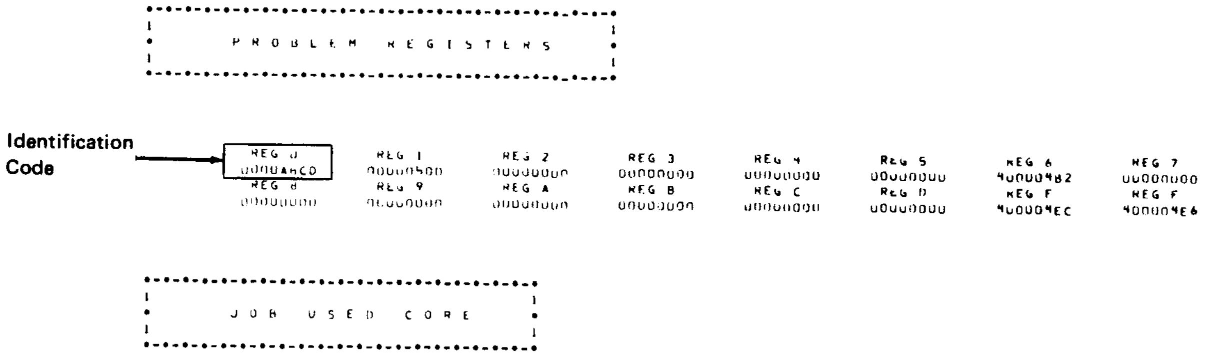


PSW

Identification Code

```
•-•-•-•-•-•-•-•-•-•
I                 I
•      T C B      •
I                 I
•-•-•-•-•-•-•-•-•-•

000000-10006500 00000200 20006500 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 400004EA 00000000 00000000  00000000 00000000 00000000 00000000 * ... ..*.........................*-006560
000080-00000000 00000000 00000000 00000100  00000000 00000000 00CF1E00 00000100 *................................*-006580
0000A0-000004CC 00000000 00000000 00000000  00000000 00000000 00000000 00000000 *................................*-0065A0
0000C0-00000000 00000000                                                      *........                        *-0065C0
```

and here:

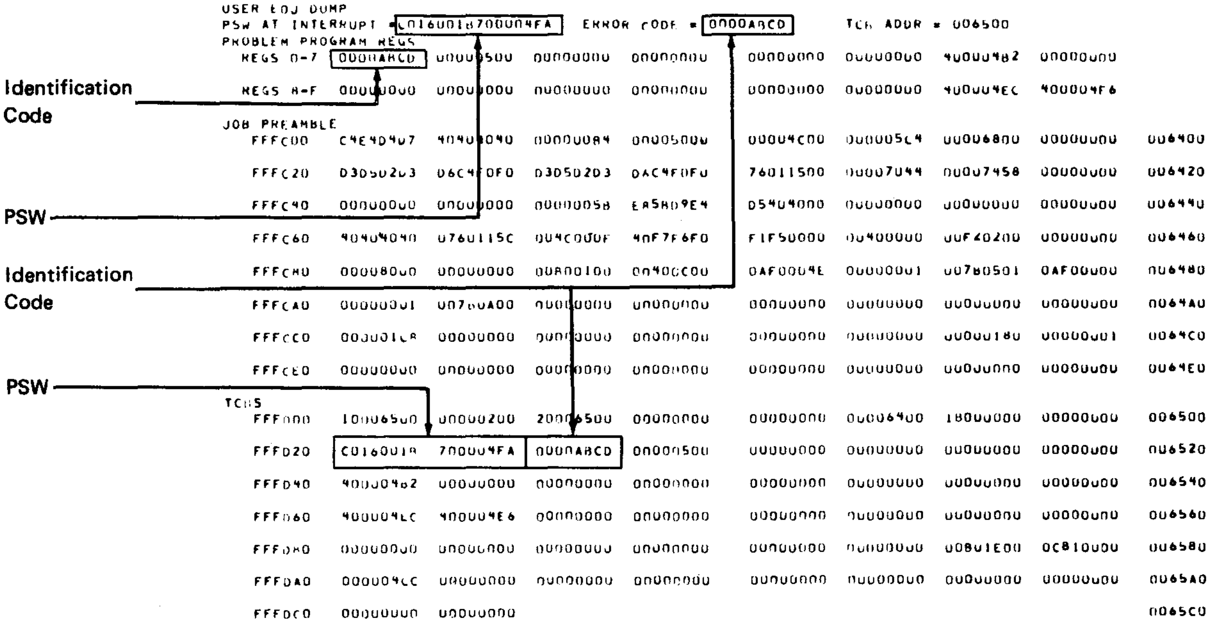


Identification Code

```
•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•
I                                                   I
•        P R O B L E M   R E G I S T E R S          •
I                                                   I
•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•

          REG 0      REG 1      REG 2      REG 3      REG 4      REG 5      REG 6      REG 7
        0000ABCD   00000500   00000000   00000000   00000000   00000000   400004B2   00000000
          REG 8      REG 9      REG A      REG B      REG C      REG D      REG F      REG F
        00000000   00000000   00000000   00000000   00000000   00000000   400004EC   400004E6

•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•
I                                           I
•        J O B   U S E D   C O R E          •
I                                           I
•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•-•
```

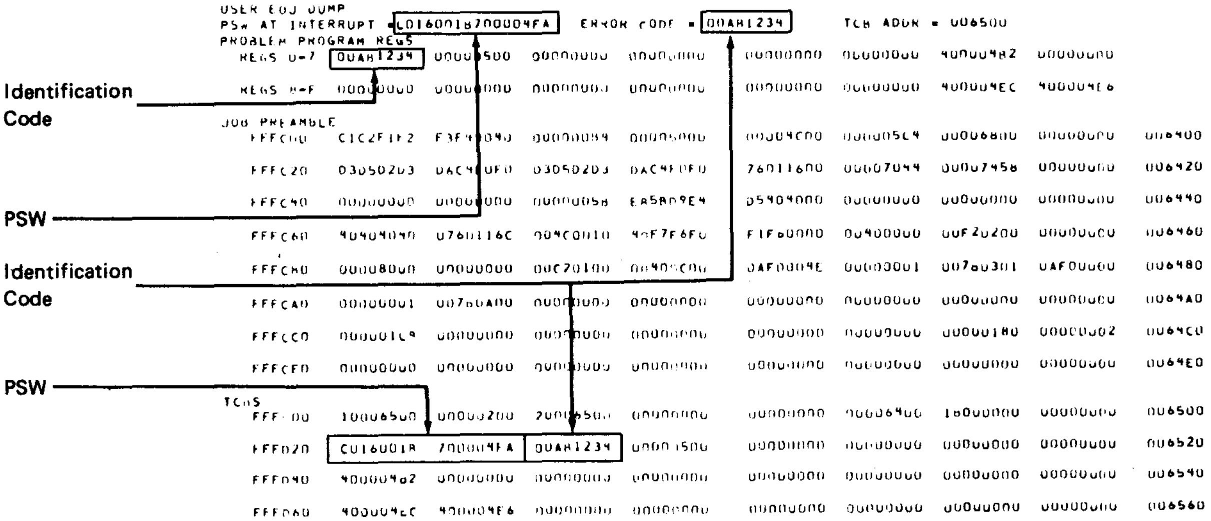
If we used the same DUMP macroinstruction (with identification code ABCD), but used an OPTION SYSDUMP job control statement, the identification code and the PSW again appear in the task control block (TCB) area and register 0 of the program registers of the portion of the dump belonging to the particular program (job step) that issued the DUMP macroinstruction.

If an OPTION DUMP job control statement was used, it would appear as:


Identification Code
PSW
Identification Code
PSW

```
USER EOJ DUMP
PSW AT INTERRUPT =C016001B700004FA   ERROR CODE =0000ABCD       TCB ADDR = 006500
PROBLEM PROGRAM REGS
  REGS 0-7  0000ABCD  00000500  00000000  00000000    00000000  00000000  40000482  00000000
  REGS 8-F  00000000  00000000  00000000  00000000    00000000  00000000  400004EC  400004F6
JOB PREAMBLE
   FFFC00   C4E4D4D7  40404040  00000084  00005000    00004C00  000005C4  00006800  00000000   006400
   FFFC20   D3D5D2D3  06C4F0F0  D3D5D2D3  0AC4F0F0    76011500  00007044  00007458  00000000   006420
   FFFC40   00000000  00000000  00000058  EA5BD9E4    D5404000  00000000  00000000  00000000   006440
   FFFC60   40404040  0760115C  004C000F  40F7F6F0    F1F50000  00400000  00F40200  00000000   006460
   FFFC80   00008000  00000000  00800100  00400C00    0AF0004E  00000001  00780501  0AF00000   006480
   FFFCA0   00000001  00760A00  00000000  00000000    00000000  00000000  00000000  00000000   0064A0
   FFFCC0   000001C8  00000000  00000000  00000000    00000000  00000000  00000180  00000001   0064C0
   FFFCE0   00000000  00000000  00000000  00000000    00000000  00000000  00000000  00000000   0064E0
TCBS
   FFFD00   10006500  00000200  20006500  00000000    00000000  00006400  18000000  00000000   006500
   FFFD20   C016001B  700004FA  0000ABCD  00000500    00000000  00000000  00000000  00000000   006520
   FFFD40   40000482  00000000  00000000  00000000    00000000  00000000  00000000  00000000   006540
   FFFD60   400004EC  400004E6  00000000  00000000    00000000  00000000  00000000  00000000   006560
   FFFD80   00000000  00000000  00000000  00000000    00000000  00000000  00801E00  0C810000   006580
   FFFDA0   000004CC  00000000  00000000  00000000    00000000  00000000  00000000  00000000   0065A0
   FFFDC0   00000000  00000000                                                                 0065C0
```

Just to see what an alphanumeric identification code printout looks like, here is an example using an OPTION DUMP job control statement with an identification code of AB1234:


Identification Code
PSW
Identification Code
PSW

```
USER EOJ DUMP
PSW AT INTERRUPT =C016001B700004FA   ERROR CODE =00AB1234       TCB ADDR = 006500
PROBLEM PROGRAM REGS
  REGS 0-7  00AB1234  00000500  00000000  00000000    00000000  00000000  40000482  00000000
  REGS 8-F  00000000  00000000  00000000  00000000    00000000  00000000  400004EC  400004E6
JOB PREAMBLE
   FFFC00   C1C2F1F2  F3F44040  00000084  00005000    00004C00  000005C4  00006800  00000000   006400
   FFFC20   D3D5D2D3  0AC4F0F0  D3D5D2D3  0AC4F0F0    76011600  00007044  00007458  00000000   006420
   FFFC40   00000000  00000000  00000058  EA5BD9E4    D5404000  00000000  00000000  00000000   006440
   FFFC60   40404040  0760116C  004C0010  40F7F6F0    F1F60000  00400000  00F20200  00000000   006460
   FFFC80   00008000  00000000  00C70100  00400C00    0AF0004E  00000001  00780301  0AF00000   006480
   FFFCA0   00000001  00760A00  00000000  00000000    00000000  00000000  00000000  00000000   0064A0
   FFFCC0   000001C9  00000000  00000000  00000000    00000000  00000000  00000180  00000002   0064C0
   FFFCE0   00000000  00000000  00000000  00000000    00000000  00000000  00000000  00000000   0064E0
TCBS
   FFFD00   10006500  00000200  20006500  00000000    00000000  00006400  18000000  00000000   006500
   FFFD20   C016001B  700004FA  00AB1234  00001500    00000000  00000000  00000000  00000000   006520
   FFFD40   40000482  00000000  00000000  00000000    00000000  00000000  00000000  00000000   006540
   FFFD60   400004EC  400004E6  00000000  00000000    00000000  00000000  00000000  00000000   006560
```

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 macroinstruction. 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 macroinstruction 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 macroinstruction included within your program.

7.1.3. Abnormal Termination

A main storage dump can also be obtained by using the CANCEL macroinstruction. However, in this case, the issuing program is terminated (and any subsequent programs in the job). This macroinstruction 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 macroinstruction can also be obtained by the console operator entering the CANCEL command at the system console.

7.2. 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 canceled 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 with *checkpoints*. You can then restart the job from the point where a checkpoint was taken rather than start the entire job all over again.

When a checkpoint is taken, a series of records are written to a SAT checkpoint file on disk, format label diskette, or tape. 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. 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. 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.)

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 macroinstruction to set a timer interrupt).

A user issuing checkpoints and using an open file with the FCB in main storage feature may receive errors on attempting a restart, as the file is not guaranteed to be available.

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 macroinstruction to access the data.

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).

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).

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.

7.2.1. How to Generate Checkpoint Records (CHKPT)

A series of checkpoint records is generated each time a CHKPT macroinstruction is executed in a program. These records must be written to a SAT file (defined by a DDCPF macroinstruction). The use of this macroinstruction and those needed to open and close the file are discussed later in this section (along with how the CHKPT macroinstruction is used in connection with these other macroinstructions). The format of the parameters of the CHKPT macroinstruction is:

LABEL	△OPERATION△	OPERAND
[symbol]	CHKPT	filename [,restart-addr][,,error-addr]

All the parameters of this macroinstruction are positional parameters.

The *filename* parameter specifies the symbolic address of the macroinstruction that defines the checkpoint record file. This macroinstruction is a DDCPF macroinstruction for a SAT file. 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 macroinstruction.

If an error occurs during the execution of the CHKPT macroinstruction, 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 7-1. Also listed are the possible restart error conditions and codes that may occur when trying to restart the job.

NOTE:

If you use the error-addr parameter, you must code two commas preceding it.

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 macroinstruction, 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 7-1. These error codes are also listed in the system messages programmer/operator reference manual, UP-8076 (current version).

Table 7—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.

7.2.2. Using a SAT File as a Checkpoint File

When using a SAT disk as a checkpoint file, 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, because you could lose data if wraparound occurs.

With tape or diskette SAT files, as with disk files, you cannot mix any of your data with checkpoint records; data and checkpoint records must go to separate files.

7.2.2.1. Estimate Space Requirements for a Disk Checkpoint File

Each checkpoint consists of a series of 256-byte records of the following type:

Data	Checkpoint Records	
Checkpoint header	1	
Prologue		
Preamble	1	
TCB	n	(2 TCB records per task)
Remainder of prologue	n	$\left(n = \frac{\text{remaining size}}{256}\right)$
File list	n	$\left(n = \frac{\text{number of open files}}{10}\right)$
User program	n	$\left(n = \frac{\text{user program}}{256}\right)$
File structures	n	(n = 4 x number of open files)

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, and six files are open at the time the checkpoint macro is issued. The checkpoint records would consist of the following:

Data		Checkpoint Records	Bytes
Checkpoint header		1	256
Prologue			
Preamble	(1 record - 256 bytes)		
TCB	(2 records - 512 bytes)		
Remainder of prologue	(2 records - 512 bytes)		
Total prologue		5	1280
File list		1	256
User program		32	8192
File structures		24	6144
	Totals	63	16128

Thus, a checkpoint for this program would consist of 63 records of 256 bytes each, or a total of 16,128 bytes. It should be noted that the previous formula results in an estimate of minimum space required. The actual space needed may be greater or less than the estimate, depending on the specific program environment (for example, the number of files open at the time of checkpoint).

If you allocate only two tracks, 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 four tracks. The current checkpoint would then overwrite the records recorded from an earlier checkpoint, while the most recent checkpoint would always be available.

7.2.2.2. Define, Open, and Close a SAT Checkpoint File (DDCPF, DCPOPN, DCPCLS)

When employing a SAT checkpoint file, a different group of macroinstructions are used to define, open, and close the file. We will now explain each.

In order to define a SAT checkpoint file, use the DDCPF macroinstruction. As you can see in the format, there are no parameters associated with this macroinstruction.

LABEL	△OPERATION△	OPERAND
filename	DDCPF	

There is only a *filename* in the label field. This *filename* is the symbolic address and is used as positional parameter 1 (*filename*) of both the CHKPT macroinstruction and the LFD job control statement.

Because the DDCPF macroinstruction does not generate executable code, it must be placed separate from your BAL instructions and imperative macroinstructions.

Use the DCPOPN macroinstruction to open the SAT checkpoint file (before the execution of the CHKPT macroinstruction). It has this format:

LABEL	△OPERATION△	OPERAND
[symbol]	DCPOPN	$\left\{\begin{array}{l}\text{filename}\\ \text{(1)}\end{array}\right\}$

The *filename* parameter specifies the symbolic address of the DDCPF macroinstruction 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 SAT checkpoint file, use the DCPCLS macroinstruction (after the last time the CHKPT macroinstruction is executed). The format is:

LABEL	△OPERATION△	OPERAND
[symbol]	DCPCLS	{ filename (1) }

The two parameter choices, *filename* or *(1)*, have the same meaning as the parameters of the DCPOPN macroinstruction: *filename* is the symbolic address of the DDCPF macroinstruction, and *(1)* indicates that the address is stored in general register 1.

Following is an example showing the relationship between the parameters of the CHKPT macroinstruction and the new macroinstructions we just discussed.

```
1        10    16
------------------------------------------------------------
         DCPOPN CKDISK
          .
          .
          .
         CHKPT  CKDISK,RESTART,,ERR1
          .
          .
          .
ERR1     (your error recovery routine)
          .
          .
          .
RESTART  (instruction where program is to begin if restarted)
          .
          .
          .
         DCPCLS CKDISK
          .
          .
          .
         EOJ
          .
          .
          .
CKDISK   DDCPF
```

7.3. 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 after 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 instruction 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 7-1.

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.

7.3.1. 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
// DVC 2Ø,17Ø
```

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.

7.3.1.1. 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 following job control statement must be inserted into the job you are monitoring:

```
// CC MO
```

If enough main storage is available at the time the job is scheduled, the monitor routine will be loaded at the time the job is initiated. If enough main storage is not available, it is possible that your job could be scheduled without the monitor routine. This could happen because the // CC job control statement cannot guarantee immediate loading of the monitor. If this happens and your job is scheduled without the monitor, you may select one of two options as follows:

1. Wait until more main storage is available before running your job.

2. Load the monitor routine into main storage by using the MO console command. The format of the command is:

   ```
   MO
   ```

   After typing in the command, you can verify that the monitor had been loaded by typing in the following console command:

   ```
   MI DA
   ```

   If the monitor routine has been successfully loaded, the symbiont SL$$MO will appear among the jobs and symbionts listed. When you have verified that the monitor is loaded, the job to be monitored should be run WITHOUT the // CC job control statement.

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 (7.3.5.3).

The control stream you submit when you want to monitor from the beginning of the job would look something like this:

```
      1          10          20
   ┌──────────────────────────────────────────────
1. │// JOB jobname
 ⎧ │        .
 ⎪ │        .
 ⎪ │        .
 ⎪ │   device assignment sets
2.⎨ │   and any other necessary
 ⎪ │   job control statements
 ⎪ │        .
 ⎪ │        .
 ⎪ │        .
 ⎩ │// CC MO
3. │// OPTION TRACE
4. │// EXEC program-name
 ⎧ │/$
5.⎨ │    monitor input-explained
 ⎪ │         in 7.3.2
 ⎩ │/*
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.

7.3.1.2. 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 7.3.5.3.) 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 (7.3.1.1) *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 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
------------------------------------
// ALTER phase-name,address,X'ØAØ4'
```

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 7.3.2, 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 7.3.1.1 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.)

```
1       10        20        30        40        50
_____________________________________________________________
// JOB jobname
// ALTER phase-name,address,X'ØAØ4'
           .
           .
           .
  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 7.3.1.1. 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 of suspending the executing program is the use of a DMDSP macroinstruction with a REP parameter. 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 DMDSP macroinstruction to resume processing under monitor control. The monitor input is exactly the same as when using the first method. That is, no /$ or /* encloses 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.) For more information on the DMDSP macroinstruction, refer to the consolidated data management macroinstructions user guide/programmer reference, UP-8826 (current version).

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 MO C 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.

7.3.2. 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 7.3.3. 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
____________________
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
```

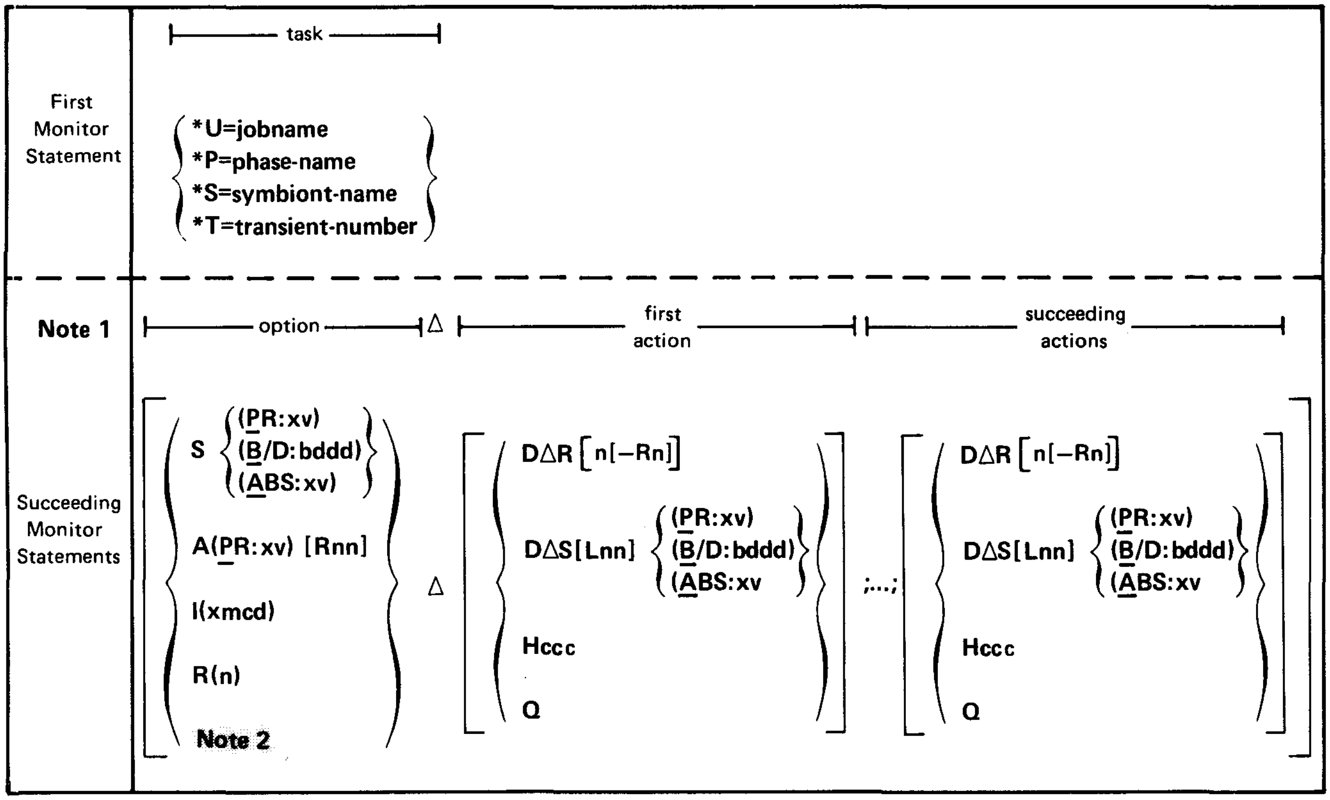
The last card used in the monitor input stream is a $ card. (Do not confuse this with the /$ job control statement, which indicates start of data.)

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 7.3.4, and the actions are defined in 7.3.5.

Figure 7-1 shows the format of the monitor statements.



NOTES:

1. If no option is specified, the monitor routine assumes a default option (7.3.4.5) and default display (7.3.5.1.3).

2. If no action is specified, the monitor routine produces a default display (7.3.5.1.3). 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 7—1. Monitor Input Format

7.3.3. 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:

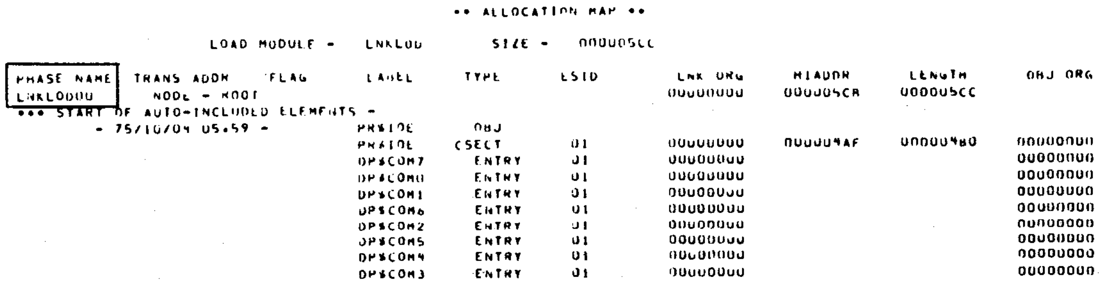
```
1            10            20
// JOB POCO
```

the monitor task statement would be:

```
*U=POCO
```

Because 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 **

                        LOAD MODULE -   LNKLOD        SIZE -   000005CC

PHASE NAME  TRANS ADDR   FLAG      LABEL      TYPE       ESID        LNK ORG     HIADDR      LENGTH      OBJ ORG
LNKLOD00      NODE - ROOT                                            00000000    000005CB    000005CC
*** START OF AUTO-INCLUDED ELEMENTS -
     - 75/10/04 05.59 -            PR$IOE     OBJ
                                   PR$IOE    CSECT       01          00000000    000004AF    000004B0    00000000
                                   DP$COM7    ENTRY      01          00000000                            00000000
                                   DP$COM0    ENTRY      01          00000000                            00000000
                                   DP$COM1    ENTRY      01          00000000                            00000000
                                   DP$COM6    ENTRY      01          00000000                            00000000
                                   DP$COM2    ENTRY      01          00000000                            00000000
                                   DP$COM5    ENTRY      01          00000000                            00000000
                                   DP$COM4    ENTRY      01          00000000                            00000000
                                   DP$COM3    ENTRY      01          00000000                            00000000
```

the monitor task statement is:

```
*P=LNKLODØØ
```

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

```
1        10        20
--------------------------
*S=SL$$U00
```

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 transient 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
```

NOTE:

If the job or symbiont you want to monitor has one primary task and one or more subtasks, you can monitor only the primary task.

7.3.4. 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 7.3.5. In this discussion, all the options are discussed first and 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 and 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 options you can specify, using the following format:

$$\left\{\begin{array}{l} \mathrm{S}\left\{\begin{array}{l}(\underline{\mathrm{P}}\mathrm{R:xv})\\ (\underline{\mathrm{B}}\mathrm{/D:bddd})\\ (\underline{\mathrm{A}}\mathrm{BS:xv})\end{array}\right\}\\ \mathrm{A}(\underline{\mathrm{P}}\mathrm{R:xv})[\mathrm{Rnn}]\\ \mathrm{I(xmcd)}\\ \mathrm{R(n)}\end{array}\right\}$$

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 subsections.

7.3.4.1. 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)

7.3.4.1.1. 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 O_{16} to $FFFFFF_{16}$. Notice that it is separated from the *PR* by a colon. For example:

```
1        10        20
______________________
S(PR:43C)
```

Because this format is shown with an underline under the P, you could also code it as:

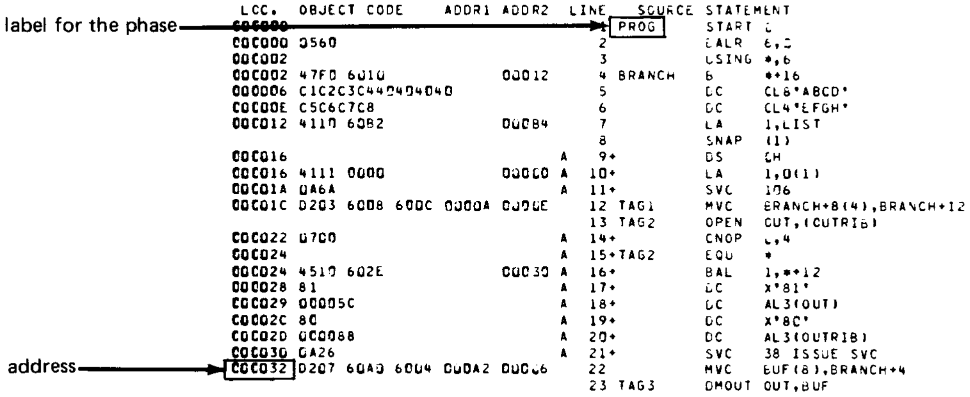
```
S(P:43C)
```

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).

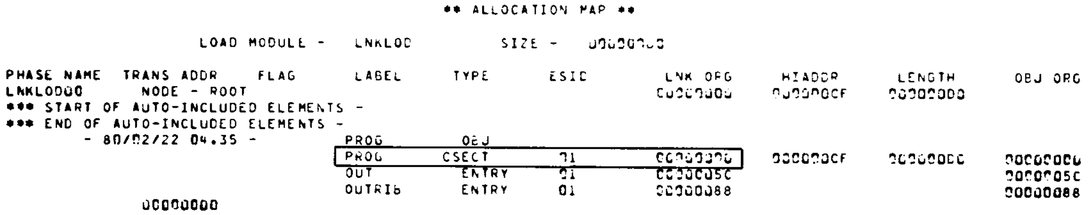
For your program, a phase of your program, or a symbiont, a program-relative address is relative to the start of the load module. In other words, the address in the assembly listing must be added to the link origin (LNK ORG) address for the control section (CSECT) of your program. This shows up in the allocation map produced by the linkage editor.

For example, if you wanted to monitor from address 32 in this program listing:


label for the phase
address

```
 LCC.  OBJECT CODE      ADDR1 ADDR2  LINE   SOURCE STATEMENT
000000                                 1  PROG     START 0
000000 0560                            2           BALR  6,0
000002                                 3           USING *,6
000002 47F0 6010             00012     4  BRANCH   B     *+16
000006 C1C2C3C440404040                5           DC    CL8'ABCD'
00000E C5C6C7C8                        6           DC    CL4'EFGH'
000012 4110 60B2             000B4     7           LA    1,LIST
                                       8           SNAP  (1)
000016                             A   9+          DS    0H
000016 4111 0000             00000 A  10+          LA    1,0(1)
00001A 0A6A                        A  11+          SVC   106
00001C D203 6008 600C 0000A 0000E     12  TAG1     MVC   BRANCH+8(4),BRANCH+12
                                      13  TAG2     OPEN  OUT,(OUTRIB)
000022 0700                        A  14+          CNOP  0,4
000024                             A  15+TAG2      EQU   *
000024 4510 602E             00030 A  16+          BAL   1,*+12
000028 81                          A  17+          DC    X'81'
000029 00005C                      A  18+          DC    AL3(OUT)
00002C 80                          A  19+          DC    X'80'
00002D 000088                      A  20+          DC    AL3(OUTRIB)
000030 0A26                        A  21+          SVC   38 ISSUE SVC
000032 D207 60A0 6004 000A2 00006     22           MVC   BUF(8),BRANCH+4
                                      23  TAG3     DMOUT OUT,BUF
```

you would have to look at the allocation map for the LNK ORG of the CSECT (4B0):



```
                                  ** ALLOCATION MAP **

                LOAD MODULE -   LNKLOD          SIZE -   00000000

PHASE NAME  TRANS ADDR    FLAG      LABEL      TYPE       ESIC        LNK ORG      HIADDR      LENGTH       OBJ ORG
LNKLOD000       NODE - ROOT                                           00000000     000000CF    00000000
*** START OF AUTO-INCLUDED ELEMENTS -
*** END OF AUTO-INCLUDED ELEMENTS -
       - 80/02/22 04.35 -           PROG       OBJ
                                    PROG       CSECT      01          000004B0     000000CF    000000DC     00000000
                                    OUT        ENTRY      01          0000005C                              0000005C
                                    OUTRIB     ENTRY      01          00000088                              00000088
             00000000
```

and add them together, producing 4E2 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 7.3.3.

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.

7.3.4.1.2. Base/Displacement Address (B/D)

To use the base/displacement address method for the storage reference option, you need this format:

```
S(B/D:bddd)
```

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:

```
1       10       20
___________________________
S(B/D:4B29)
```

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.

7.3.4.1.3. 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:

```
S(ABS:35AE)
```

7.3.4.2. 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 $FFFFFF_{16}$). 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:

```
1       10      20
________________________
A(PR:CØ2)
```

or

```
A(P:CØ2)
```

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):

```
A(P:CØ2)RØE
```

monitor action begins when the instruction at program relative address C02 is reached, and continues for 14 bytes (0E). 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 0E was omitted, and it was coded as this:

```
A(P:CØ2)RE
```

monitoring would continue for 224 bytes to program relative address CE2.

7.3.4.3. Instruction Sequence Option (I)

This option requests the monitor routine to taken 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

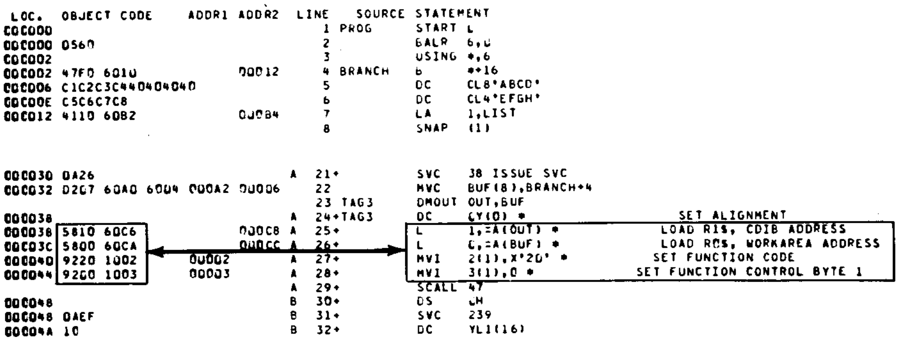
For example, if you want monitor action to start when a supervisor call instruction for supervisor routine 31 occurs (SVC 31 in machine code = 0A1F), code it as:

```
1       10       20
I(ØA1F)
```

If you want monitor action whenever *any* branch on condition instruction is reached (hexadecimal code = 47), you would code:

```
I(47)
```

But, if you want monitor action to occur whenever the following sequence of instructions occurs (even though we are showing a series of inline expansion codes):



```
 LOC.  OBJECT CODE     ADDR1 ADDR2  LINE   SOURCE STATEMENT
COC000                                1 PROG     START L
00C000 0560                           2          BALR  6,U
COC002                                3          USING *,6
00C002 47F0 6010             00012    4 BRANCH   B     *+16
COC006 C1C2C3C440404040               5          DC    CL8'ABCD'
COC00E C5C6C7C8                       6          DC    CL4'EFGH'
00C012 4110 60B2             0J0B4    7          LA    1,LIST
                                      8          SNAP  (1)


00C030 0A26                        A  21+        SVC   38 ISSUE SVC
00C032 D2C7 60A0 6004 C00A2 0U006     22         MVC   BUF(8),BRANCH+4
                                      23 TAG3    DMOUT OUT,BUF
000038                             A  24+TAG3    DC    CY(0) *              SET ALIGNMENT
000038 5810 60C6             0J0C8 A  25+        L     1,=A(OUT) *        LOAD R1S, CDIB ADDRESS
00C03C 5800 60CA             0U0CC A  26+        L     C,=A(BUF) *        LOAD RCS, WORKAREA ADDRESS
000040 9220 1002       0U002       A  27+        MVI   2(1),X'20' *      SET FUNCTION CODE
000044 92C0 1003       00003       A  28+        MVI   3(1),0 *         SET FUNCTION CONTROL BYTE 1
                                   A  29+        SCALL 47
000048                             B  30+        DS    UH
000048 0AEF                        B  31+        SVC   239
00004A 10                          B  32+        DC    YL1(16)
```

you would code it as:

```
I(581Ø6ØC658ØØ6ØCA922Ø1ØØ292ØØ1ØØ3)
```

7.3.4.4. Register Change Option (R)

This option requests the monitor routine to take action whenever a specified register is changed. It has only one format:

```
R(n)
```

The n is the hexadecimal number of the registers to be checked. Because this monitor action is triggered by the comparison of the current register contents to its previous contents, the instruction displayed when the change occurs will be the instruction following the instruction that caused the change. This is similar to the storage reference option for a program relative address (7.3.4.1.1.), which also occurs after the storage location changes. (Remember, it is possible to get two displays from a single storage reference option: one before and one after the area changes.)

For example, if you want monitor action to take place whenever the contents of register 10 change, you would code:

```
R(A)
```

Because the contents of registers are changed frequently during the course of most programs, the register change option may produce a large amount of display printout.

7.3.4.5. No Option Specified? You Get a Default

If you omit the option specifications (if your monitor input consists only of the *U, *P, *S, or *T card, and the $ card), the monitor routine interrupts each instruction in the task before its execution and prints out pertinent information at that point in the program. The processor then executes the interrupted instruction (identified on the printout as the NEXT INST). The succeeding instruction is then interrupted, the printout produced, and the instruction executed. This interrupt, printout, and execution pattern is performed for each instruction processed. This could require excessive processor time and could produce a huge printout of unneeded information. Therefore, you would use the default option only for special cases.

The program information printed is the same as for the default display described in 7.3.5.1.3, except that there is no option mentioned on the printout because one is not specified.

7.3.5. 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△R, D△S, H* or *Q* (plus a default), as shown in this format:

$$\left\{\begin{array}{l} \mathtt{D\triangle R[\,n[-Rn\,]\,]} \\ \mathtt{D\triangle S[\,Lnn\,]}\left\{\begin{array}{l}\mathtt{(\underline{P}R:xv)}\\ \mathtt{(\underline{B}/D:bddd}\\ \mathtt{(\underline{A}BS:xv)}\end{array}\right\} \\ \mathtt{Hccc} \\ \mathtt{Q} \end{array}\right\}$$

NOTE:

If no action is specified, the monitor routine produces a default display (7.3.5.1.3).

The *D△R* and *D△S* actions (for display register or display storage) print out program information, including specified registers *(D△R)* or storage *(D△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.

7.3.5.1. Display Actions

There are two types of display specifications: register display (D△R) and storage display (D△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.

7.3.5.1.1. Register Display (D△R)

If you select this type of action, you get the following items:

1. The jobname, TCB address, and program base address. Because 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):

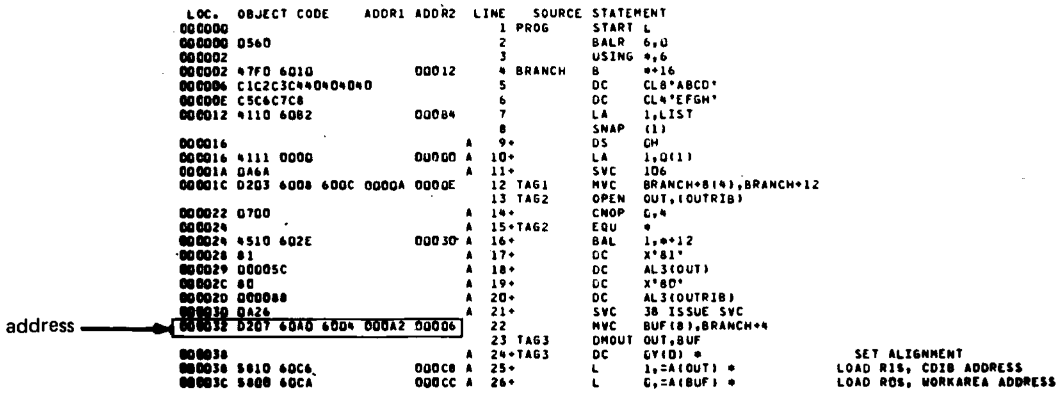
```
D△R[n[-Rn]]
```

- D△R, which prints the contents of all 16 general registers

- D△Rn, which prints a specific register, with *n* being the hexadecimal number (0–F) of the register you want

- D△Rn–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 32 in this listing (remember, the assembly address (32) must be added to the LNK ORG address, which in this case is 0, to obtain the program relative address – 32):

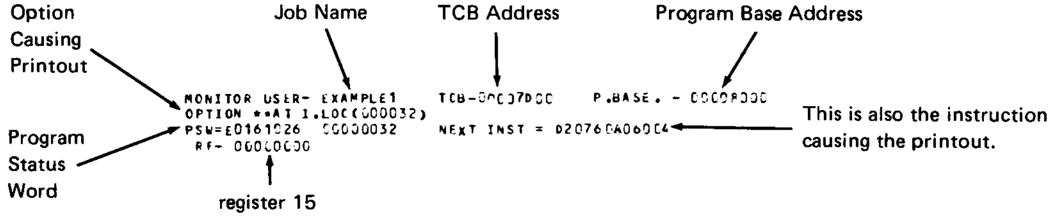


```
LOC.   OBJECT CODE       ADDR1 ADDR2  LINE   SOURCE STATEMENT
000000                                  1 PROG     START L
000000 0560                             2          BALR  6,0
000002                                  3          USING *,6
000002 47F0 6010               00012    4 BRANCH   B     *+16
000006 C1C2C3C440404040                 5          DC    CL8'ABCD'
00000E C5C6C7C8                         6          DC    CL4'EFGH'
000012 4110 6082               00084    7          LA    1,LIST
                                        8          SNAP  (1)
000016                               A  9+         DS    0H
000016 4111 0000               00000 A 10+         LA    1,0(1)
00001A 0A6A                          A 11+         SVC   106
00001C D203 6008 600C 0000A 0000E      12 TAG1     MVC   BRANCH+8(4),BRANCH+12
                                       13 TAG2     OPEN  OUT,(OUTRIB)
000022 0700                          A 14+         CNOP  0,4
000024                               A 15+TAG2     EQU   *
000024 4510 602E               00030 A 16+         BAL   1,*+12
000028 81                            A 17+         DC    X'81'
000029 00005C                        A 18+         DC    AL3(OUT)
00002C 80                            A 19+         DC    X'80'
00002D 000088                        A 20+         DC    AL3(OUTRIB)
000030 0A26                          A 21+         SVC   38 ISSUE SVC
000032 D207 60A0 6004 000A2 00006      22          MVC   BUF(8),BRANCH+4
                                       23 TAG3     DMOUT OUT,BUF
000038                               A 24+TAG3     DC    0Y(0) *               SET ALIGNMENT
000038 5810 60C6               000C8 A 25+         L     1,=A(OUT) *         LOAD R1S, CDIB ADDRESS
00003C 5800 60CA               000CC A 26+         L     0,=A(BUF) *         LOAD R0S, WORKAREA ADDRESS
```

you would code:

```
1        10        20
A(PR:32) D RF
```

and your monitor printout would look like this:

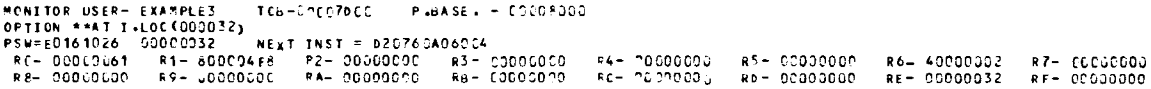


```
MONITOR USER- EXAMPLE1   TCB-00007D00   P.BASE. - 00008000
OPTION **AT I.LOC(000032)
PSW=E0161026  00000032   NEXT INST = D2076DA060C4
 RF- 00000000
```

If, for the same program, you coded:

```
A(PR:32) D R
```

the contents of all 16 registers (plus items 1 through 4) are displayed, like this:



```
MONITOR USER- EXAMPLE3    TCB-00007DCC    P.BASE. - 00008000
OPTION **AT I.LOC(000032)
PSW=E0161026  00000032    NEXT INST = D2076DA060C4
 R0- 00000061   R1- 800004F8   R2- 0000000C   R3- 00000000   R4- 00000000   R5- 00000000   R6- 40000002   R7- 00000000
 R8- 00000000   R9- 0000000C   RA- 00000000   RB- 00000000   RC- 00000000   RD- 00000000   RE- 00000032   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 7.3.5.3) 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:

```
1        10        20
_____________________________________
A(PR:32) D R;Q
```

Notice that a semicolon is used to separate the actions.

7.3.5.1.2. Storage Display (D△S)

Most of the information provided by a storage display type of action is similar to that of a register display (7.3.5.1.1): 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:

```
D△S[Lnn](PR:xv)
D△S[Lnn](B/D:bccc)
D△S[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 7.3.4.1, 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:

```
1        10        20
I(47) D SL14(PR:3C)
```

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:

```
I(47) D S(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 S(ABS:35AE)
```

If you wanted only four bytes at absolute address 35AE whenever any branch condition is reached, code:

```
I(47) D SLØ4(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 S(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.

7.3.5.1.3. 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 7.3.5.1.1, 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
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 (7.3.4.5). The only difference between the default display caused by omitting the option and the default display caused by omitting the action is that the option causing the display is not printed.

7.3.5.2. Halt Action (H)

This action, like the other actions, prints out items 1, 2, 3, and 4 (detailed in 7.3.5.1.1). 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:

```
1        10        20
______________________________
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.

7.3.5.3. 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:

```
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 7-2 summarizes the program information that is displayed by each action.

Table 7—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.

7.3.6. 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. A CANCEL command should not be issued while monitoring is in progress.

7.4. 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 7–3 summarizes the debugging aids described in the following subsections.

Table 7–3. Summary of System Debugging Aids (Part 1 of 2)

Function	Use	Console Command	Results
Pseudo monitor*	To identify the routine changing a particular byte	SE HA,PM,address [,job-name]	HPR 99130202 (Press START to continue)
Resident monitor*	To identify the instruction changing a particular byte	SE HA,RM, address[,job-name]	HPR 991304 (Press START to continue)
Verify bytes 0-B*	To identify the routine destroying low-order storage	Included in supervisor debug option	HPR 99130303 (Take SYSDUMP to find problem)
History tables*	To provide some recent history in SYSDUMPs	Included in supervisor debug option	Continuous updating of tables
Halt on transient load	To halt if and when a particular transient is loaded	SE HA,TL,hex-id	HPR 990COC (Press START to continue)
Halt on transient call*	To halt if and when a particular transient is called	SE HA,TC,hex-id	HPR 990COD (Press START to continue)
Halt on transient exit*	To halt if and when a particular transient exits	SE HA,TE,hex-id	HPR 990COE (Press START to continue)
Halt on symbiont load	To halt if and when a particular symbiont phase is loaded	SE HA,SY,idnn	HPR 997C (Press START to continue)
Halt on shared code call*	To halt if and when certain (or all) shared code modules are called	SE HA,SC[,{module-name prefix.}]	HPR 991D01 (Press START to continue)
Halt on shared code return*	To halt if and when certain (or all) shared code modules return	SE HA,SR[,{module-name prefix.}]	HPR 991D02 (Press START to continue)
Halt on shared code return with error*	To halt if and when certain (or all) shared code modules return with error	SE HA,SE[,{module-name prefix.}]	HPR 991D03 (Press START to continue)

Table 7—3. Summary of System Debugging Aids (Part 2 of 2)

Function	Use	Console Command	Results
Pause on shared code call*	To pause a task if and when certain (or all) shared code modules are called	SE PA,SC$\left[,\begin{Bmatrix}\text{module-name}\\ \text{prefix.}\end{Bmatrix}\right]$	SE25 console message (Enter C to continue)
Pause on shared code return*	To pause a task if and when certain (or all) shared code modules return	SE PA,SR$\left[,\begin{Bmatrix}\text{module-name}\\ \text{prefix.}\end{Bmatrix}\right]$	SE25 console message (Enter C to continue)
Pause on shared code return with error*	To pause a task if and when certain (or all) shared code modules return with error	SE PA,SE$\left[,\begin{Bmatrix}\text{module-name}\\ \text{prefix.}\end{Bmatrix}\right]$	SE25 console message (Enter C to continue)
PIOCS debug option	To identify checksum errors or internal PIOCS problems	SE DE,IO	HPR 990F
Transient debug option	To halt on transient errors (100 - 1FF)	SE DE,TR	HPR 99080800
Loader debug option	To halt on loader errors (52-5F)	SE DE,LD	HPR code 991500 (Press RUN to continue.)
Shared code debug option	To halt on shared code errors	SE DE,SC	HPR code 990809 (Press RESTART to take a SYSDUMP and continue.) HPR 99130A when dynamic buffer pool links are destroyed.
Dynamic buffer debug option*	To halt on dynamic buffer overflow	SET DE,DB	HPR code 991300
Screen format coordinator input/output debug option	To take a snapshot dump of all input and output buffer blocks when using the screen format coordinator	SET DE,INO	Writes snapshot dump to job log
Screen format coordinator format/input/output debug option	To take a snapshot dump of the format block; the input buffer (on input operations); the output buffer (on output operations); and, if errors occur, the screen format coordinator blocks	SET DE,FS	Writes snapshot dump to job log or printer system
Reset pause options	Resets all SE PA commands	SE PA,OFF	None
Reset halts	Resets all SE HA commands	SE HA,OFF	None
Reset debug options	Resets all SE DE commands	SE DE,OFF	None

*Supervisor debug option required at IPL

7.4.1. Supervisor Debug Option

The supervisor debug option is set at initial program load (IPL) time by entering D as the option 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 over 2K) to support the supervisor debug option.

The following functions are provided:

- A normal halt (HPR code 99130101) between IPL and supervisor initialization. This indicates the supervisor has been successfully loaded by the IPL and also allows changes to be made to the supervisor prior to loading the supervisor initialization load module. Normally, however, you should simply press the START key to continue.

- A pseudo monitor to detect when any byte within the supervisor has been changed. This function is activated via the SE HA,PM console command (Table 7-3). The byte specified is checked on every interrupt and on every pass through the switcher. When changed, the supervisor halts (HPR code 99130202) without restoring the original contents. If you want to continue, simply press START. The new value becomes the original value and the supervisor will halt if the byte is changed again.

- A resident supervisor monitor to detect when any byte in main storage has been changed. This function is activated by the SE HA,RM console command (Table 7-3). The specified address (either absolute or relative to the preamble of a currently active job) is monitored on every instruction executed by the operating system, including interrupt processing, transients, symbionts, shared code, and job control. The only code not monitored is nonkey 0 code (user jobs) because the hardware storage protection feature prevents such code from destroying any part of the supervisor.

When the specified byte is changed, the resident monitor halts (HPR code 991304) without restoring the original contents. The double word at absolute location 80_{16} contains the PSW at the time the byte was changed. If you want to continue, press START. The new value then 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 much more slowly than it normally would. This is because every instruction executed now requires about 10 additional instructions to verify the byte being monitored. For this reason, you should only use the resident monitor for short periods of time.

To turn off the resident monitor, use the SE HA,OFF console command. The resident monitor must not be used when the normal monitor (7.3) is active.

- 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 (99130303). Although you may continue past this HPR by pressing START, you should take a SYSDUMP here to determine why these bytes are being altered.

- History tables that provide the following information:

  - Critical event history table. This shows the last 16 *critical events* that occurred in the supervisor and the value in the interval timer register (ITR) at the time they occurred. These are listed in 4-byte entries as follows:

    Byte 0 = EBCDIC event code:

    I (X'C9') = Interrupt

    S (X'E2') = Switcher call

    T (X'E3') = Task given control by switcher

    L (X'D3') = Transient load

    O (X'D6') = Transient issued a call for an overlay

    R (X'D9') = Transient release

    Byte 1 = Interrupt type (if event code = I)

    0 = I/O

    1 = Exigent machine check

2 = Program exception

3 = SVC

4 = External interrupt (timer or interrupt key)

5 = Repressible machine check

Bytes 2–3 = Value in ITR when event occurred. The ITR decrements once every millisecond.

- Dynamic buffer management history table. This shows the last 16 requests to either obtain a buffer (GETBUF) from a dynamic buffer pool (DBP) or release a buffer (FREEBUF). These are listed in 8-byte entries as follows:

Byte 0	=	EBCDIC G (X'C7') if GETBUF
	=	EBCDIC F (X'C6') if FREEBUF
Bytes 1–3	=	Buffer address
Byte 4	=	Buffer characteristics:

Bits 0–1	=	0
Bit 2	=	1 indicates buffer is allocated, and bits 5–7 indicate the software using the buffer.
Bit 3	=	0
Bit 4	=	1 if buffer is under storage protection
Bits 5–7	=	5 if shared code local storage
	=	4 if data management buffer
	=	3 if external TCB
	=	2 if stack frame
	=	1 if shared code
	=	0 if none of the above

Bytes 5–7	=	Buffer size, in bytes, if GETBUF
	=	TCB address if FREEBUF

- TCB history table. This shows the absolute addresses of the last 6 TCBs given control by the switcher. The address of the last active TCB is found in the system information block (relative location X'94'), not in this table.

- Interrupt history table. This shows the PSW (8 bytes) at the time of the last 8 interrupts. Bits 4–7 of each PSW are set to an interrupt ID, as follows:

  0 = I/O interrupt

  1 = Exigent machine check interrupt

  2 = Program exception interrupt

  3 = SVC interrupt

  4 = External interrupt (timer or interrupt key)

  5 = Repressible machine check

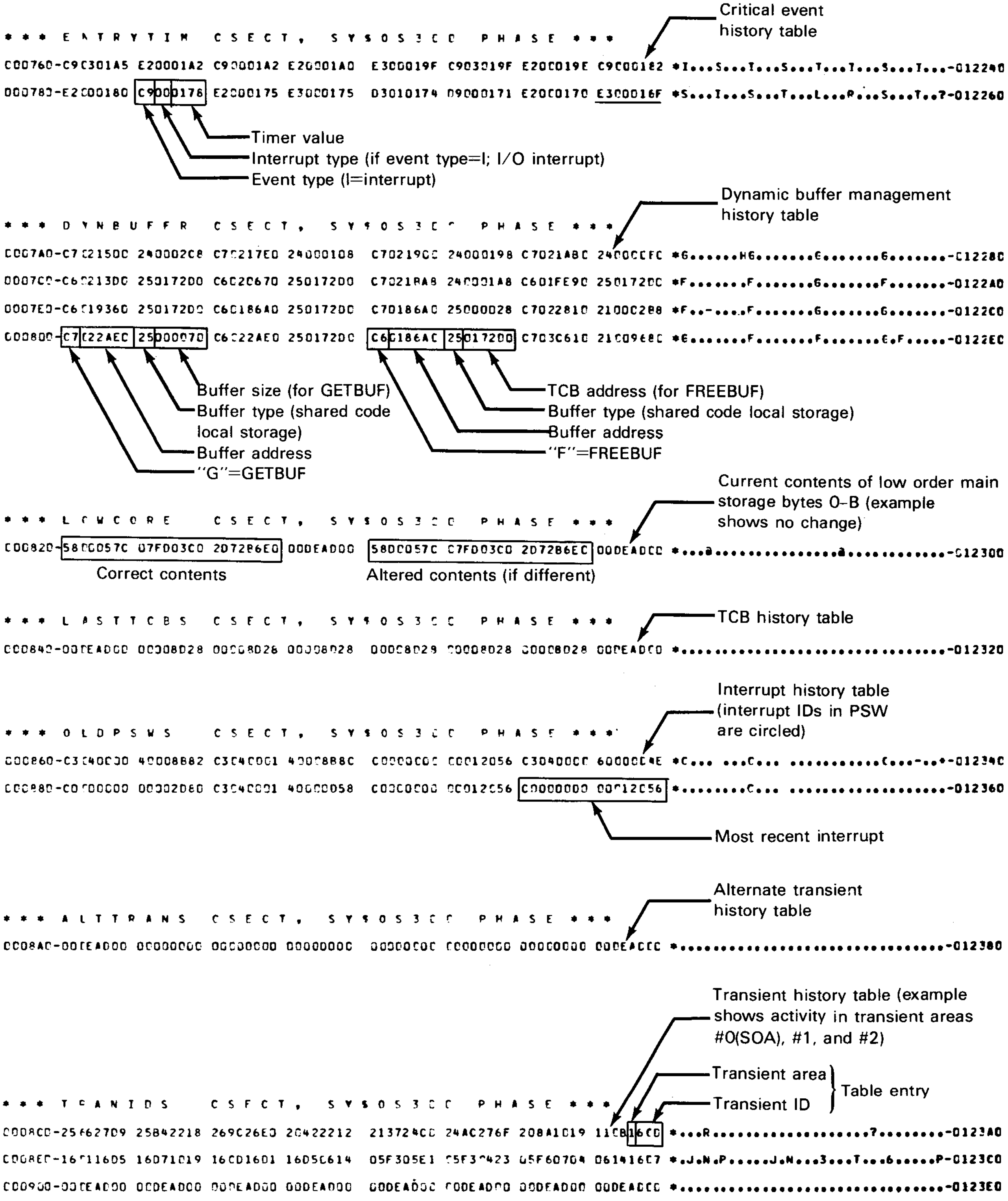
- Alternate transient history table. This shows the transient IDs of the last 12 transients loaded from the alternate transient file ($Y$TRANA). This table would normally be all zeros.

- 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 previously described reside near the end of the supervisor. They can be easily identified in a SYSDUMP by the CSECT names, as follows:

ENTRYTIM	–	Critical event history table
DYNBUFFR	–	Dynamic buffer management history table
LOWCORE	–	Correct and altered contents of the low order 12 bytes in main storage
LASTTCBS	–	TCB history table
OLDPSWS	–	Interrupt history table
ALTTRANS	–	Alternate transient history table
TRANIDS	–	Transient history table

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.



```
* * *  E N T R Y T I M   C S E C T ,   S Y S O S 3 C C   P H A S E  * * *
C00760-C9C301A5 E20001A2 C90001A2 E20001A0  E300019F C903019F E20C019E C9C00182 *I...S...I...S...T...I...S...I...-012240
000780-E2C00180 C9000176 E2000175 E3000175  D3010174 D9000171 E20C0170 E300016F *S...I...S...T...L...R...S...T..?-012260

* * *  D Y N B U F F R   C S E C T ,   S Y S O S 3 C C   P H A S E  * * *
C007A0-C7C2150C 240002C8 C7C217E0 24000108  C70219CC 24000198 C7021A8C 2400CCFC *G......HG.......G.......G.......-C1228C
0007C0-C6C213DC 25017200 C6C20670 25017200  C7021BA8 24C001A8 C601FE9C 250172DC *F.......F.......G.......F.......-0122A0
0007E0-C6C1936C 250172DC C6C186A0 250172DC  C70186AC 25000028 C7022810 21000288 *F..-....F.......G.......G.......-0122C0
C00800-C7C22AEC 25000070 C6C22AE0 250172DC  C6C186AC 25017200 C703C61C 21C096EC *G.......F.......F.......G.F.....-0122EC

* * *  L O W C O R E     C S E C T ,   S Y S O S 3 C C   P H A S E  * * *
C00820-58CC057C 07FD03C0 2D72B6E0 000EADOO  58DC057C C7FD03C0 2D72B6EC 000EADCC *...a...............a............-C12300

* * *  L A S T T C B S   C S E C T ,   S Y S O S 3 C C   P H A S E  * * *
C00840-000EADCC 00008D28 00008D26 00008D28  00008D29 C0008028 00008028 000EADCO *................................-012320

* * *  O L D P S W S     C S E C T ,   S Y S O S 3 C C   P H A S E  * * *
C00860-C3C40C00 40008B82 C3C4C001 40008B8C  C000C0CC CC012056 C3040000 60000C4E *C... ...C... ..........C...-..+-01234C
CCC880-C0C00C00 00002D60 C3C4C001 40000058  C03C0C00 CC012C56 C0000000 00012C56 *........C... ..................-012360

* * *  A L T T R A N S   C S E C T ,   S Y S O S 3 C C   P H A S E  * * *
CC08AC-000EA000 0C00000C 0C000000 0000000C  0000C0C0 C0000000 00000000 000EACCC *................................-012380

* * *  T R A N I D S     C S F C T ,   S Y S O S 3 C C   P H A S E  * * *
C008C0-25F627D9 25842218 269C26E0 20422212  213724CC 24AC276F 208A1019 11CB16CD *...R..................?........-0123A0
C008E0-16011605 16D71019 16CD1601 16D5C614  05F305E1 C5F30423 05F60704 061416C7 *.J.N.P.....J.N...3...T...6.....P-0123C0
C00900-000EAD00 0CDEADCO 000EAD00 000EAD00  000EADOC C00EAD00 000EAD00 000EADCC *................................-0123E0
```

Critical event history table
Timer value
Interrupt type (if event type=I; I/O interrupt)
Event type (I=interrupt)
Dynamic buffer management history table
Buffer size (for GETBUF)
Buffer type (shared code local storage)
Buffer address
"G"=GETBUF
TCB address (for FREEBUF)
Buffer type (shared code local storage)
Buffer address
"F"=FREEBUF
Current contents of low order main storage bytes 0-B (example shows no change)
Correct contents
Altered contents (if different)
TCB history table
Interrupt history table (interrupt IDs in PSW are circled)
Most recent interrupt
Alternate transient history table
Transient history table (example shows activity in transient areas #0(SOA), #1, and #2)
Transient area
Transient ID
Table entry

In addition to the history tables at the end of the supervisor, the supervisor debug option causes a shared code history table to be added to the end of all task control blocks. The EBCDIC names of the last eight shared code modules called on behalf of this task are listed, with the most recent eight-byte load module name always last (i.e., highest address).

7.4.2. Console Debug Options

A number of 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 an Ixx error code). This is useful because normal recovery from an Ixx 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
  ```

- Shared code debug option. Causes system halt (HPR code 990809) on any error detected by the system during execution of dynamic shared code. The console command is:

  ```
  SE DE,SC
  ```

- Dynamic buffer debug option. Causes system halt (HPR code 99130D) whenever the supervisor debug option detects the alteration of any of the last 32 bytes of any dynamic buffer, usually caused by dynamic buffer overflow. The console command is:

  ```
  SE DE,DB
  ```

  To turn off all debug options, the console command is:

  ```
  SE DE,OFF
  ```

7.4.3. Transient Management Halts

When trapping a system problem, it is often desirable to halt the processor whenever a particular transient or supervisor overlay is loaded into main storage. Every transient is uniquely identified with a transient ID. By using this ID (in hexadecimal), you can cause the system to halt in any of three ways:

1. Halt on transient load. The SE HA,TL console command causes an HPR 990C0C whenever transient management loads the specified 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 START. Note that this halt occurs only when a transient has just been loaded from SYSRES. A few transients can be reused; the halt will, therefore, occur only when the transient is initially loaded.

2. Halt on transient call. The SE HA,TC console command causes an HPR 990C0D whenever a transient or overlay calls the specified transient as an overlay. For example, if transient 200 processes errors by overlaying transient 204, the SE HA,TC,204 command causes a halt before transient 204 is loaded on top of transient 200.

3. Halt on transient exit. The SE HA,TE console command causes an HPR 990C0E whenever a specified transient or overlay exits, either by releasing the transient area or by calling an overlay.

All three halts described can be set simultaneously, if desired, but only the last SET command of each type is recognized. When the halts occur, problem register 15 can be used to find the address of the transient area involved. Refer to the operations handbook for instructions on reading problem registers.

The halt on transient load is available on all supervisors. Halt on transient call and exit require use of the supervisor debug option at IPL.

7.4.4. Symbiont Halt

The SE HA,SY console command causes an HPR 997C whenever a particular symbiont or phase of a symbiont is loaded. This could be useful when debugging a particular symbiont.

To halt whenever a specific symbiont is loaded, simply key in SE HA,SY,id where id is the 2-character symbiont id (e.g., RU,FI,PR,SU). To halt when a phase other than the root phase is loaded, key in SE HA,SY,idnn where nn is the decimal EBCDIC phase number (00–99).

The HPR occurs less than 10 instructions prior to the symbiont phase being given control. To continue normally, press START.

7.4.5. Shared Code Halts and Pauses

SET console commands are available to interrupt or halt processing when shared code modules are called or when they return. These commands allow the operator to request an interrupt or halt on the call or return for:

- a specific module;
- a specific group of modules, which have a common prefix; or
- all modules.

The format of these commands is:

$$\mathtt{SE}\left\{\begin{matrix}\mathtt{HA}\\\mathtt{PA}\end{matrix}\right\}\mathtt{,}\left\{\begin{matrix}\mathtt{SC}\\\mathtt{SR}\\\mathtt{SE}\end{matrix}\right\}\left[\mathtt{,}\left\{\begin{matrix}\mathtt{prefix.}\\\mathtt{name}\end{matrix}\right\}\right]$$

The first and second parameters form individual commands, which are discussed in the following paragraphs. The third parameter determines what modules these commands affect. You specify an individual module by its full name, a module group by its prefix followed immediately by a period, or all modules by omitting the parameter completely. For example, the command SE HA,SC,DM. causes an HPR upon a call to any module whose name begins with DM.

You can continue past any HPR resulting from these commands by pressing RUN. The supervisor debug option is required at IPL time for all of these functions. The individual commands are:

- Halt on shared code call. The SE HA,SC command causes an HPR of 991D01 when a module is called.

- Halt on shared code return. The SE HA,SR command causes an HPR of 991D02 when control returns from a module.

- Halt on shared code return with error. The SE HA,SE command causes an HPR of 991D03 when control returns from a module with an error condition.

- Pause on shared code call. The SE PA,SC command interrupts processing and displays the following message when a module is called:

  ```
  SE25 SC PAUSE ON shared-code-name. CONTINUE? (Y, HELP)
  ```

  This message shows which shared code module has been called. A reply of Y causes processing to resume. A reply of HELP displays the following information: the job or symbiont name, the name of the calling module, the TCB address, the base address of the calling module, and the local store address.

- Pause on shared code return. The SE PA,SR command interrupts processing and displays the SE25 message when control returns from a module by execution of the SRETURN macroinstruction. If requested to, this command displays the same shared code information as SE PA,SC does, except that it shows what module is being returned to rather than what module called the shared code.

- Pause on shared code return with error. The SE PA,SE command interrupts processing and displays the SE25 message when control returns from a module in which an error has occurred. If requested to, this command displays the same shared code information as SE PA,SR does.

7.4.6. Soft-Patch Symbiont (PT)

The PT symbiont is used to temporarily patch transients (transient overlays), load modules, and shared code modules at the time they are loaded in main storage (soft patch), instead of permanently patching the disk (hard patch). This is useful if you want to test a patch to see if it is effective before hard-patching or if you want to trap a problem by temporarily applying a patch. To use the PT symbiont, you must have included the supervisor debug option at IPL time.

When initiated, the PT symbiont builds a patch table from input keyed in from the workstation console or read from cards. The PT symbiont then locks itself into the supervisor so it can scan this table on every load of a transient, load module, or shared code module. During the scan, if the module name matches an entry on the patch table, the specified patches are applied. These patches are temporary. Patches to transients remain in effect until the PT symbiont is cancelled. Load and shared code modules that are loaded in main storage while the PT symbiont is active remain patched until reloaded.

The PT symbiont is also used to apply patches to the resident supervisor; however, these patches remain in effect until you IPL the system again.

7.4.6.1. Soft-Patching Using the Workstation Console

To apply a soft patch from the workstation console, you must initiate the PT symbiont by keying in a console command:

```
PT C
```

Once initiated, the PT symbiont solicits input from the workstation console. The entries you make identify the modules to be patched and the patches to be applied. The input is entered in card-image format. Four types of input entries are solicited by the PT symbiont:

1. The first entry is for compatibility purposes. It is necessary when using the transient patch (TRNPAT) program, which applies core to transients.

   Format:

   ```
   1 D=R
   ```

   A 1 must be keyed in first, followed by a blank, and then D=R.

2. The second entry defines the type and the id (or name) of the module to be patched. The form of this entry depends upon the module type.

   Formats:

`2 T=decimal-id`	(for transients)
`2 S=module-name`	(for shared code modules)
`2 L=module-name`	(for load modules – the load modules can be the resident supervisor, a symbiont, or a module loaded from a user library)
`2 O=module-name`	(for resident supervisor modules specifying the csect or object module name)

   In all formats, a 2 must be keyed in first, followed by a blank. Each module to be patched must be defined with one of these entries.

3. The third entry defines the patch. Only one patch can be entered at a time, and the patch is applied only to the module specified in the preceding 2 entry.

   Format:

   ```
   P addr,patch
   ```

   A P is entered first, followed by a blank. Then, the hexadecimal address (relative to the start of the transient or module phase to be patched) is entered. The address must be within the module specified or the entry will be ignored. The adress is followed by a comma and then the patch. (The patch is also given in hexadecimal, and embedded blanks are not permitted.) The patch character string can be any length, though the entire P entry must fit on one line of the console (or on one card, if using card input).

   More than one patch can be made to a module by keying in more than one P entry. All patches to be applied to a given module should be specified in succesive P entries, following the 2 entry that defines the module.

4. The last entry signifies the end of the patches.

   Format:

   ```
   /*
   ```

Examples:

PT C	Initiates the PT symbiont
1 D=R	Can be eliminated if not using TRNPAT
2 L=MYSAL	Defines the load module MYSAL
P 1A,47ØØØØØØ	Defines a patch to be applied to MYSAL
2 T=1539	Defines a transient
P 94,CØ	Defines two patches to be applied to the transient
P 12A,478ØF2E49966	
2 S=MYSHRCOD	Defines a shared code module MYSHRCOD
P 24,Ø7CØ	Defines a patch to be applied to MYSHRCOD
2 O=SM$DEBUG	Defines an object module SM$DEBUG
P 27,FF	Defines a patch to be applied to SM$DEBUG
/*	Indicates the end of the patches

There are some optional features available to you when soft-patching directly from the console. For example, you can enter all the information for a single patch on one line from the workstation console. The following is the format of this option:

```
PT ⎧T,transient-id             ⎫,addr,patch
   ⎪S,shared-code-module-name  ⎪
   ⎨L,load-module-name         ⎬
   ⎩O,object-module-name       ⎭
```

Although this form eliminates the need of separate entries for 2 and P type card-image inputs, it can only be used to make a patch at one location (module address). To patch more than one location, use one of the other forms of the command, or key in this form one time for each location to be patched.

Examples:

```
PT L,MYSAL,1A,47000000
PT T,440,F0,45A0F220
PT S,MYSHRCOD,24,07C0
PT O,SM$DEBUG,27,FF
```

If your system has card input/output capabilities, you can key in the following console command to initiate the PT symbiont:

```
PT[dev-addr] C
```

This form of the command not only solicits patch input from the workstation console, but also punches that input on the device specified. The card deck produced contains the patches that you can reuse at some later time.

7.4.6.2. Soft-Patching Using Card Input

If your system has card input capabilities, you can use a card deck as input for a soft patch. The input deck is comprised of cards containing the same four types of input entries described in 7.4.6.1. The sequence and formats for the cards are the same as previously described for the input entries made via the workstation console. After you produce the card deck, it must be placed in the system reader prior to initiating the PT symbiont. Once the card deck is in place, the PT symbiont can be initiated via the following console command:

```
PT
```

This form of the command causes the PT symbiont to accept the patches on the card deck from the system reader device.

7.4.6.3. Using the PT Command

Whether you use console input or card input, you can enter the PT symbiont command more than once, and the input is simply added to the end of the patch table. In addition, any combination of the various forms can be used. For example, you can key in PT and a patch table will be built from card input. Later in the same session, you can key in PT C and enter additional patches. These additional patches will be added to the existing patch table.

7.4.6.4. Cancelling the PT Symbiont

Regardless of how the input is entered, the PT symbiont can be cancelled at any time by keying in the following console command:

```
CA PT,S,N
```

Cancelling the PT symbiont eliminates all the patches entered, except those that changed the resident supervisor. (These will remain in effect until you IPL the system again.) Shared code and load modules that were loaded while the PT symbiont was active will remain patched until reloaded. Subsequent loads of modules, however, will not be patched.

7.4.6.5. PT Symbiont Error Messages

Error messages are produced by the PT symbiont and appear on the workstation console screen. The following is a list of the error messages that might occur, the condition that caused the error, and the corrective action to take:

PT01 TWO 2 IMAGES IN A ROW

Two 2 entries have been made in a row. This is invalid because a module has been specified to be patched, but no patches have been entered. The first 2 entry is ignored, and the PT symbiont continues. This could result in incorrectly applied patches. To avoid this, cancel the PT symbiont, correct the input deck, and begin a new PT symbiont session.

PT02 INVALID CHARACTER STRING, CHARACTER ON CARD

A nonhexadecimal digit (other than 0–9 and A–F) was entered in a field requiring a hexadecimal digit. This message is also produced if an odd number of characters was entered for a patch (patches cannot be half bytes in length). Cancel the PT symbiont, correct the input deck, and begin a new PT symbiont session.

PT03 PATCH TABLE OVERFLOW - SOME PATCHES LOST

Too many patches have been entered. There is a limited amount of space that can be allotted to the patch table, and the PT symbiont will stop accepting input when this limit is exceeded. This could result in a patch table that contains only part of the patches you intended to apply. To avoid this, cancel the PT symbiont, limit the number of soft patches you enter, and begin a new PT symbiont session.

PT04 INVALID PT - NEEDS SUPV DEBUG OPTION SET AT IPL

The supervisor debug option, which is required if the PT symbiont is to be used, was not specified at IPL time. The PT command is ignored, and the symbiont cannot be initiated. You must IPL the supervisor again, specifying the debug option; then begin a new PT symbiont session.

PT05 PUNCH SPECIFIED BUT NO CONSOLE INPUT

The form of the PT command specifying a punch device was used, but the input was not specified as coming from the workstation console. This will occur if the C following the device address was not entered. The PT command is ignored under this condition. Reenter the command, including the final C.

PT06 csect-name NOT FOUND

The object module or csect name specified on the 2 O= entry was not found on the supervisor currently loaded. The 2 O= entry and all the P entries until the next 2 entry are ignored. If an incorrect object module or csect name was entered, you can enter the correct name later in the session, and the input will be added to the patch table.

PT07 SYSRDR NOT AVAILABLE

The form of the PT command used requires the system reader device, but in this case it is unavailable. The PT command is ignored. When the system reader becomes available, reenter the command.

PT08 INVALID INPUT FORMAT

An error was made in entering the information for a patch on a single line from the workstation console. The PT command is ignored under this condition. Check to make sure that all commas are in the right place, and reenter the command.

Glossary

A

alias phase name

A 1- to 6-character phase name supplied by the user to the linkage editor, which can be used in place of the linkage editor-generated name for the phase.

alternate load library

A library other than the *run library* or *load library* from which you may call a load module and which you must specify on the EXEC job control statement for that program.

alternate transient file

A duplicate of the *transient* file, from which the supervisor calls transients if it cannot get them from the transient file.

B

block number processing

A technique to ensure correct tape positioning whereby a block sequence number is written on output tape files and checked on input tape files. This block count is recorded on tape in the first three bytes of the block, and consists of a 4-bit tape mark count and a 20-bit block number count.

breakpoint

A point in a spool subfile where the subfile is closed, then reopened so that the contents of the subfile can be output to the physical device before the job step terminates.

C

CDIB (common data interface block)

A control block that exists for each MIRAM file on disk as well as for files on all other OS/3 devices. It contains the file name and serves as the primary interface of the file to consolidated data management.

checkpoint

A point in a program or routine where a recording of the contents of main storage and the state of peripheral units is made so that, in the event of machine failure or some other interruption, a job can be restarted at an intermediate point rather than from the beginning.

communication region

A 12-byte field in the job preamble used to pass information from one job step to the next.

control stream

A sequence of control statements that define one or more jobs to the operating system and may include source code or data as required by the jobs.

current ID

A field in the PCA table that contains the starting address of the logical partition or the address of the current record being processed.

D

DTF (define the file) table

A control block that contains the file name and operating and physical characteristics of a file used by the SAT routines.

dump

To copy the contents of all or part of main storage. Also, the data resulting from the process.

E

ECB (event control block)

A control block that identifies a subtask and indicates status to the other tasks within a job step. An ECB is created for each subtask.

embedded data set

Data in the form of card images entered into the system with the job control stream.

end of data ID

A field in the PCA table that contains the address of the last logical record of the partition.

F

FCB (file control block)
File and device information compiled by job control and stored by the supervisor in the PIOCB.

file identifier
The physical label of a file. It is specified in the LBL job control statement and, for a disk file, may have up to 44 bytes. It is used to locate the VTOC entry that contains control information for this disk file.

filelock
A lock to prevent unauthorized access to a file. This lock is set at the logical level. Only macroinstructions specifying a required password prefixed to the file name can access the file.

file name
The name of the logical file. It is specified in the LFD job control statement and may have up to eight alphanumeric characters. It is the logical file name used to access a file and to locate the FCB block for a file.

H

HPR (HALT AND PROCEED)
A privileged machine instruction that halts the processor when an error occurs. Depending on the HPR, you may or may not be able to continue. If you can resume processing, correct the problem, then press the START key.

I

interlace
A technique whereby blocks on a partitioned file are spaced on the track so that more than one I/O operation may be performed per disk revolution. The interlace factor is specified by the LACE keyword parameter of the PCA macroinstruction.

interrupt
An external event which the supervisor must handle in some way to permit processing to continue.

IPL (initial program load)
The procedure by which the operator loads into main storage and initializes the resident portion of the supervisor.

island code

Closed routines by which you may handle interrupts, contingencies, or events not normally processed by the supervisor. These island code routines are activated when your program is interrupted for:

- Abnormal termination

  An error occurs, making continuation of the program impossible.

- Interval timer

  The time interval previously specified by a SETIME macroinstruction elapses.

- Operator communications

  An unsolicited message is entered at the system console by the operator.

- Program check

  An operation in the problem program causes a hardware program check interrupt, such as an addressing violation, an arithmetic overflow, or an operation exception.

J

job

A total processing application comprising one or more processing steps. Each job is divided into job steps (programs) that are executed serially.

job region

An area in main storage reserved for each job containing the control information (prologue) and the program code for the job step being executed.

job step

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

L

library search order

The default order of search employed by the loader is:

1. Load library file ($Y$LOD)

2. Job run library file ($Y$RUN)

If the job run library file ($Y$RUN) is specified on the EXEC job control card, the order of search is:

1. 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. 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 it is generally more efficient to link modules together than to create a series of smaller, separately linked load modules.

load library
The system file containing all system-supplied load modules used in normal processing; its file identifier is $Y$LOD.

load module
A single-phase, multiphase, or multiregion program produced by the linkage editor and loaded into main storage for execution.

loader
The supervisor routine that brings a program, in *load module* form, from disk to main storage and optionally gives control to the newly-loaded program.

M

machine check
An interrupt that notifies the operating system of hardware failure.

main storage consolidation
The repositioning of jobs in main storage in order to run a job requiring more contiguous space than is currently available.

MIRAM (multiple indexed random access method)
A data management access method that can read and write records sequentially, randomly by relative record number, or randomly by key (up to five separate keys).

module
A program element existing in either source, proc, object, or loadable program format and serving as input or output to various system functions.

monitor

A supervisor routine that interrupts each instruction in a program, or a part of a program, so that a trace of program execution can be made.

multijobbing

The concurrent scheduling, loading, and execution of more than one job at a time. Up to seven jobs can be processed concurrently, with each job consisting of one or more job steps.

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.

P

PCA (partition control appendage)

A control block that contains the characteristics of a partition within a file used by the SAT routines.

PIOCS (physical input/output control system)

A set of resident routines that controls the activity between the processor and all peripheral devices connected to the multiplexer, selector, and integrated channels.

preamble

The fixed portion of the job region prologue at the beginning of the prologue.

primary task

A task that represents a job step and is capable of creating subtasks of equal or lower priority.

priority

One of several levels within the supervisor *switch list*, each of which may be assigned one or more tasks.

processor

A unit of the computer that includes the circuits controlling the interpretation and execution of instructions. Synonymous with central processing unit.

program exception

An interrupt generated by hardware when it detects the improper specification or use of machine instructions or data.

program phase (load module phase)

A program segment that can perform one or more specific processing operations. A program phase is output by the linkage editor and stored in a load library, then located and read into main storage by the program loader routine.

prologue

The portion of the job region containing the preamble, TCBs, and disk storage extent control information for a job.

PSW (program status word)

A hardware structure that contains an instruction address and control information for the program currently being executed. When an interrupt occurs, the supervisor stores the PSW in a PSW save area, suspends the task, processes the interrupt, and then uses the stored PSW to resume the interrupted task.

PUB (physical unit block)

A control block in main storage for each I/O device containing the physical address, characteristics, status, and other control information for the device.

R

relocation register

A hardware register, one of which exists for each job in main storage. Whenever a program within a job addresses main storage, the address, which is job-relative, is added to the job base address contained in its corresponding relocation register to get the absolute address the hardware needs. The process is invisible to user programs.

repressible machine check

An interrupt generated by a hardware malfunction from which the supervisor may recover, but which may indicate a potentially serious hardware problem.

restart

To resume processing a job from some intermediate point (called a checkpoint) following an interruption.

rollout/rollin

The temporary transfer of jobs from main storage to disk to make room for a job with a preemptive scheduling priority (rollout) and the transfer of those jobs back into main storage when the preemptive job has finished (rollin).

run library

The file that most system programs use by default for input/output storage; its file identifier is $Y$RUN.

S

SAT (system access technique)

An input/output control system that provides a standard interface for tape and disk subsystems between OS/3 data management and the PIOCS.

shared code

Executable code that does not write to itself and, therefore, can be used simultaneously by more than one program.

SIB (system information block)
An area in main storage, which is part of the resident supervisor, containing system control information.

spooling (simultaneous peripheral operation on line)
A technique that increases the throughput of a system by 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.

subtask
A task that is created by a user ATTACH macroinstruction and executes concurrently with the parent task.

supervisor
The set of programs that forms the central control of OS/3 and coordinates tasks within OS/3 with each other and with external events such as interrupts.

SVC (SUPERVISOR CALL)
A machine instruction that acts as a user program's only interface to the supervisor.

switch list
A supervisor table divided into priorities; each priority may specify one or more tasks. The *switcher* uses the switch list to coordinate all the tasks currently in a system.

switcher
A supervisor routine that maintains the *switch list*, selecting one task among all those on the list to get processor control.

symbiont
A system utility routine that operates concurrently with other system programs and with user programs and is executed in the same manner as a job step.

T

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 (the primary task) and may have additional tasks (subtasks), all of which compete independently for processor time. There may be a maximum of 256 tasks per job.

TCA (tape control appendage)
A control block that contains the characteristics of a magnetic tape file used by the SAT routines.

TCB (task control block)

A control table generated for each task and stored in the job region prologue. A TCB is constructed for each job step submitted by job control for execution. Additional TCBs are constructed for each subtask created by the ATTACH supervisor macroinstruction.

trace

A diagnostic technique in the monitor routine that prints program information (PSW and register contents, etc) at specified points in a program executing in the monitor mode so that errors can be located and corrected.

transient

A supervisor routine that resides in the transient file on disk and is called into main storage for execution only when needed; after execution is finished, the transient may be executed again or overlaid by other transients.

V

VTOC (volume table of contents)

A directory on a disk volume that defines the files contained in that volume.

Index

SPERRY UNIVAC

USER COMMENT SHEET

Your comments concerning this document will be welcomed by Sperry Univac for use in improving subsequent editions.

Please note: This form is not intended to be used as an order blank.

____

(Document Title)

________________ ________________ ________________

(Document No.) *(Revision No.)* *(Update No.)*

Comments:

From:

____

(Name of User)

____

(Business Address)

Fold on dotted lines, and mail. (No postage stamp is necessary if mailed in the U.S.A.)
Thank you for your cooperation

Cut along line.

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 21 BLUE BELL, PA.

POSTAGE WILL BE PAID BY ADDRESSEE

SPERRY UNIVAC

ATTN.: SYSTEMS PUBLICATIONS

P.O. BOX 500
BLUE BELL, PENNSYLVANIA 19424

FOLD

CUT

SPERRY UNIVAC

USER COMMENT SHEET

Your comments concerning this document will be welcomed by Sperry Univac for use in improving subsequent editions.

Please note: This form is not intended to be used as an order blank.

(Document Title)

(Document No.) *(Revision No.)* *(Update No.)*

Comments:

From:

(Name of User)

(Business Address)

Fold on dotted lines, and mail. (No postage stamp is necessary if mailed in the U.S.A.)
Thank you for your cooperation

Cut along line.

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 21 BLUE BELL, PA.

POSTAGE WILL BE PAID BY ADDRESSEE

SPERRY UNIVAC

ATTN.: SYSTEMS PUBLICATIONS

P.O. BOX 500
BLUE BELL, PENNSYLVANIA 19424

CUT

FOLD

SPERRY UNIVAC

USER COMMENT SHEET

Your comments concerning this document will be welcomed by Sperry Univac for use in improving subsequent editions.

Please note: This form is not intended to be used as an order blank.

(Document Title)

(Document No.) *(Revision No.)* *(Update No.)*

Comments:

From:

(Name of User)

(Business Address)

Fold on dotted lines, and mail. (No postage stamp is necessary if mailed in the U.S.A.)
Thank you for your cooperation

long line.

FOLD

CUT

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 21 BLUE BELL, PA.

POSTAGE WILL BE PAID BY ADDRESSEE

SPERRY UNIVAC

ATTN.: SYSTEMS PUBLICATIONS

P.O. BOX 500
BLUE BELL, PENNSYLVANIA 19424

FOLD