Basic Data Management

OS/3

User Guide

Environment: 90/25, 30, 30B, 40 Systems

H

SPERRY UNIVAC

UP-8068 Rev. 4

SPERRY UNIVAC

PUBLICATIONS UPDATE

Operating System/3 (OS/3)

Consolidated Data Management

User Guide

UP-8068 Rev. 4–D

This Library Memo announces the release and availability of Updating Package D to "SPERRY Operating System/3 (OS/3) Basic Data Management User Guide", UP-8068 Rev. 4.

This 8.1 release update documents a correction applicable to a feature present in basic data management prior to the 8.1 release.

Copies of Updating Package D 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 representative. To receive only the updating package, order UP-8068 Rev. 4–D. To receive the complete manual, order UP-8068 Rev. 4.

LIBRARY MEMO ONLY	LIBRARY MEMO AND ATTACHMENTS	THIS SHEET IS
Mailing Lists BZ, CZ and MZ	Mailing Lists A00, A01, 18, 18U, 19, 19U, 20, 20U 21, 21U, 75, 75U, 76 and 76U (Package D to UP-8068 Rev. 4, 7 pages plus Memo)	Library Memo for UP-8068 Rev. 4–D
		RELEASE DATE: June, 1983

SPERRY UNIVAC

PUBLICATIONS UPDATE

Operating System/3 (OS/3)

Basic Data Management

User Guide

UP–8068 Rev. 4–C

This Library Memo announces the release and availability of Updating Package C to "SPERRY UNIVAC Operating System/3 (OS/3) Basic Data Management User Guide", UP–8068 Rev. 4.

This update documents the following new information on the basic data management file lock feature for the 8.0 release:

- How to avoid unnecessary locking out of files
- Additional information on file shareability

All other changes are corrections or expanded descriptions applicable to features present in basic data management prior to the 8.0 release.

Copies of Updating Package C 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–8068 Rev. 4–C. To receive the complete manual, order UP–8068 Rev. 4.

LIBRARY MEMO ONLY	LIBRARY MEMO AND ATTACHMENTS	THIS SHEET IS:
Mailing Lists BZ, CZ and MZ	Mailing Lists A00,A01,18,18U,19, 19U,20,20U,21,21U,75,75U,76 and 76U (Package C to UP–8068 Rev. 4, 32 pages plus Memo)	Library Memo for UP–8068 Rev. 4–C RELEASE DATE: February, 1983

SPERRY UNIVAC
COMPUTER SYSTEMS

PUBLICATIONS UPDATE

Operating System/3 (OS/3)

Basic Data Management

User Guide

UP-8068 Rev. 4–B

This Library Memo announces the release and availability of Updating Package B to "SPERRY UNIVAC Operating System/3 (OS/3) Basic Data Management User Guide", UP-8068 Rev. 4.

This update for the 8.0 release indicates the availability of a new conversion routine for basic data management. This routine is the OS/3 Sequential DTF Mode to CDI Mode Converter (DTFCDI301). This converter processes a basic data management BAL source program module and produces a consolidated data management source module that, with minimal modification, can be used in the consolidated data management environment.

All other changes are corrections or expanded descriptions applicable to features present in basic data management prior ot the 8.0 release.

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 only the updating package, order UP-8068 Rev. 4–B. To receive the complete manual, order UP-8068 Rev. 4.

LIBRARY MEMO ONLY	LIBRARY MEMO AND ATTACHMENTS	THIS SHEET IS:
Mailing Lists BZ, CZ and MZ	Mailing Lists A00, A01, 18, 18U, 19, 19U, 20, 20U, 21, 21U, 75, 75U, 76, and 76U (Package B to UP-8068 Rev. 4, 29 pages plus Memo)	Library Memo for UP-8068 Rev. 4–B RELEASE DATE: September, 1982

SPERRY UNIVAC
COMPUTER SYSTEMS

PUBLICATIONS UPDATE

Operating System/3 (OS/3)

Basic Data Management

User Guide

UP-8068 Rev. 4-A

This Library Memo announces the release and availability of Updating Package A to "SPERRY UNIVAC Operating System/3 (OS/3) Basic Data Management User Guide", UP-8068 Rev. 4.

This update documents the following new basic data management features for the 7.0 release:

- Consolidated Data Management migration considerations
- New information on the file lock feature

All other changes are corrections or expanded descriptions applicable to features present in basic data management prior to the 7.0 release.

Copies of the 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-8068 Rev. 4-A. To receive the complete manual, order UP-8068 Rev. 4.

LIBRARY MEMO ONLY	LIBRARY MEMO AND ATTACHMENTS	THIS SHEET IS
Mailing Lists BZ, CZ and MZ	Mailing Lists 18, 18U, 19, 19U, 20, 20U, 21, 21U, 75, 75U, 76 and 76U (Package A to UP-8068 Rev. 4, 38 pages plus Memo)	Library Memo for UP-8068 Rev. 4-A RELEASE DATE: December, 1981

PAGE STATUS SUMMARY

ISSUE: Update D – UP-8068 Rev. 4
RELEASE LEVEL: 8.1 Forward

Part/Section	Page Number	Update Level
Cover/Disclaimer		Orig.
PSS	1	D
Preface	1 2 3, 4	Orig. A Orig.
Contents	1 thru 11 12 13, 14 15 16, 16a 17, 18 19	Orig. C Orig. B A Orig. C
PART 1	Title Page	Orig.
1	1 2 2a 3 4 thru 18	Orig. A A A Orig.
PART 2	Title Page	Orig.
2	1 thru 4	Orig.
3	1 thru 31	Orig.
4	1 thru 5	Orig.
5	1 thru 12	Orig.
6	1 thru 12	Orig.
7	1 thru 31	Orig.
PART 3	Title Page	Orig.
8	1 thru 17	Orig.
9	1 thru 62	Orig.
PART 4	Title Page	Orig.
10	1 thru 7 8 9, 10 11, 12 12a 13 14, 15	Orig. A Orig. A A Orig. A
10 (cont)	16 thru 22	Orig.
11	1 2 3 thru 7 8 9 10 thru 15 16 17 thru 51	Orig. A Orig. A C Orig. A Orig.
12	1 thru 9 10 11 thru 13	Orig. B Orig.
13	1 thru 18 18a 19 thru 29	Orig. C Orig.
13A	1 2 3 4 4a 5 thru 13	Orig. B Orig. B B Orig.
13B	1 2 3 thru 5 6 7 thru 12 13 14 15 thru 17 18 19 thru 21	Orig. B Orig. D Orig. C D Orig. B Orig.
14	1 thru 13	Orig.
15	1 thru 7 8, 9 10 11, 12 13 14 15, 16 17 18 thru 20 21 22 thru 111	Orig. C Orig. C Orig. C Orig. C Orig. C Orig.
16	1 2 3 4 4a 5	B Orig. A B C C
16 (cont)	6 thru 14	Orig.
PART 5	Title Page	Orig.
17	1 thru 75	Orig.
PART 6	Title Page	Orig.
Appendix A	1 thru 11	Orig.
Appendix B	1 thru 15	Orig.
Appendix C	1 thru 11	Orig.
Appendix D	1 thru 32	Orig.
Appendix E	1 thru 26	Orig.
Appendix F	1 2, 3	A B
Index	1, 2 3 4 thru 6 7 8 thru 10 11 thru 23 24 25 thru 27	Orig. A B Orig. C Orig. C Orig.
User Comment Sheet		

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

PAGE STATUS SUMMARY

ISSUE: Update C – UP-8068 Rev. 4
RELEASE LEVEL: 8.0 Forward

Part/Section	Page Number	Update Level
Cover/Disclaimer		Orig.
PSS	1	C
Preface	1 2 3, 4	Orig. A Orig.
Contents	1 thru 11 12 13, 14 15 16, 16a 17, 18 19	Orig. C Orig. B A Orig. C
PART 1	Title Page	Orig.
1	1 2 2a 3 4 thru 18	Orig. A A A Orig.
PART 2	Title Page	Orig.
2	1 thru 4	Orig.
3	1 thru 31	Orig.
4	1 thru 5	Orig.
5	1 thru 12	Orig.
6	1 thru 12	Orig.
7	1 thru 31	Orig.
PART 3	Title Page	Orig.
8	1 thru 17	Orig.
9	1 thru 62	Orig.
PART 4	Title Page	Orig.
10	1 thru 7 8 9, 10 11, 12 12a 13	Orig. A Orig. A A Orig.
10 (cont)	14, 15 16 thru 22	A Orig.
11	1 2 3 thru 7 8 9 10 thru 15 16 17 thru 51	Orig. A Orig. A C Orig. A Orig.
12	1 thru 9 10 11 thru 13	Orig. B Orig.
13	1 thru 18 18a 19 thru 29	Orig. C* Orig.
13A	1 2 3 4 4a 5 thru 13	Orig. B Orig. B B Orig.
13B	1 2 3 thru 5 6 7 thru 12 13, 14 15 thru 17 18 19 thru 21	Orig. B Orig. B Orig. C Orig. B Orig.
14	1 thru 13	Orig.
15	1 thru 7 8, 9 10 11, 12 13 14 15, 16 17 18 thru 20 21 22 thru 111	Orig. C Orig. C Orig. C Orig. C Orig. C Orig.
16	1 2 3 4 4a	B Orig. A B C*
16 (cont)	5 6 thru 14	C Orig.
PART 5	Title Page	Orig.
17	1 thru 75	Orig.
PART 6	Title Page	Orig.
Appendix A	1 thru 11	Orig.
Appendix B	1 thru 15	Orig.
Appendix C	1 thru 11	Orig.
Appendix D	1 thru 32	Orig.
Appendix E	1 thru 26	Orig.
Appendix F	1 2, 3	A B
Index	1, 2 3 4 thru 6 7 8 thru 10 11 thru 23 24 25 thru 27	Orig. A B Orig. C Orig. C Orig.
User Comment Sheet		
	3	B

*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-8068 Rev. 4
RELEASE LEVEL: 8.0 Forward

Part/Section	Page Number	Update Level
Cover/Disclaimer		Orig.
PSS	1	B
Preface	1 2 3, 4	Orig. A Orig.
Contents	1 thru 11 12 13, 14 15 16, 16a 17 thru 19	Orig. A Orig. B A Orig.
PART 1	Title Page	Orig.
1	1 2 2a 3 4 thru 18	Orig. A A A Orig.
PART 2	Title Page	Orig.
2	1 thru 4	Orig.
3	1 thru 31	Orig.
4	1 thru 5	Orig.
5	1 thru 12	Orig.
6	1 thru 12	Orig.
7	1 thru 31	Orig.
PART 3	Title Page	Orig.
8	1 thru 17	Orig.
9	1 thru 62	Orig.
PART 4	Title Page	Orig.
10	1 thru 7 8 9, 10 11, 12 12a 13 14, 15 16 thru 22	Orig. A Orig. A A Orig. A Orig.
11	1 2 3 thru 7 8 9 thru 15 16 17 thru 51	Orig. A Orig. A Orig. A Orig.
12	1 thru 9 10 11 thru 13	Orig. B Orig.
13	1 thru 29	Orig.
13A	1 2 3 4 4a 5 thru 13	Orig. B Orig. B B* Orig.
13B	1 2 3 thru 5 6 7 thru 17 18 19 thru 21	Orig. B Orig. B Orig. B Orig.
14	1 thru 13	Orig.
15	1 thru 111	Orig.
16	1 2 3 4 5 6 thru 14	B Orig. A B A Orig.
PART 5	Title Page	Orig.
17	1 thru 75	Orig.
PART 6	Title Page	Orig.
Appendix A	1 thru 11	Orig.
Appendix B	1 thru 15	Orig.
Appendix C	1 thru 11	Orig.
Appendix D	1 thru 32	Orig.
Appendix E	1 thru 26	Orig.
Appendix F	1 2 3	A B B*
Index	1, 2 3 4 thru 6 7 thru 27	Orig. A B 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.

PAGE STATUS SUMMARY

ISSUE: Update A – UP-8068 Rev. 4
RELEASE LEVEL: 7.0 Forward

Part/Section	Page Number	Update Level
Cover/Disclaimer		Orig.
PSS	1	A
Preface	1 2 3, 4	Orig. A Orig.
Contents	1 thru 11 12 13 15, 16 16a 17 thru 19	Orig. A Orig. A A* Orig.
PART 1	Title Page	Orig.
1	1 2 2a 3 4 thru 18	Orig. A A* A Orig.
PART 2	Title Page	Orig.
2	1 thru 4	Orig.
3	1 thru 31	Orig.
4	1 thru 5	Orig.
5	1 thru 12	Orig.
6	1 thru 12	Orig.
7	1 thru 31	Orig.
PART 3	Title Page	Orig.
8	1 thru 17	Orig.
9	1 thru 62	Orig.
PART 4	Title Page	Orig.
10	1 thru 7 8 9, 10 11, 12 12a 13 14, 15 16 thru 22	Orig. A Orig. A A* Orig. A Orig.
11	1 2 3 thru 7 8 9 thru 15 16 17 thru 51	Orig. A Orig. A Orig. A Orig.
12	1 thru 13	Orig.
13	1 thru 29	Orig.
13A	1 thru 13	Orig.
13B	1 thru 21	Orig.
14	1 thru 13	Orig.
15	1 thru 111	Orig.
16	1, 2 3 thru 5 6 thru 14	Orig. A Orig.
PART 5	Title Page	Orig.
17	1 thru 75	Orig.
PART 6	Title Page	Orig.
Appendix A	1 thru 11	Orig.
Appendix B	1 thru 15	Orig.
Appendix C	1 thru 11	Orig.
Appendix D	1 thru 32	Orig.
Appendix E	1 thru 26	Orig.
Appendix F	1, 2	A
Index	1, 2 3 4 thru 27	Orig. A 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: UP-8068 Rev. 4
RELEASE LEVEL: 7.0 Forward

Part/Section	Page Number	Update Level
Cover/Disclaimer		
PSS	1	
Preface	1 thru 4	
Contents	1 thru 19	
PART 1	Title Page	
1	1 thru 18	
PART 2	Title Page	
2	1 thru 4	
3	1 thru 31	
4	1 thru 5	
5	1 thru 12	
6	1 thru 12	
7	1 thru 31	
PART 3	Title Page	
8	1 thru 17	
9	1 thru 62	
PART 4	Title Page	
10	1 thru 22	
11	1 thru 51	
12	1 thru 13	
13	1 thru 29	
13A	1 thru 13	
13B	1 thru 21	
14	1 thru 13	
15	1 thru 111	
16	1 thru 14	

Part/Section	Page Number	Update Level
PART 5	Title Page	
17	1 thru 75	
PART 6	Title Page	
Appendix A	1 thru 11	
Appendix B	1 thru 15	
Appendix C	1 thru 11	
Appendix D	1 thru 32	
Appendix E	1 thru 26	
Index	1 thru 27	
User Comment Sheet		

Part/Section	Page Number	Update Level

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

Preface

This manual is one of a series designed to instruct and guide the programmer in the use of the SPERRY UNIVAC Operating System/3 (OS/3). This manual specifically describes OS/3 basic data management and its effective use. Its intended audience is the applications programmer with a basic knowledge of data processing, but with limited programming experience, as well as the seasoned applications programmer.

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

For systems with interactive facilities, an additional series of manuals is provided to instruct and guide the programmer in the use of OS/3 consolidated data management. These are:

- Introduction to consolidated data management, UP-8824
- Consolidated data management concepts and facilities, UP-8825
- Consolidated data management macro language user guide/programmer reference, UP-8826

In general, any further references to the term *data management* in this user guide imply basic data management.

This user guide is divided into the following parts:

- PART 1. OS/3 DATA MANAGEMENT

  Introduces OS/3 data management in terms of what it is and how it is used; introduces and briefly describes consolidated data management; describes the data management/user interface and the relation of data management to other OS/3 software.

- PART 2. CARD, DISKETTE, and PRINTER FILES

  Describes file and format conventions and the function and operation of OS/3 data management in relation to punched card, diskette, and printer files.

- PART 3. MAGNETIC TAPE FILES

  Describes file and format conventions and the function and operation of OS/3 data management in relation to magnetic tape files.

- PART 4. DISK FILES

  Describes file and format conventions and function and operation of OS/3 data management as related to disk files. Describes the indexed sequential access method (ISAM) both with and without an index structure, the sequential access method (SAM), the direct access method (DAM), the indexed random access method (IRAM), the multiple indexed random access method (MIRAM), and the nonindexed access method. Also includes information on disk space management.

- PART 5. PAPER TAPE FILES

  Describes record, character, and file conventions and the functions of OS/3 data management for perforated paper tape files.

- PART 6. APPENDIXES

  Provide selected functional characteristics of peripheral devices relevant to data management use; explain the OS/3 data management procedures for error and exception handling; compare the EBCDIC/ASCII/Hollerith codes and other card codes used in OS/3; describe the systems standard labels for magnetic tape and disk files; and describe the consolidated data management migration considerations.

Statement Conventions

The conventions used to delineate the data management macroinstructions are:

- 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 CNTRL is a data management macroinstruction with three optional positional parameters: A, B, and C.

```
TAG1 CNTRL A
TAG2 CNTRL A,B
TAG3 CNTRL A,B,C
TAG4 CNTRL A,,C
```

- 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; however, a comma must neither be coded in column 16 of a continuation line nor follow the last keyword of a string.

  Example:

  Assume that the data management DTF macro for a card file (called CARDIN) has three keyword parameters: IOAREA1, BLKSIZE, and WORKA.

  ```
  CARDIN  DTFCD IOAREA1=AREA1,BLKSIZE=80,WORKA=YES
  ```

- Capital letters, commas, equal signs, and parentheses must be coded exactly as shown. The exceptions are those acronyms that are part of generic terms representing information to be supplied by the user and the commas preceding keyboard parameters of declarative macroinstructions. (These commas serve to remind the user that keyboard parameters coded in a string must be separated by commas.)

  Examples:

  ```
  FIELDS=([ADDR][,A2TD][,FREQ])
  REOC=(MERGE,label,reel,to)
  CMceNUMBCHAR=n
  X'aa'(NOV)
  ```

- 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).

  Examples:

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

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

  Examples:

  ```
  {filename}
  {(1)      }
  ```

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

  Examples:

  [INPUT=NO]
  [OUTPUT=NO]
  [, {workname / (0)}]

- An optional parameter which has a list of optional entries may have a default specification which is supplied by the operating system when the parameter is not specified by the user. Although the default may be specified by the user with no adverse effect, it is considered inefficient to do so. For each 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 under an if-omitted heading in the parameter description.

  Examples:

  [,MERGE={YES / NO}]

  [, {ASCII / EBCDIC}]

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

  Example:

  param-1,...,param-n

- Commas are required when positional parameters are omitted, except after the last parameter specified.

  Example:

  positional parameter 1, positional parameter 2,, positional parameter 4

Contents

PART 2. CARD, DISKETTE, AND PRINTER FILES

2. CARD FORMATS AND FILE CONVENTIONS

3. FUNCTION AND OPERATION OF PUNCHED CARD SAM

4. DISKETTE FORMATS AND FILE CONVENTIONS

PART 4. DISK FILES

10. ISAM FORMATS AND FILE CONVENTIONS

11. FUNCTIONS AND OPERATION OF ISAM

12. IRAM FORMATS AND FILE CONVENTIONS

13. FUNCTIONS AND OPERATIONS OF IRAM

13A. MIRAM FORMATS AND FILE CONVENTIONS

13B. FUNCTIONS AND OPERATIONS OF MIRAM

14. NONINDEXED DISK FILE FORMATS AND CONVENTIONS

15. NONINDEXED FILE ACCESS METHODS: FUNCTION AND OPERATION

16. SYSTEM RESOURCE CONTROL

PART 5. PAPER TAPE FILES

17. PAPER TAPE DATA MANAGEMENT

PART 6. APPENDIXES

FIGURES

TABLES

PART 1. OS/3 DATA MANAGEMENT

1. Introduction

1.1. THE FUNCTION OF DATA MANAGEMENT

As you know, data processing programs produce desired results by accepting data as input, processing the data as appropriate, and outputting the results of the processing performed.

Because most data movement and retrieval operations are inherently the same, regardless of the application involved, generalized, preprogrammed data management packages have been developed to assist you in performing these tasks.

The degree of assistance you receive from these packages depends on the insight into your problems by data management developers and the success they achieve in providing you with the most flexible and convenient data management aids possible. The extent to which you can inform the data management system of the characteristics of your data and the specific function you want performed on that data is also integral. Therefore, it is necessary to establish conventions to communicate, or interface, with your data management system.

Data management services available to you, the programmer, via OS/3 are varied, flexible, and powerful. Descriptions of these services and conventions for using them go well beyond the scope of what a language manual can and should contain. Hence, this and other manuals dealing exclusively with this subject are provided to facilitate your use of OS/3 data management.

1.2. BASIC AND CONSOLIDATED DATA MANAGEMENT

Until recently, the only method of data management available under OS/3 was DTF (define-the-file) or basic data management. The programmers' means for interfacing with this data management system is through certain declarative and imperative macros related directly to the device from which data is being retrieved or to which data is being moved.

Now under OS/3, another method of data management is available: CDI (common data interface) or consolidated data management.

Consolidated data management generally provides all the services basic data management does, and then some. The single major difference is that MIRAM (multiple indexed random access method) files are the only disk files supported by consolidated data management.

Consolidated data management can best be described by answering the following question: What does consolidated data management provide that basic data management doesn't? The answers are:

- A single uniform set of declarative and imperative macroinstructions

  With basic data management, you must use a specific declarative macroinstruction (DTF) to define your file and the method used to access that file: DTFMT for a magnetic tape file, DTFPR for a printer file, DTFIS for an ISAM disk file, and so on. Also, with basic data management, the imperative macroinstructions are not the same for all types of access methods. Different instructions are used to perform the same functions. For example, to write a record to a tape file you must use a PUT instruction. To write a record to an ISAM file, you must use a WRITE,NEWKEY instruction.

  Consolidated data management, on the other hand, has a uniform set of declarative and imperative macroinstructions that you use to define and process all types of files. There are two declarative macroinstructions. These are the CDIB and RIB instructions. The CDIB instruction identifies the file and the RIB instruction describes the file characteristics and processing requirements. The consolidated data management imperative macroinstructions are also the same for all types of files. For example, if you want to write a record, you use the DMOUT instruction regardless of the file type.

- Control structures cannot be modified

  The control structures for each basic data management DTF macroinstruction are generated and maintained within your program region. As a result, these structures can be inadvertently modified and compromise the integrity of the file.

  Consolidated data management eliminates this problem because all control structures it uses are generated and maintained outside of your program region. As a result, you cannot inadvertently modify these control structures. This preserves the integrity of the file and prevents the distortion of any action taken on that file by data management. The CDIB and RIB macroinstructions generate parameter passing structures that are used to communicate information to data management.

- A single file access method for all disk files

  Basic data management supports a variety of disk access methods: SAM, DAM, ISAM, ASAM, and so on. Thus, you are faced with a decision each time you want to use a disk file. You must decide how you want to process the file and then select the access method that meets your needs or is required by the programming language you intend to use.

  Consolidated data management uses one single disk access method, MIRAM, which provides all the functions provided by the various basic data management disk access methods. As a result, you only have to decide how you want to process the file.

- Enhanced file sharing

  With basic data management a file can be shared; that is, it can be used by more than one program at a time. This sharing, however, is limited because several programs can read from the file at the same time, but only one program can write to the file.

  Consolidated data management, however, allows complete flexibility; more than one program can read from or write to the file at the same time.

- Interactive capabilities

  Basic data management does not support interactive capabilities.

  Consolidated data management, however, supports a wide range of interactive capabilities. These allow you to: enter or display data from a workstation; create workstation screen formats that aid you in entering data or presenting output data; and develop and include dialogs (question and answer sessions) in your program. In addition, consolidated data management also supports interactive services that allow you to operate your jobs from a workstation, perform housekeeping tasks, and communicate with other workstations in your system.

- A high degree of device independence

  Device independence means that, at program execution time, you can change the type of device used for a file in your program by changing the job control device assignment set for that file.

  This is not possible with basic data management because changing the device type requires changing the file definition, recompiling and relinking your program, and changing the job control stream before you can execute it with a different device.

  With consolidated data management, a high degree of device independence is possible whenever you are processing records sequentially. This is possible because device assignment takes place when the file is opened based upon the job control device assignments. As a result, you can change the device that a file is processed on by changing the job control device assignment set for that file. The file you have defined must be compatible with the types of devices you want to use. For example, if you define a file that has an 80-byte record size, the records can be output to a card punch, printer, tape, disk, diskette, or workstation. If, on the other hand, you define a file that has a 200-byte record size, the records can be output to a tape, disk, diskette, or workstation. However, they cannot be output to a card punch or printer without truncating the data because of the physical limitations of these devices. In addition to your files being compatible, your program must also be compatible with the devices you want to use. This means you cannot have any instructions in your program that are device dependent such as random operations when using a disk or diskette, forms control operations when using a printer, or screen management operations when using a workstation.

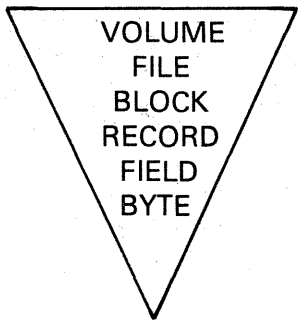
Use of a particular data management mode is specified at systems generation time. The capabilities just described are provided when consolidated data management (CDI mode) alone or in combination with basic data management (CDI/DTF mixed mode) is specified.

This manual specifically describes basic data management. For more detailed information on consolidated data management, see the consolidated data management concepts and facilities, UP-8825 (current version) and the consolidated data management macro language user guide/programmer reference, UP-8826 (current version).

If you want to migrate to consolidated data management, see Appendix F. This appendix describes the migration requirements for programs written in BAL, RPG II, 1968 American National Standard COBOL, 1974 American National Standard COBOL, and FORTRAN.

1.3. DATA STRUCTURE

The structural entities recognized by OS/3 data management are illustrated in the following diagram:



The hierarchy shown is not always followed exactly. The volume concept is not truly applicable to printers or card devices. On disk, diskettes, and magnetic tape, a file may sometimes be larger than a volume. A record may sometimes be equal to a block, or a field equal to a byte. Figure 1—1 illustrates the organization of data on typical peripheral devices.

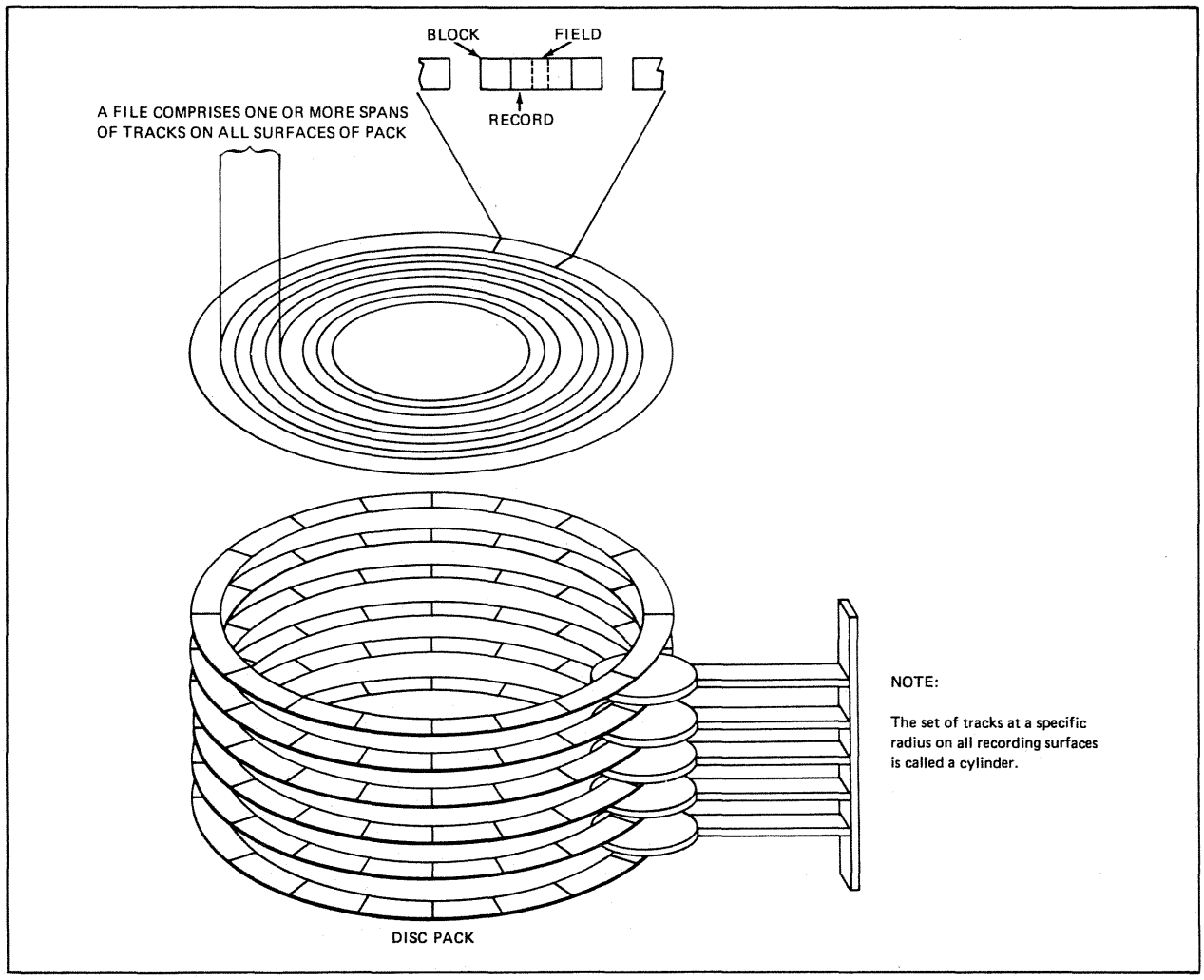


Figure 1—1. Organization of Data on Typical Peripheral Devices (Part 1 of 2)

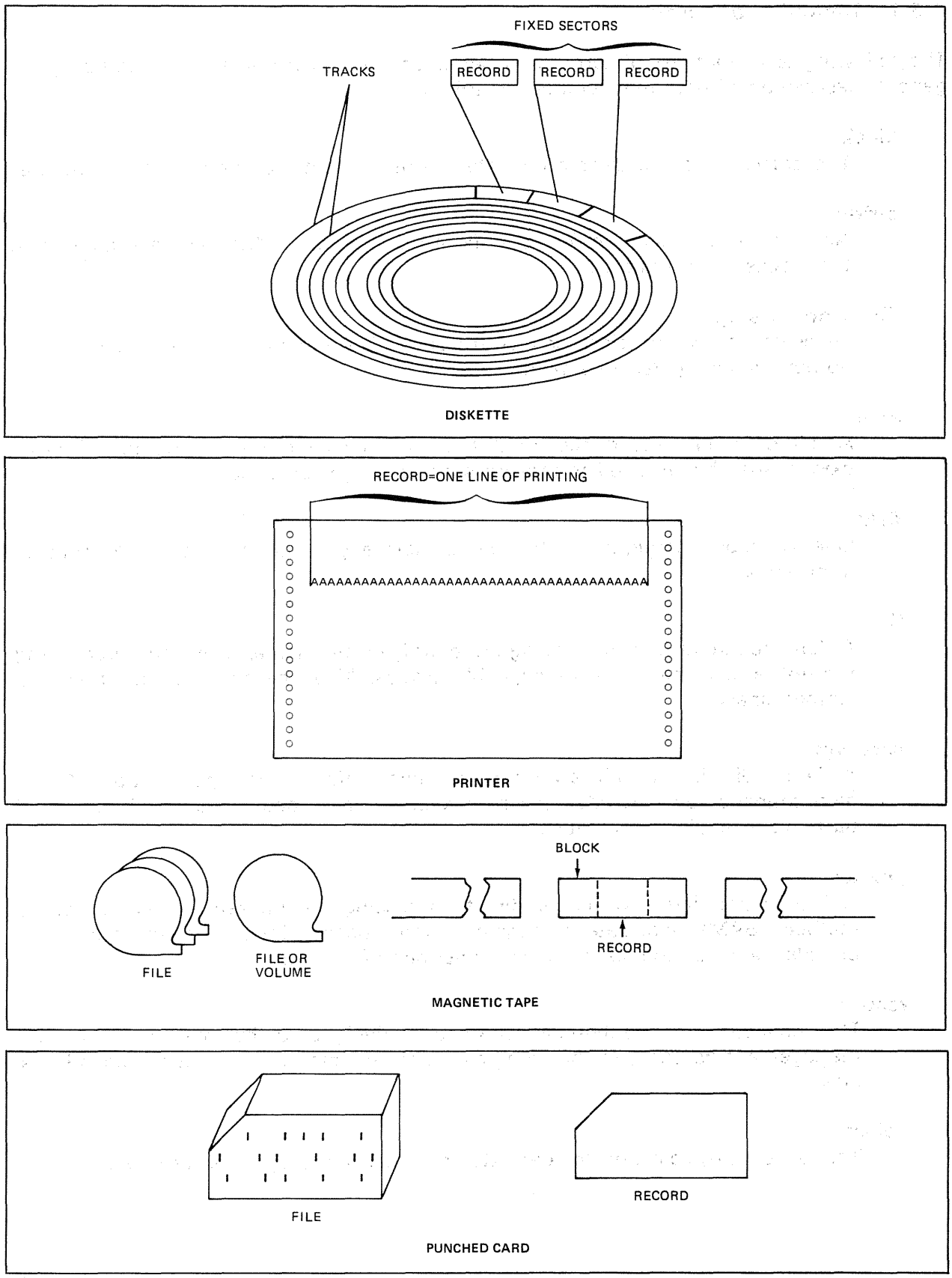


Figure 1—1. Organization of Data on Typical Peripheral Devices (Part 2 of 2)

1.3.1. Definition of Terms

The following is a brief list of terms and definitions to assist you in understanding the general description of data management in this section:

block
The portion of a file transferred into or out of main storage by a single access.

buffer
An area in main storage for handling a block of data. Must not be smaller than the blocks to be handled.

direct addressing
Retrieving a specific block or record from disk storage by a single access, using numeric values given in a field.

extent
A set of contiguous tracks on disk assigned exclusively to one file. Several extents may be required to provide space enough for a file.

field
One or more contiguous characters, normally comprising a single unit of information.

file
A delimited storage space having an identifying file name; useful for subdividing the entire data mass into manageable groups. Also, the data residing in such a storage space.

partition
A file subdivision, which is required to have uniform block specifications. OS/3 data management provides partition-relative block addressing, and individual partition extension capabilities.

pointer
A field containing a value for direct addressing. In indexed sequential access method (ISAM) files, data management introduces pointers between records to provide for maintenance of logical sequence of records.

record
The collection of contiguous characters designated by the user to data management as such, for handling as a unit. Record size must not exceed block size.

volume
The largest physical unit for data storage, such as a tape reel or disk pack.

1.3.2. Punched Card Files

A punched card file consists of a card deck, input via a reader, or output via a punch. Records can comprise either a portion of a card or a complete card. The records are made up of fields of related characters. Punched card files must be treated as sequential files (handled one record at a time in sequential order).

You must not confuse a deck of cards to be handled as a data management card file with control stream cards or with data cards embedded in control streams. You must place the data management deck in the card reader when there is a console message calling for assignment of the reader to the program. You may begin the deck immediately with a data card, but you must end it with an end-of-data card (/*). You cannot place an end-of-data card within the deck.

Details of punched card records and files are presented in Section 2.

1.3.3. Diskette Files

Diskette files are sequential, unblocked files processed similarly to card files. In fact, diskettes are intended as rapid replacement media for card processing equipment. Each diskette is a single-sided, single plate disk with tracks containing fixed sectors. Records are recorded on the tracks, one record per sector. (Sections 4 and 5 discuss the diskette in more detail.)

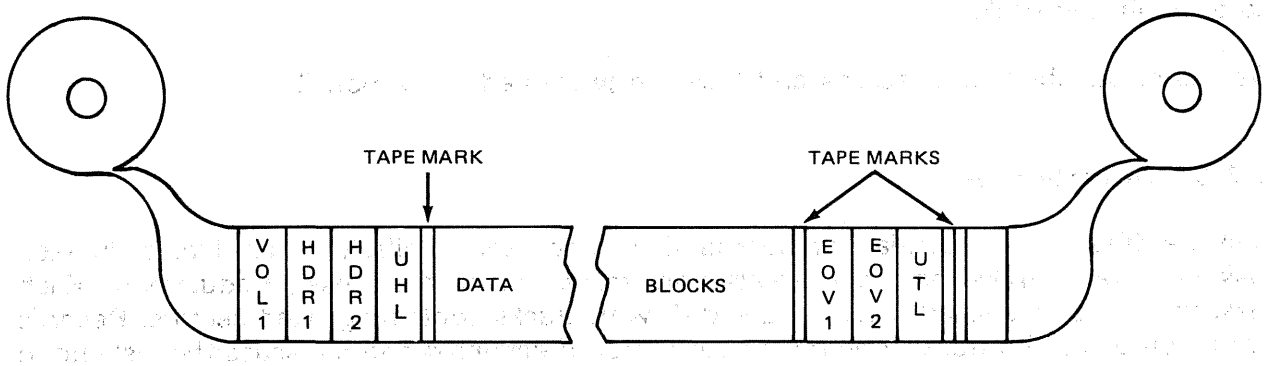
1.3.4. Printer Files

Printer files include standard text, listings, forms, and similar printed ouptut. The files are composed of individual records that are formed in an output area or work area by your program and then output to the printer in increments of one record (line). The file is output, character by character, in a serial manner. When the printer buffers are loaded, the line is then printed. This process repeats, each line in succession, until the entire file is printed. A printer record can also contain certain control characters which, although part of the output record, are not printed. The control characters allow you to advance the paper to a home position, specify a procedure in case of overflow, or select a number of lines to be skipped by the printer. Section 6 gives details on printer files and records; Section 7 describes the uses of control characters.

1.3.5. Magnetic Tape Files

Magnetic tape files are also sequential files, and can span more than one volume (reel). Each magnetic tape file is identified by two file header labels; each volume of the file has a volume label. Because most magnetic tape files can be read in both a forward and backward direction, the file labels are placed both at the beginning and at the end of each of these file levels.

Figure 1—2 illustrates the relationship of the various elements of a magnetic tape file. The volume label (VOL1) has a standard system format that describes the contents of the tape volume. The two file header labels (HDR1 and HDR2) also are in a standard system format. User header labels (UHL), which are optional, may be in a standard format or one that you, as a user, can structure. A tape mark is next in the sequence, and acts as a delimiter to indicate that data blocks or records follow. After the data, another tape mark, two end-of-volume (EOV) labels, optional user trailer labels (UTL), and two more tape marks are provided as delimiters. A complete description of the magnetic tape file organization and conventions is presented in Section 8; Appendix E describes the labels for magnetic tape files.



LEGEND:

VOL1	Volume Label
HDR1	File Header Label 1
HDR2	File Header Label 2
UHL	User Header Labels (optional)
EOV1	End of Volume Label
EOV2	End of Volume Label
UTL	User Trailer Labels (optional)

DIRECTION OF MOVEMENT, FORWARD READ

Figure 1—2. Magnetic Tape File Organization

1.3.6. Disk Files

Provisions for disk files differ from those for sequential devices in that there are several data management programs from which to choose. You implement your choice by selecting one of several operation codes at the point in your program where the DTF *(define the file)* procedure (proc) is coded. You must consider the services offered by the programs to determine which is best suited to your needs for the particular file. (There is a certain amount of overlap in the services available, so it is possible for you to meet a particular need through either of two programs.) The desire for rapid storage and retrieval is usually paramount. In this context, several considerations are pertinent:

- Is search-by-key needed?
- Is appending new data to a series satisfactory, or are insertions necessary?
- Are direct addressing or sequential access, or both necessary?

- Is reading or writing blocks sufficient, or is assistance with records needed?

To satisfy these questions, detailed descriptions of the disk file services are presented in Sections 11 through 13.

1.3.7. Paper Tape Files

Punched (or perforated) paper tape files are handled at the logical record level by a paper tape data management system described in Section 17. The system provided by OS/3 includes macros, transients, and processing modules with which you can define paper tape files and read and write data on paper tape. Translation and letter/figure shifting capabilities are provided.

1.4. PROGRAMMING FOR DATA MANAGEMENT

All users of OS/3 must employ the conventions established for designating existing files and new files in the job control stream. In using data management, you must also code appropriately in your BAL program.

- By issuing a DTF declarative macroinstruction provided by data management, you cause a DTF table to be created in a data area. By using keyword parameters, you describe the file and provide addresses of buffers and work spaces. You must also indicate your desire to handle all returns inline, or you must give the address of your routine for accepting control when errors or exceptions occur.

- By using ordinary assembler instructions, you must reserve sufficient amounts of main storage buffer space and workspace, and you must provide the error/exception routine.

- By issuing imperative macroinstructions provided with each access method, you request data management to perform specific file-processing functions.

- You must realize that general registers 0, 1, 14, and 15 are loaded by the imperative macroinstructions before the contents of your registers are saved. You cannot afford to have vital data in these registers when you call on data management.

- You must provide a 72-byte register save area. The address of this area can be placed in the DTF by specifying the SAVAREA keyword parameter, common to all DTFs. Failing to do this, you must provide the address in register 13 when you enter each data management imperative macro.

- The storage area you specify with the SAVAREA keyword parameter is often useful to examine in snaps or dumps of your program region. It comprises 18 full words, the first three of which are used by data management. Following these is a display of full words for 15 of the general registers, presented in the following order: 14, 15, 0, 1, 2, and so on, through 12. Register 13 is not included.

1.5. OS/3 DATA MANAGEMENT ENHANCEMENTS

OS/3 data management departs from traditional data management systems in several areas.

1.5.1. ISAM Files

In OS/3 ISAM files, inserted records are placed in overflow blocks, forming chains between individual prime records. This causes all data records to remain where originally placed and eliminates time consuming record pushdown. Moreover, the stability of records makes it possible to offer direct addressing to every record, a convenience for those who can benefit from this feature.

The ISAM program can also operate on files where the key index structure is never formed. This precludes the use of keyed instructions, but leaves the rest of the repertoire operative. The ISAM load is still effective for file creation. The resulting file is then susceptible to sequential and direct access without keys.

Eliminating the index does not preclude the ability to insert records. The position for insertion cannot be reached by a key search. However, both direct access and sequential progression are available to reach any record so that a new record may be inserted after it.

1.5.2. SAM and DAM Files

The flexibility of sequential access method (SAM) and direct access method (DAM) processing has been augmented in OS/3 by provision of a DTFNI macroinstruction and processing module. This module supports an extended repertoire of imperative macroinstructions applicable to a file described by the DTFNI macroinstruction. Combinations of SAM and DAM imperative macroinstructions may be used; NOTE and POINT imperative macroinstructions are provided. There is also provision for partitioning a file, using different block specifications for each partition. These are supported by partition-relative block addressing.

1.5.3. IRAM Files

The indexed random access method (IRAM) is an access method in OS/3 for handling disk files and is intended for use by programs written in RPG II language, the sort, and data utilities. The functionality of IRAM is equivalent to that provided by OS/3 ISAM and ASAM, and by the OS/3 nonindexed access disk methods SAM and DAM (relative record addressing); however, those modules (ISAM, ASAM, SAM, and DAM) are considerably larger. The IRAM processor cannot access disk files that have been created by other access methods nor can IRAM files be processed by other OS/3 disk access methods.

1.5.4. MIRAM Files

The multiple indexed random access method (MIRAM) in OS/3 is used for handling sequential, relative, and indexed files in programs. These programs are written in the OS/3 version of the 1974 American National Standard COBOL, and for sequential and relative (direct) files in programs that are written in FORTRAN IV. MIRAM provides the same functions as those provided by OS/3 ISAM, ASAM, IRAM, SAM, and DAM disk access methods. The MIRAM processor can access only MIRAM and IRAM characteristic files that it has created or IRAM files created by the IRAM processor. It cannot access disk files that have been created by other access methods nor can MIRAM files be processed by other OS/3 disk access methods. MIRAM files, however, can be processed by using the sort/merge and data utilities programs.

1.5.5. Error and Exception Returns

OS/3 data management differs somewhat from other data management systems in its method of returning control to your program. Control is always returned, whether or not an error or exception has been detected. A reply field is always set to indicate the nature of the exception. If the function is executed with no defects, control is always returned inline to the instruction following the macro call. If you provide the address of an error/exception routine in your DTF macroinstruction, control is returned to that address on all occurrences of errors or exceptions. In the absence of this address, all returns are made inline (register 14 always contain the inline return address). Appendix B describes the error and exception handling features of OS/3 data management.

Because data management interprets a zero value in the DTF error field of the DTF file table as the nonexistence of an error routine, you must not locate your error routine at location 0 relative to the load module.

Errors occurring during file extend operations are always associated with inability to acquire output space for a buffer and consequent loss of output data. On extend failure errors, file extend procedures now minimize loss of output data to one record.

1.5.6. Disk Flexibility and Hardware Constraints

The obligation to handle disk devices with different characteristics has influenced the design of OS/3 data management. It was considered desirable that the disk file processing modules should be independent of the disk type used and should present the same interface to you. As a result, OS/3 data management requires, throughout, that all blocks in a track or partition be uniform in size and format. On the fixed-sector disk devices, it is also necessary that all blocks be multiples of 256 bytes. Furthermore, spanned records (those extending beyond a block boundary) are not supported.

Consequently, during sequential blocking of records, block filling continues until a submitted record will not fit in remaining block space. At that point, the full-size block is written to disk, and the rejected record is used to begin the next block.

The fixed-sector disk does not provide an RO record for identifying the portion of track devoted to useful data. Furthermore, the hardware search interrogates the first n bytes of *every* 256-byte sector. These characteristics cause some restrictions of relative-track DAM functions. When employing the WRITE,AFTER macro, you must fill each track because the unfilled portion is not identified. When employing keyed operations, you must use a block size of 256 bytes; otherwise, false internal hits could be made in blocks of 512 bytes and other multiples of 256 bytes.

1.5.7. Shared Data Management Modules

Under OS/3, all data management modules are shared-code modules. There is only one data management module for each access method and when a particular access method is requested by a program, one copy of the corresponding module is loaded into main storage. This module is then used or shared by all programs requesting the same access method.

1.6. DATA MANAGEMENT/USER INTERFACE

The interface between you and data management consists of:

- Declarative macroinstructions
- Imperative macroinstructions

1.6.1. Declarative Macroinstructions

Your program must inform the system of the parameters, special conditions, current status, and options pertaining to a file. You must include a declarative (file definition) macroinstruction for each file required by your program. As implied by the term *declarative,* these macroinstructions generate nonexecutable code, such as constants and storage areas for variables. Therefore, you should separate these macroinstructions from the inline file processing coding. The declarative macroinstruction and the selected keyword parameters in the operand define the file. The first three characters of the operation code must be DTF. The last two characters usually indicate the type of device or method of accessing. A keyword parameter consists of a word or code immediately followed by an equal (=) sign and one specification.

The format of the declarative macroinstruction is:

LABEL	Δ OPERATION Δ	OPERAND
filename	**DTFcc**	**keyword-1=x,...,keyword-n=z**

The symbolic name of the file must appear in the label field. The name can have a maximum of seven characters and must begin with an alphabetic character. The appropriate DTF designation must appear in the operation field. The keyword parameters can be written in any order in the operand field and must be separated by commas. However, a comma must neither be coded in column 16 of a continuation line nor follow the last keyword of a string. Appropriate assembler rules regarding macroinstructions apply to blank columns and continuation statements. Register numbers are specified to the data management declarative macroinstructions (DTF) by enclosing the number in parentheses. Certain DTF parameters can be changed at run time via the data definition job control statement (DD). (See Tables 3—1, 7—3, 9—1, 11—3, 13—1, 13B—1, 15—1, 15—2, 15—3, and 17—1.)

The DTFs may have the following forms:

- DTFCD

  Defines an input, output, or combined punched card file.

- DTFDA

  Defines either an input or output direct access disk file.

- DTFIR

  Defines input or output indexed or nonindexed IRAM disk files.

- DTFIS

  Defines an indexed sequential disk file.

- DTFMI

  Defines an input or output indexed or nonindexed MIRAM disk files.

- DTFMT

  Defines an input, output, or in/out magnetic tape file.

- DTFNI

  Defines a nonindexed input and output disk file.

- DTFPR

  Defines a printer output file.

- DTFPT

  Defines an input or output paper tape file.

- DTFSD

  Defines an input, output, or combined sequential access disk file.

- DPCA

  Similar to a DTF macroinstruction, but defines a partition of a disk file rather than the entire file.

1.6.2. Imperative Macroinstructions

Your program must be able to communicate with the data management modules in order to process files that have been defined by declarative macroinstructions. Imperative (file processing) macroinstructions included in your program communicate with the transient routines and logical IOCS shared-code modules. The imperative macroinstructions are expanded as inline executable code. Not all macroinstructions are available for use on all devices. Some are specifically input-type macroinstructions and cannot be used for a device that is exclusively used for output; the opposite is true, also.

The format of the imperative macroinstruction is:

LABEL	Δ OPERATION Δ	OPERAND
[name]	xxxx	yyyy,...,zzzz

A symbolic name can appear in the label field. The name can have a maximum of eight charcters and must begin with an alphabetic character. The appropriate verb or code must appear in the operation field. The positional parameters (as signified by the name) must be written in the specified order in the operand field and be separated by commas. When a positional parameter is omitted, the comma must be retained to indicate the omission except in the case of omitted trailing parameters. Appropriate assembler rules regarding macroinstructions apply to blank columns and continuation statements.

1.6.3. Assembler Rules for Operand Field

The operand field of a macroinstruction begins in column 16 and may not extend beyond column 71. An operand may be continued onto the next line by inserting an arbitrary nonblank character in column 72. Each continuation line starts in column 16.

The operand field is terminated by the first blank which is not enclosed within apostrophes. As operand specification is usually completed before column 40, columns 41 through 71 are available for comments, but at least one blank space must occur between the end of operand specification and the beginning of the comments.

Comments are not continued by the insertion of a nonblank character in column 72. Lengthy comments can be entered by coding an asterisk (*) in column 1. You will note the applications of these rules in the programming examples throughout this manual. Operands may be continued onto the next line by placing a comma after the last operand on the first line and a nonblank character in column 72. However, if you omit the comma and at least one blank exists between the last operand on the first line and the nonblank character in column 72, the second line of operands is treated as comments. Because the second line is treated as comments and not as part of your operand specification, the assembler does not flag the missing comma as an error. Up to two comment lines are permitted.

1.7. RELATED OS/3 SOFTWARE

Several OS/3 software components are indirectly involved with data management, while others perform functions related to and required for program operation. These components include:

- System service programs (SSP)
- Job control
- Supervisor
- Linkage editor
- Data utilities

1.7.1. System Service Programs (SSP)

The service programs provided to prepare disk and magnetic tape files to accept data records and blocks are the disk prep program and the magnetic tape prep program.

The disk prep routine performs a surface analysis for the disk tracks and assigns alternate tracks if defects are discovered. The disk prep also establishes a volume table of contents (VTOC) for the device so that files can then be placed on the disk.

The magnetic tape prep routine prepares magnetic tapes in standard label format by writing the initial volume label, dummy file header label, dummy file trailer label, and tape marks.

Other system service programs include dump routines in non-narrative or narrative formats. The SYSDUMP and JOBDUMP routines provide the resident shared-code directory and the preamble EXTRN table in narrative format.

These routines are described in detail in the system service programs (SSP) user guide, UP-8062 (current version).

1.7.2. Job Control

The main functions of job control, as related to data management, are:

- Allocation of required peripheral devices
- File control block (FCB) management
- Catalog management allowing automatic identification of files by name
- Loading printer vertical format buffer (VFB) and load code buffers
- Defining software facilities (SFT) needed to support the user program
- Modifying DTF specifications at run time (DD).

Peripheral devices are assigned through job control statements that specify logical unit numbers, alternate device types, and information about the file. These job control statements include:

// DVC Statement assigns device number.

// VOL Statement describes tape and disk volumes.

// EXT Statement provides disk extent information.

// LBL Statement provides additional tape and disk identification information.

// LFD Statement links the file defined by the DTF macroinstruction with the file and device information in the control stream.

Each part of this manual that deals with a particular access method or device type provides you with job control stream examples that illustrate the relationship between data management entries coded for program assembly and the job control stream statements that control the program.

A separate file control block is maintained automatically in main storage for each active file. This block contains all descriptive information about the file and is used for reference when the file is being accessed.

OS/3 automatically loads the data management modules needed by your job. However, if you have written your own shared-code modules, you must use the SFT job control statement to identify and load these modules. SFT statements are effective only during the job step in which they are specified.

For details on OS/3 job control, refer to the job control user guide, UP-8065 (current version).

1.7.3. Supervisor

The supervisor provides the greatest amount of support for the user program and data management. This support includes the following:

- PIOCS

  Those macroinstructions and routines that schedule and monitor execution of channel programs, controlling the actual transfer of physical records between external sources and main storage. These routines also provide for device I/O error recovery.

- Transient scheduling

  The routines that retrieve transients from auxiliary storage and bring them into main storage for execution. These include file *open* and *close* routines.

- Operator communication

  The routines that handle the communications concerning volume mounting requests, tape mount requests, etc.

- File protection

  Protection of files and records during shared file processing.

- Timer services

  Used as a reference for computing run time, scheduling, etc.

- Disk space management

  Routines for allocating space to disk files and maintaining space accounting through standard procedures for updating the volume table of contents (VTOC).

- System Access Technique (SAT)

  An input/output control systems that provides a standard interface for tape and disk subsystems between OS/3 data management and the PIOCS.

For details concerning the supervisor, refer to the supervisor user guide, UP-8075 (current version).

1.7.4. Linkage Editor

The linkage editor is a system service program that constructs a load module from object modules. The linkage editor control statements that define the load module are contained in the job control stream beginning with a LOADM control statement and terminated by another LOADM control statement or by an end of data (/*) job control statement. For details concerning the linkage editor, refer to the system service programs (SSP) user guide, UP-8062 (current version).

1.7.5. Data Utilities

The OS/3 data utility and service routines are provided to assist you in manipulating data files and preparing card decks. Your use of these routines requires only a minimum amount of programming effort. You simply code the appropriate job control statements, together with utility and data statements or control specifications, to exchange information with OS/3, submit parameters, and start your job.

The OS/3 data utilities and service routines are described in detail in the data utilities user guide/programmer reference, UP-8069 (current version):

PART 2. CARD, DISKETTE, AND PRINTER FILES

2. Card Formats and File Conventions

2.1. GENERAL

This section describes the data formats and file conventions that apply to the card reader and card punch subsystems supported by the SPERRY UNIVAC 90/30 System and the OS/3:

- SPERRY UNIVAC 0604 Card Punch Subsystem
- SPERRY UNIVAC 0605 Card Punch Subsystem
- SPERRY UNIVAC 0716 Card Reader Subsystem
- SPERRY UNIVAC 0717 Card Reader Subsystem
- SPERRY UNIVAC 0719 Card Reader Subsystem

For the functional characteristics of these subsystems, refer to Appendix A.

2.2. FILE ORGANIZATION

Your punched card decks may include a start-of-data job control card at the beginning and must include an end-of-data job control card at the end of the card deck (Figure 2—1). Punched card files can be input (card read), output (card punched), or combined (read/punched).

The basic punched cards for subsystems supported by OS/3 are standard 80-column, 12-row rectangular tab cards. However, optional hardware features, available on both 0716 and 0717 card readers, allow reading of 51- and 66-column stub cards. The 0716 is capable of reading 96-column card data files.

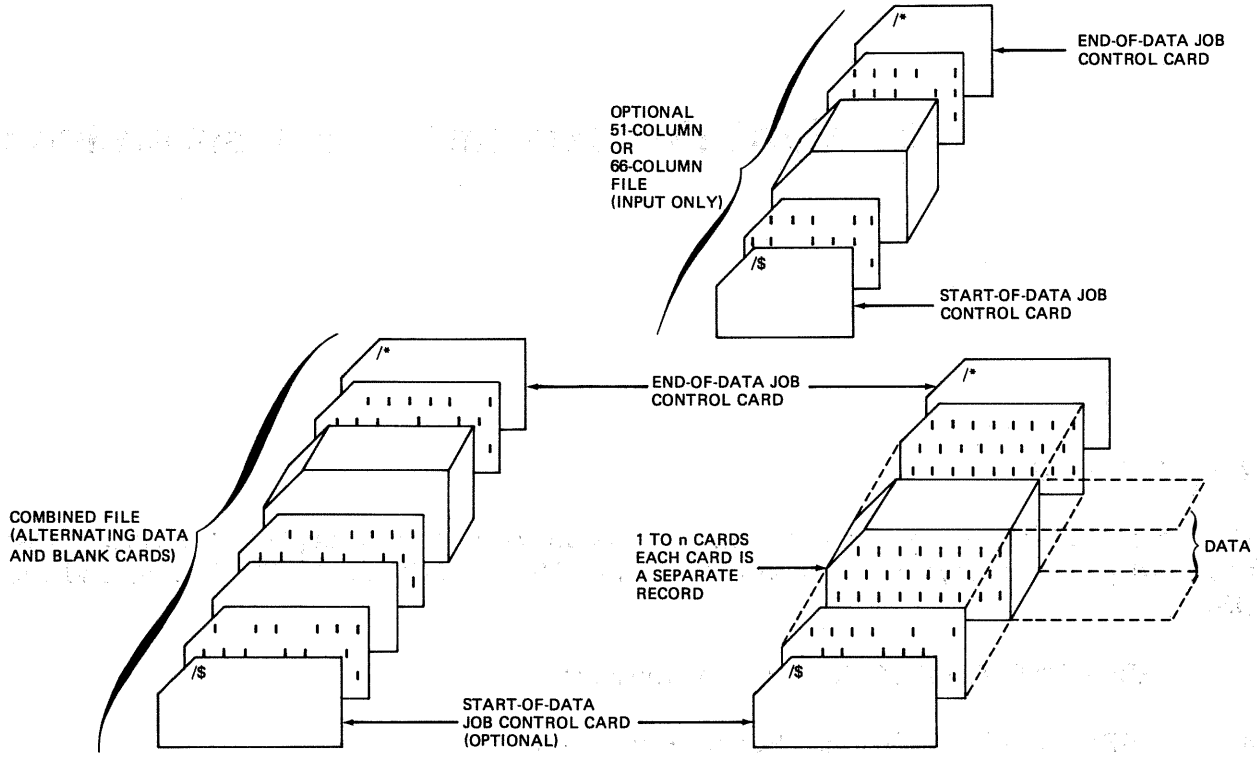
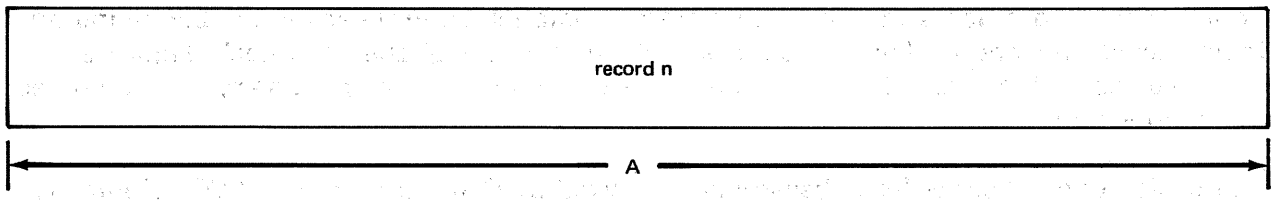


Figure 2—1. Typical Card File Structure

2.2.1. Card Input Files

The card reader handles fixed-length unblocked records, which always have the same length for your entire file. This length is equivalent to the value you selected for the BLKSIZE keyword parameter of the DTFCD macroinstruction when defining the file. Figure 2—2 illustrates the fixed-length unblocked format related to card input files; the same format is used for combined files (8.2.3).



NOTES:

1. The record length, A, must be an even number of bytes, at least as many as specified by the BLKSIZE keyword parameter.

2. The I/O area must be aligned on a half-word boundary and comprise an even number of bytes.

3. When 51-column stub cards are processed, the BLKSIZE keyword parameter may specify a 51-byte length, but the I/O area must be 52 bytes in length. A work area may be 51 bytes long.

Figure 2—2. Fixed-length Unblocked Record Format for Input and Combined Card Files

2.2.2. Card Output Files

The card punch files (output) consist of data that is formed into physical records, usually in the I/O area, and then output to the card punch, where the records are punched in the standard 80-column format. The cards are then accumulated in a stacker, which keeps them in sequence.

2.2.3. Combined Files

Combined read/punch files are allowed only where the optional read/punch feature is installed as part of the card punch. This feature allows you to read cards and punch cards in the same file (deck) on a single pass through the card punch. Reading and punching of cards can be accomplished in the following ways:

- Data can be read from a card then punched on the same card. This requires the nonoverlap mode of processing (3.3).
- A card deck containing alternating punched and blank cards can be entered; each punched card is read and data is punched on the blank card following. The overlap mode of processing must be specified (3.3).
- Punched cards and blank cards can be grouped; the punched cards are then read, and the following group of blank card is punched with the new data. The overlap mode of processing must be specified (3.3).

2.3. RECORD FORMATS

2.3.1. Start-of-Data Job Control Statement (/$)

Data management does not check for a start-of-data card. For consistency, you may choose the /$ card convention as a card file identification. If you do so, your program should include a check for this card.

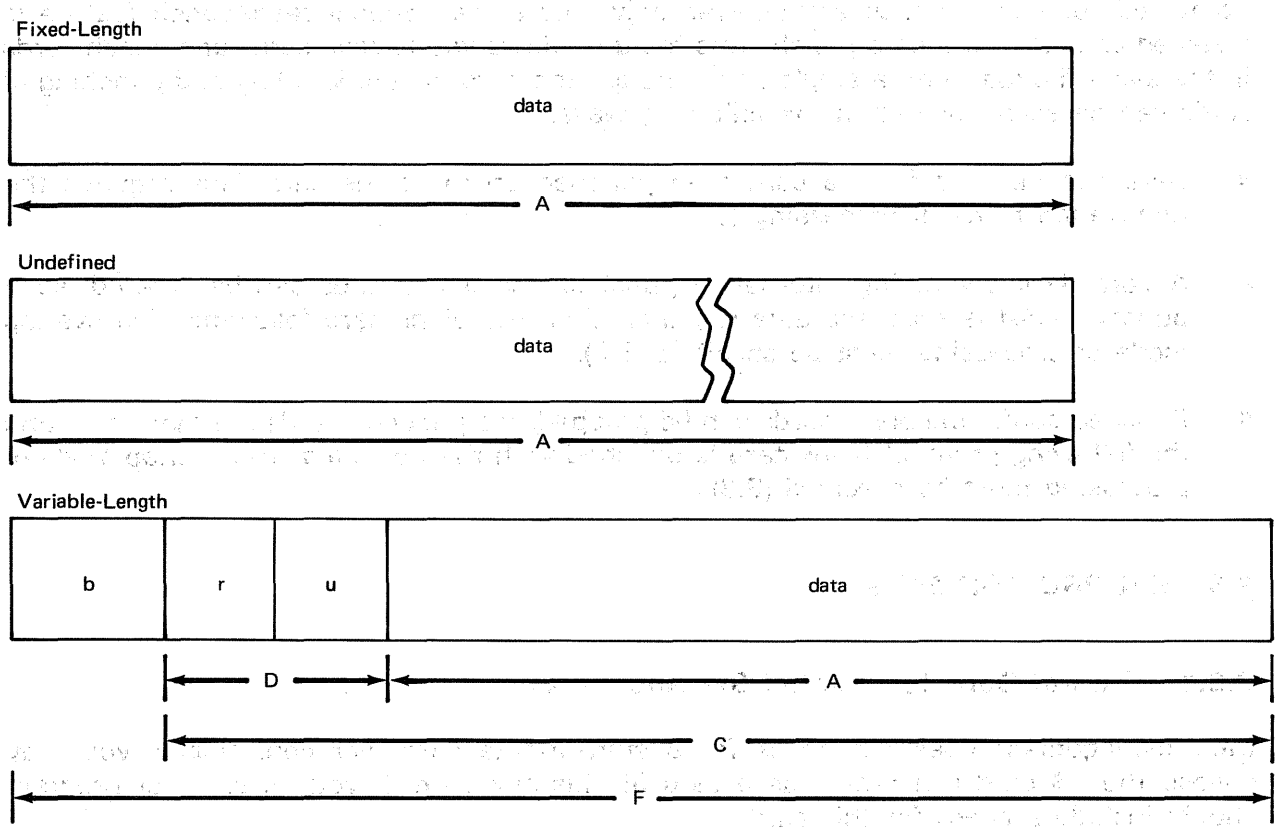
2.3.2. End-of-Data Job Control Statement (/*)

Data management checks for an end-of-data card when you are reading cards. The format of this card is identical to that required by job control. The first two columns contain /*. When this configuration is sensed, control is transferred to the end-of-file address specified for the file. When an output file is punched, the end-of-data card is not punched by logical IOCS. You must supply the end-of-data card for input and combined files.

2.3.3. Card Punch Records

You may form card punch records either in the I/O area or in a designated work area of main storage. The records, illustrated in Figure 2—3, are of three types:

- Fixed-length records
- Variable-length records
- Undefined-length records



LEGEND:

b Block size field, four bytes
r Record length field, two bytes, binary
u Reserved (two bytes); may be any two characters chosen by the user
A Data record length
C Variable record length
D Record size field
F I/O area layout

NOTES:

1. An I/O area must be so aligned that the first character to be punched falls on a half-word boundary.
2. Record length, as a binary number, must be placed in the first two bytes of the record length field (r) before punching a variable-length, unblocked record.
3. An even number of bytes should be allocated for data in I/O areas, even though an odd number of columns are to be punched. The I/O areas for a file with an odd block size should provide at least block size+1 bytes.

Figure 2—3. Card Punch (Output File) Record Formats

3. Function and Operation of Punched Card SAM

3.1. GENERAL

The OS/3 includes data management modules that you can use to move and manipulate sequential access method (SAM) card reader files, card punch files, and combined (read/punch) files. These modules allow you to configure your program for each particular application and related device types.

This section contains a brief functional description of punched card SAM operation for input files, output files, and combined input/output files. Following the functional description is a detailed explanation of the declarative macroinstructions that define the three types of files. The section concludes with detailed descriptions of the imperative macroinstructions that initiate, conduct, and terminate file processing.

3.2. FUNCTIONAL DESCRIPTION

3.2.1. Punched Card Input

The card reader is a unit record device and is connected to the integrated peripheral channel or to the multiplexer channel, if several relatively slow peripheral devices are to share I/O jointly. The punched card file comprises data in the Hollerith punched card code (Appendix C). The cards are usually divided into fields; these files, in turn, are combined to form physical records.

You define, to the system, the type of file, structure of the data, and the operating environment in which your file will be processed through a *define the file* (DTFCD) declarative macroinstruction. At system installation, the system macro library file (YMAC) is loaded with source code modules that are common to several machine operations. These modules include data management modules that are common to several device types and access methods.

When assembling the program, you define the files (input, output, or combined) used in the operation through the DTFCD macroinstruction. The source modules for the particular data management operation are called in from the macro library during program assembly by using imperative macroinstructions which place the modules in your program as inline code. Each macroinstruction available for punched card file processing is described in detail in 3.3 and 3.4.

3.2.2. Punched Card Output

The punched card output records are constructed in the I/O area or a designated work area. Processing data and creating an output on punched cards are similar to the procedure described in 3.2.1 except that the CNTRL macroinstruction can be used with the 0604 Card Punch Subsystem (0604 card punch).

If the punched card deck is to be inserted in a program, you must punch a start-of-data (/$) job control card and an end-of-data (/*) job control card and add these to the beginning and end of the punched card deck.

DTFCD (card)

3.3. DEFINE A SAM CARD FILE (DTFCD)

Function:

The DTFCD declarative macroinstruction is required for you to define punch card files that are accessed by OS/3 SAM. Following is a listing, in alphabetic order, of the required and optional keyword parameters that might appear in the operand of the DTFCD macroinstruction. A summary of the keyword parameters is provided in Table 3—1.

A comma is shown preceding each keyword parameter except the first, to remind you that all keywords coded in a string must be separated by commas. However, a comma must neither be coded in column 16 of a continuation line, nor follow the last keyword in the string. Refer to the coding examples which follow.

Format:

LABEL	Δ OPERATION Δ	OPERAND
filename	DTFCD	[ASCII=YES] [,AUE=YES] [,BLKSIZE=n] [,CONTROL=YES] [,CRDERR=RETRY] [,EOFADDR=symbol] [,ERROR=symbol] ,IOAREA1=symbol [,IOAREA2=symbol] [,IOREG=(r)] [,ITBL=symbol] [,MODE= { BINARY / CC / STD / TRANS }] [,OPTION=YES] [,ORLP=YES] [,OTBL=symbol] [,OUBLKSZ=n] [,RECFORM= { FIXUNB / UNDEF / VARUNB }] [,RECSIZE= { (r) / n }] [,SAVAREA=symbol] [,STUB= { 51 / 66 }]

LABEL	Δ OPERATION Δ	OPERAND
filename (cont)	DTFCD	[,TYPEFLE= { INPUT / OUTPUT / CMBND }] [,WORKA=YES]

Keyword Parameter ASCII:

ASCII=YES
Specifies processing in American Standard Code for Information Interchange (ASCII).

For input files, you must specify this parameter if you desire your card data to be translated into ASCII code for internal processing and storage.

For output files, you must specify this parameter if internal processing is in ASCII and you desire output in Hollerith punched card code (Appendix C).

The keyword parameter MODE should be written as MODE=STD if this parameter is supplied.

Keyword Parameter AUE:

AUE=YES
Inhibits data management error processing when validity check errors are detected on nonbinary input files.

In punched card input files, a validity check error (also termed a unique unit error) is the occurrence of more than one punch in rows 1 through 7 of any column in a card, and usually indicates a mispunch. Each time a validity check error is detected, the operator receives a PIOCS message at the system console indicating the problem. The card containing the error is the last card in the stacker. The operator has three options: He can place the error card in the input hopper and reply "R" to reread the card, or he can reply "I" or "U", to indicate that the error is to be ignored or is unrecoverable. If you have specified AUE=YES, and the operator replies "I" or "U", data management does not branch to your error routine. The error card is skipped (not passed on to the user).

On the other hand, if you do not specify AUE=YES and a validity check error is detected, data management branches to your error routine if the operator replies "U". When your error routine receives control, data management will have set the *unique unit error* flag (byte 0, bit 2) of *filenameC* in your DTFCD file table. Refer to Appendix B. Data management ignores the AUE keyword parameter if 8413 diskette files are used.

Keyword Parameter BLKSIZE:

BLKSIZE=n
Specifies the length of the I/O area in bytes. If the records in a file are variable length, *n* specifies the maximum size for records. For variable, unblocked records, *n* includes the block size and record length fields.

The user can specify BLKSIZE=1 to BLKSIZE=96 to read from 1 to 96 columns of a 96-column card.

A user program that specifies BLKSIZE=1 to BLKSIZE=80 can read from 1 to 80 columns of 96-column cards. A program that has a BLKSIZE of 1 to 80 and that has been used with 80-column cards, in any mode except binary, can read 96-column cards with no program changes.

A program that specifies BLKSIZE=1 to BLKSIZE=96 to the DTFCD proc call can be used to read up to 96 columns of a 96-column card. Such a program can also read up to 80 columns of 80-column cards. The DMCS will blank out (insert the appropriate blank character, based on data mode) bytes in the user's I/O and work areas for columns beyond 80. At OPEN time, DMCS checks for 80-column cards being read with BLKSIZE from 81 to 96. If such a condition exists, a message is issued to the operator with a required reply.

If omitted, the block size is determined from the keyword parameters MODE, RECFORM, and STUB.

Keyword Parameter CONTROL:

CONTROL=YES
Specifies that your program will issue one or more CNTRL imperative macros to control stacker selection on the 0604 card punch; used only for output or combined card files.

The use of the CNTRL macro, which applies neither to input card files nor to the 0605 card punch, is explained in 3.4.4. If you specify CONTROL=YES in the DTF for an input file, the parameter is ignored, and a diagnostic message is printed in the DTF expansion in your assembly listing.

Keyword Parameter CRDERR:

CRDERR=RETRY
Specified if card punch error recovery should be attempted on hole-count errors for 0604 and 0605 combined read/punch files or on the 0604 card punch output files with stacker selection. Error cards are automatically selected into the error select stacker. If error recovery is not successful, the logical IOCS returns control to the address of the user's error routine (ERROR). If keyword parameters CRDERR and ERROR are not specified, the card system returns to your program inline when a punch error (hole count error) is encountered. See 3.6.2.

Error recovery is provided for hole-count errors on 0604 combined read/punch card files, 0604 card output files with stacker selection, and punch check errors on 0605 combined read/punch card files. If the 8413 diskette is used, the CRDERR=RETRY parameter is ignored.

Keyword Parameter EOFADDR:

EOFADDR=symbol
Specifies the address to which control is transferred when the end-of-data card is sensed. This keyword parameter is required for all input and combined files.

Keyword Parameter ERROR:

ERROR=symbol
Specifies the address of your error handling routine. When a fatal hardware or detectable logical error occurs on a file, you may have control transferred to a special error handling routine. If not specified, errors return inline (see 3.6 and Appendix B).

Keyword Parameter IOAREA1:

IOAREA1=symbol
Specifies the address of an I/O area that each input or output file must have reserved for its individual use. Keyword parameter IOAREA1 specifies the input area for a combined file; IOAREA2 must also be used to specify the combined file's output area. I/O areas must contain an even number of bytes for data to be punched or read. Odd numbers of columns can be read or punched. If the BLKSIZE specification is an odd number of bytes, the I/O areas must be at least BLKSIZE + 1 bytes long; this means, for example, if you are using all of the 51-column stub card and have therefore specified BLKSIZE=51, that the length of the storage area you define for IOAREA1 must be 52 bytes. The first data byte (character read or to be punched) must be aligned on a half-word boundary. The length of the area is specified by the keyword parameter BLKSIZE.

Keyword Parameter IOAREA2:

IOAREA2=symbol
May specify a secondary I/O area for standby processing; must be used to specify the output area for a combined file. You must allocate I/O areas that provide an even number of bytes of data. The first data byte must be aligned on a half-word boundary.

Keyword Parameter IOREG:

IOREG=(r)
Specifies the number of the general register (2 through 12) used to reference current data. If SAVAREA is specified, register 13 may be used for IOREG. If a work area is not required, this keyword parameter must be specified when there are two I/O areas. This parameter may not be specified if either a work area or a combined file is specified through the DTFCD macroinstruction. Do not specify WORKA if this parameter is specified.

Keyword Parameter ITBL:

ITBL=symbol
Specifies the address of the 256-byte translation table in your problem program when records in an input or combined file are to be translated on input. If the keyword parameter MODE=TRANS is specified, the keyword parameter ITBL must also be specified.

Keyword Parameter MODE:

This keyword parameter is used to specify the input/output mode of the file and is required as part of the DTFCD macroinstruction. There are four forms of the keyword parameter which can be used with all types of files:

MODE=BINARY

This form is used for cards read on the card reader in binary mode or for cards read or punched on the card punch in column binary (image) mode. An I/O area of 160 bytes is required for one 80-column card.

The binary mode is not available for 96-column cards. This parameter is ignored if the 8413 diskette is used.

MODE=CC

On the card punch, this form must be specified for cards read or punched in compressed code. An I/O area of 80 bytes is required for one 80-column card.

MODE=STD

Should be specified for cards to be read or punched in EBCDIC. This keyword parameter must be specified if the ASCII=YES parameter is specified. If no MODE keyword is supplied, this option is assumed.

MODE=TRANS

You should specify this option to have cards read in EBCDIC and translated by your ITBL translation table, or translated by your OTBL translation table and then punched in EBCDIC. Each position of your 256-byte translation table contains a bit-pattern you have assigned to it.

On reading a byte or card column in the record to be translated, data management places into the receiving byte of your I/O area the bit-pattern it finds in the position of your translation table which corresponds to the position which the bit- or hole-pattern to be translated occupies in Table C—1 (Appendix C). For example, on reading 12-0-9-8-1 hole-pattern, which occupies position 0 in the EBCDIC column labeled *Hollerith Punched Card Code* in Table C—1, data management will place into your I/O area the bit-pattern it finds in position 0 of your ITBL translation table. If you move to your I/O or work area the bit-pattern which occupies position 1 of your OTBL translation table, data management will punch the hole-pattern (12-9-1) which occupies position 1 in the EBCDIC column labeled *Hollerith Punched Card Code* in Table C—1, and so on.

Do not use the *Hollerith Punched Card Code* column in the ASCII portion of Table C—1 for this translation table feature.

Keyword Parameter OPTION:

OPTION=YES

Specifies an optional file: one which you anticipate will not invariably be required for every execution of your program.

When the OPTION keyword parameter is used, *optional file processing* is performed by data management:

- if the OPT positional parameter is included in your DVC job control statement and the device is not avaialble at execution time; or
- when no device is assigned to the file by your job control statements (i.e., no DVC-LFD device assignment set).

Optional file processing:

- For an input or combined file, which you issue a GET imperative macroinstruction, data management branches to your end-of-file routine (EOFADDR). No cards are read. You should close the card file.
- For an output or combined file, if you issue a PUT or a CNTRL imperative macroinstruction, data management disables these and immediately returns control to your program at the normal point. No I/O is performed.

If you do not specify OPTION=YES, and one of the foregoing conditions occurs, the file is not opened. Data management branches to your error routine, if you have supplied one, or to the normal return point in your program if you have not. You will not be able to perform further processing on the file.

Keyword Parameter ORLP:

ORLP=YES
May be specified for combined files processed in overlap mode, when you are using a card read/punch unit with the prepunch read station feature installed.

In the overlap mode, each GET or PUT macro processes a different card. Use this mode to read a card and then punch data on the following card. In the nonoverlap mode, you can read and punch the same card. If you issue a GET macroinstruction, you cause a card to be read. If you issue a GET macro and then a PUT macro, you punch data on the same card that was read by the GET macro. In either mode of operation, you can issue a series of GET macros or a series of PUT macros. Five successive GET macros read five cards; five successive PUT macros punch five cards.

Three possible combinations for issuing GET and PUT macroinstructions are:

1. Alternating GET and PUT macroinstructions, when used with alternating prepunched and blank cards, produce valid results if overlap is specified. Each GET macroinstruction applies to prepunched input data cards, and each PUT macroinstruction applies to punching data into a blank card.

2. Multiple GET macroinstructions between single PUT macroinstructions, when used with multiple prepunched cards between single blank cards, produce valid results if, in every case, the number of GET macroinstructions corresponds to the number of prepunched cards between each of the blanks that the PUT macroinstructions reference.

3. Multiple GET and multiple PUT macroinstructions, when used with multiple prepunched cards between multiple blanks, produce valid results if the number of GET macroinstructions and PUT macroinstructions and the number of prepunched and blank cards are consistent through the program.

Keyword Parameter OTBL:

OTBL=symbol
Specifies the address of the 256-byte translation table in your program when records in an output or combined file are to be translated on output. A translation table is required if the keyword parameter MODE=TRANS is specified.

Keyword Parameter OUBLKSZ:

OUBLKSZ=n
Specifies the length (in bytes) of the secondary I/O area (IOAREA2) for a combined file. If OUBLKSZ is omitted, the size of the output block is assumed to be the same length as BLKSIZE.

Keyword Parameter RECFORM:

RECFORM=FIXUNB
Fixed-length unblocked records are assumed by the logical IOCS when this keyword parameter is omitted. For input or combined files, this option (RECFORM=FIXUNB) must be used.

RECFORM=UNDEF
Used for undefined records in output files only. You must specify the RECSIZE keyword parameter when this option is used. If the 8413 diskette is used, this parameter specification causes the generation of an invalid DTF field message, DM61.

RECFORM=VARUNB
Used for variable-length, unblocked records in output files only.

Keyword Parameter RECSIZE:

RECSIZE=(r)
Specified for output files with undefined record format; (r) indicates the number (2 through 12) of the general register that holds the length of the output record. The record size must be entered into the general register before the PUT macroinstruction is issued. If SAVAREA is specified, register 13 may be used for RECSIZE.

RECSIZE=n
Specifies the record size in bytes used in conjunction with the BLKSIZE parameter value. Data management uses both values to invoke multi-sector I/O operations in processing diskette files.

Keyword Parameter SAVAREA:

SAVAREA=symbol
Specifies the label of a 72-byte register save area, aligned on a full-word boundary.

Specified for each card file defined for a program. Only one user register save area is needed for each program.

If you have a program written for the SPERRY UNIVAC 9200/9300 Series, in which register 13 is employed, it may be converted to OS/3 specifications by adding a 72-byte labeled save area (aligned on a full-word boundary) and by specifying the SAVAREA keyword parameter. Refer to 1.4 for the content of this area.

If SAVAREA is not specified, register 13 must be loaded with the address of a 72-byte register save area, aligned on a full-word boundary, before any imperative macros are issued.

Keyword Parameter STUB:

This keyword parameter is used with 0716, 0717, and 0719 card readers and must be supplied when the stub card read feature is to be used. If the 8413 diskette is used, both the STUB=51 and STUB=66 parameter specifications are ignored.

STUB=51
The stub card read feature applies to cards with 51 columns.

STUB=66
The stub card read feature applies to cards with 66 columns.

The block size specified (BLKSIZE=n) must be less than or equal to the number of columns in the card; however, because of the requirement for an even number of bytes in the length of the I/O buffer, you must reserve 52 bytes for the buffer in defining it with a DC or DS statement in your program when you are reading all columns of a 51-column stub card.

If omitted, standard 80-column cards are assumed.

Keyword Parameter TYPEFLE:

TYPEFLE=INPUT
Describes an input file only. This option is assumed if this keyword parameter is not specified.

TYPEFLE=OUTPUT
Describes an output file only.

TYPEFLE=CMBND
Describes the combined file when both the read and punch features are to be used.

Keyword Parameter WORKA:

WORKA=YES
Specified if I/O records are to be processed in a work area rather than in the I/O area. The address of the current work area must be specified with each GET or PUT macroinstruction. If this keyword parameter is specified, the keyword parameter IOREG must not be specified.

The following are examples of coding the DTFCD macroinstruction.

Examples:

```
LABEL     ΔOPERATIONΔ     OPERAND                                  72
1         10              16
*READER   FILE DEFINITION -SAMPLE
CARDIN    DTFCD   IOAREA1=AREA1,                                   X
                  BLKSIZE=80,                                      X
                  EOFADDR=LASTCD,                                  X
                  RECFORM=FIXUNB,                                  X
                  TYPEFLE=INPUT,                                   X
                  WORKA=YES

*PUNCH    FILE DEFINITION -SAMPLE
CARDOUT   DTFCD   IOAREA1=AREA1,                                   X
                  IOAREA2=AREA2,                                   X
                  IOREG=(5),                                       X
                  BLKSIZE=80,                                      X
                  RECFORM=UNDEF,                                   X
                  RECSIZE=(71),                                    X
                  TYPEFLE=OUTPUT,                                  X
                  MODE=STD,                                        X
                  ERROR=EXIT
```

Table 3—1. Summary of Keyword Parameters for the DTFCD Macroinstruction (Part 1 of 2)

Keyword	Specification	Input	Files		Remarks
			Output	Cmbnd	
ASCII	YES	X	X	X	Specifies processing in ASCII; if used, MODE=STD must be specified
AUE*	YES	X			Skip cards containing validity check errors.
BLKSIZE**	n	X	X	X	The maximum block size in bytes
CONTROL*	YES		X	X	Specified if CNTRL macro is to be issued to file
CRDERR*	RETRY		X		For card punch error recovery; if used, CONTROL must not be specified
EOFADDR	symbol	R		R	End-of-file routine for input and combined files
ERROR	symbol	X	X	X	Address of the user's unrecoverable error routine
IOAREA1	symbol	R	R	R	Address of input/output area; output area for a combined file
IOAREA2	symbol	X	X	X	Address of alternate input/output area; output area for a combined file
IOREG	(r)	X	X		General register (2—13) that contains the address of the current record when processing in two I/O areas. Omit WORKA=YES. Must not be used for a combined file
ITBL	symbol	X		X	Address of input translate table; required when MODE=TRANS is specified
MODE	BINARY*	Y	Y	Y	Specifies cards are to be read or punched in column binary
	CC	Y	Y	Y	Specifies records are to be read or written in compresed code
	STD	Y	Y	Y	Specifies records are to be read and written in EBCDIC
	TRANS	Y	Y	Y	For records to be read or written in EBCDIC and translated by the table specified in the ITBL or OTBL entry, respectively
OPTION	YES	X	X	X	Specifies an optional file
ORLP	YES			X	Specifies that a combined file is to be processed in an overlap mode
OTBL	symbol		X	X	Address of output translate table; required when MODE=TRANS is specified
OUBLKSZ	n			R	Specifies the length of IOAREA2 for a combined file
RECFORM**	FIXUNB	R	Y	R	For fixed-length records
	UNDEF*		Y		For undefined records
	VARUNB		Y		For variable-length records

Table 3—1. Summary of Keyword Parameters for the DTFCD Macroinstruction (Part 2 of 2)

Keyword	Specification	Input	Files		Remarks
			Output	Cmbnd	
RECSIZE†	(r)		X		For undefined records; general register (2—13) contains record size
	n	X	X		Specifies record size in bytes and is used with BLKSIZE for multi-sector I/O on diskette.
SAVAREA	YES	X	X	X	Specifies 72-byte register save area
STUB*	51	X			Stub card read for 51-column cards
	66	x			Stub card read for 66-column cards
TYPEFLE	INPUT	R			For input files
	OUTPUT		R		For output files
	CMBND			R	For combined read/punch file
WORKA	YES	X	X	X	Records are to be processed in work area. Omit IOREG

LEGEND:

R Required
X Optional
Y One option required
*Not appliable for diskette files
**Parameter may be changed on DD job control statement.
†Parameter may be changed on DD job control statement for diskette only.

3.4. IMPERATIVE MACROINSTRUCTIONS

There are five imperative macroinstructions available to you for processing punched card SAM files:

Macroinstruction	Use
OPEN	File control
GET	Record processing
PUT	Record processing
CNTRL	Output and combined file record control
CLOSE	File control

The following paragraphs provide you with a detailed description of these macroinstructions, and provide coding examples with explanations, when required, to clarify use.

OPEN

3.4.1. Open a Card SAM File (OPEN)

Function:

Before a file can be accessed by the logical IOCS, you must issue an OPEN macroinstruction. The transient routine called by the OPEN macroinstruction performs certain validation checks and initiates file processing. A check is made to determine that you have supplied all the necessary keyword parameters defining the file. The device allocation performed by the job control program is determined.

The actions performed by the OPEN transient routine depend on whether the file you are dealing with is an input or an output file. For input files, the first data record is not available to you until a GET macroinstruction is issued. For output files, no data is written; however, the data area is made available for use. Only one file per card reader should be open at any one time during execution of a program.

Format:

LABEL	ΔOPERATIONΔ	OPERAND
[name]	OPEN	{filename-1[,...,filename-n] / (1) / 1}

Positional Parameter 1:

filename
Is the label of the corresponding DTF macroinstruction in the program. The file name may have a maximum of seven characters; the maximum number of file names is 16.

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

Example:

LABEL (1)	ΔOPERATIONΔ (10)	OPERAND (16)	Δ
	OPEN	INPUT,OUTPUT,LISTING	

Enters the transient routines necessary to prepare the DTF macroinstructions whose labels are INPUT, OUTPUT, and LISTING. Checks that they are prepared to access these files with the next imperative macroinstruction (GET, PUT, etc.).

PUT

3.4.3. Output a Record (PUT)

Function:

The PUT macroinstruction delivers an output record to the logical IOCS in either the output area or a work area specified by you.

Data management delivers the records singly to your output peripheral device. A general register (2 through 13) must be supplied (by means of the IOREG keyword parameter) when a standby area (IOAREA2) is specified and when no work area is used. This register provides the logical IOCS with a place to put the address of the current output area. Records processed in an I/O area can be referenced directly by means of the name you have given to that area (IOAREA1). The output areas are not cleared after the current output data is sent to the device. You should exercise care to clear the area before use or to supply records, including blanks, which completely fill the output area to the logical IOCS to prevent spurious characters from appearing in the data.

When records are processed in a work area, the logical IOCS moves the record into the I/O area. This frees the work area for your subsequent processing.

When punching a record containing an odd number of characters, data management places a nonpunching character in the I/O area at the very end of the data you supplied.

Format:

LABEL	Δ OPERATION Δ	OPERAND
[name]	PUT	{ filename / (1) / 1 } [, { workarea / (0) / 0 }]

Positional Parameter 1:

filename
Is the label of the corresponding DTF macroinstruction in the program.

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

Positional Parameter 2:

workarea
Is the label of the work area from which the record may be obtained.

(0) or 0

Indicates that register 0 has been preloaded with the address of the work area.

If omitted, indicates you have chosen processing either by means of a register (IOREG keyword parameter) or by directly accessing the data relative to the name of the I/O area.

NOTE:

When the work area is specified, the keyword parameter WORKA=YES must be present in the DTF statement.

Example:

LABEL (1)	OPERATION (10)	OPERAND (16)
	PUT	(1), OUTWORK

Programming Considerations:

- Variable-Length, Unblocked Records

  You must determine the size of the output record and must insert the size at the beginning of the record before issuing the PUT macroinstruction. Record size includes the 4-byte record length field. You may not access the first four bytes, which are reserved for block size.

- Undefined Output Records

  RECSIZE=(r) must be specified; you must determine and place the record size in this register before issuing each PUT macroinstruction.

CNTRL

3.4.4. Controlling Stacker Selection on the Card Punch (CNTRL)

Function:

The CNTRL macroinstruction allows you to control stacker selection on the 0604 card punch for output or combined files. In processing a combined file, you may read a card, process the data read from a card, and then select an output stacker in accordance with the data on the same card. If you issue the CNTRL imperative macro in your program, you must specify the CONTROL keyword parameter in the DTFCD declarative macro (3.3).

The CNTRL macro is ignored if you issue it to a card file processed on the 0605 card punch because its small error stacker is not designed for selecting cards.

Format:

LABEL	Δ OPERATION Δ	OPERAND
[name]	CNTRL	{ filename / (1) / 1 } ,SS [{ ,1 / ,2 }]

Positional Parameter 1:

filename
Is the label of the DTFCD declarative macro defining the output or combined file.

(1) or 1
Indicates that you have preloaded register 1 with the address of the DTFCD declarative macro.

Positional Parameter 2:

SS
Specifies stacker selection on the 0604 card punch.

Positional Parameter 3:

1
Specifies selection of the normal stacker.

2
Specifies selection of the select (error) stacker. If the third positional parameter is omitted, specification of the select stacker is assumed. If the third positional parameter is specified, but is not specified as *1* or *2*, specification of the normal stacker is assumed; an error flag appears in your program listing.

3.4.4.1. Using the CNTRL Imperative Macro

You issue the CNTRL macro after any PUT or GET imperative macro that punches or reads the card you want to select, and before any PUT or GET macro that processes the following card. If you issue several CNTRL macros in succession, the last one you issue controls which hopper the card goes into the next time card motion occurs.

A look at the following schematic diagram of card flow through the 0604 card punch may be helpful in visualizing what the CNTRL macro does for you. In Figure 3—1, a card moves from left to right, from the input hopper, past the optional prepunch read station, to the punch station. It then passes into one of the two output hoppers: either the normal stacker or the select stacker, according to the position of the deflector. The 0604 card punch subsystem itself automatically deflects error cards into the select stacker; it is the deflector that can be controlled by the CNTRL macros issued in your program.

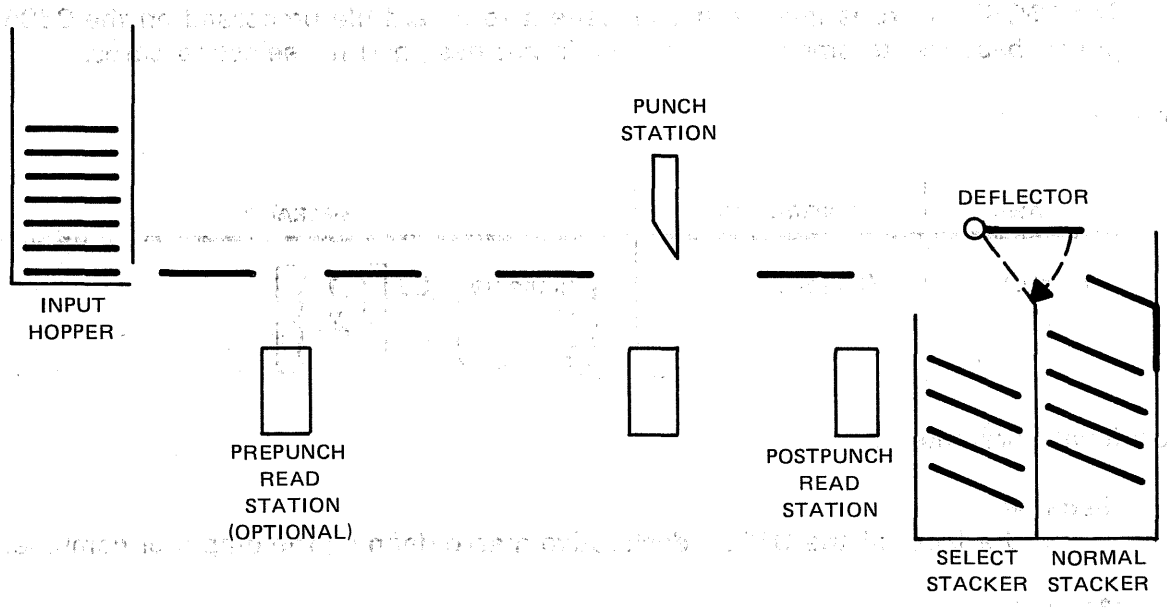


Figure 3—1. Schematic Diagram of Card Flow through 0604 Card Punch

The normal sequence is that a card in the postpunch station passes into the normal stacker when the following card enters the punch station; if you have set the deflector by issuing the CNTRL macro, however, it passes into the select stacker when the card following it moves (is fed or punched). Stacker selection for a card that has gone through the punch station thus takes effect when a macro is executed that moves the following card — a GET or PUT macro, depending on file type and your processing.

When a card file is opened, then, cards passing the punch station are sent to the normal output stacker until you issue:

- CNTRL filename,SS; or
- CNTRL filename,SS,2.

Then, the following imperative macro (GET or PUT) that causes card motion results in one card being placed in the select stacker. If you do not issue any further CNTRL macros, cards following the card that went into the select stacker are sent into the normal stacker. Thus, a CNTRL macro to send a card to the select stacker applies only to one card: To send 10 cards to the select stacker, you must issue 10 CNTRL macros, properly interleaved with PUT or GET macros. Note that the CNTRL macro causes no card motion itself.

For an output file, each PUT macro causes a card to be fed into the 0604 card punch and a card to be directed into an output hopper. You must issue the CNTRL macro after the PUT macro that punches the card you want selected and before your next PUT macro.

Example:

	LABEL (1)	OPERATION (10)	OPERAND (16)
		⋮	
		OPEN	PUNCH
1.		PUT	PUNCH
2.		PUT	PUNCH
		CNTRL	PUNCH,SS
3.		PUT	PUNCH
		CNTRL	PUNCH,SS,2
		CNTRL	PUNCH,SS,1
4.		PUT	PUNCH
		CNTRL	PUNCH,SS,2
5.		PUT	PUNCH
6.		PUT	PUNCH
7.		PUT	PUNCH
8.		CLOSE	PUNCH
		⋮	
9.	PUNCH	DTFCD	TYPEFILE=OUTPUT,
		⋮	

1. Punches card 1.
2. Punches card 2; card 1 is sent to the normal stacker.
3. Punches card 3; card 2 is sent to the select stacker.
4. Punches card 4; card 3 is sent to the normal stacker.
5. Punches card 5; card 4 is sent to the select stacker.

6. Punches card 6; card 5 is sent to the normal stacker.

7. Punches card 7; card 6 is sent to the normal stacker.

8. On closing the output file, card 7 is sent to the normal stacker.

9. PUNCH is the logical file name of the output card file being processed.

When you are processing combined files (TYPEFLE=CMBND), you have two processing modes, overlap and nonoverlap. The overlap mode is specified with the ORLP keyword, as you recall from 3.3; if you omit the ORLP keyword from the DTF of a combined card file, you process the file in the nonoverlap mode. The action of the CNTRL macro is slightly different in each of these modes; consider the overlap mode first.

For a combined card file with overlap, each GET or PUT macro advances a card, and the CNTRL macro applies to the card processed by the previous GET or PUT macro. The sequence of instructions in the following coding example processes a combined file deck named COMBO, in which each punched card is followed by one blank card. A punched card is read, data is processed, and some resulting data is punched into the blank card following it.

Example:

```
LABEL     ΔOPERATIONΔ          OPERAND                    Δ
1         10        16
                    |
          OPEN      COMBO
P1        EQU       *
          GET       COMBO
          CNTRL     COMBO,SS
          PUT       COMBO
          B         P1
ENDFL     CLOSE     COMBO
          EOJ
          :
COMBO     DTFCD     TYPEFLE=CMBND,
                    ORLP=YES,
                    EOFADDR=ENDFL,
                    :
```

The instruction sequence shown causes all the prepunched cards to be sent to the select stacker and all of the newly punched cards to be sent to the normal stacker. Processing continues until an end-of-data (/*) card is detected (2.3.2); at this point, the end-of-file routine ENDFL closes the file, and the job terminates.

On the other hand, consider the nonoverlap mode of processing a combined card file: A single card may be processed by a GET and a PUT macro. A CNTRL macro may be issued after any macro that processes a card. If a card is processed by both a GET and a PUT macro, CNTRL may be issued after either the GET or the PUT macro to control stacker selection of that card. If several CNTRL macros, which apply to a single card, are issued, the last CNTRL determines which stacker is selected.

The following coding example reads three cards from a combined card file named COMB02, processed in nonoverlap mode and containing no blank cards. It punches data on all three cards; the first is sent to the select stacker, and the other two cards are sent to the normal stacker.

Example:

```
LABEL    ΔOPERATIONΔ        OPERAND                    Δ
1        10        16
         OPEN      COMB02
         GET       COMB02
         PUT       COMB02
         CNTRL     COMB02,SS
         GET       COMB02
         PUT       COMB02
         GET       COMB02
         PUT       COMB02
         CLOSE     COMB02
         .
         .
         .
COMB02   DTFCD     TYPEFILE=CMBND,
         .
         .
```

CLOSE

3.4.5. Close a Card SAM File (CLOSE)

Function:

The CLOSE macroinstruction initiates the termination procedures for your card SAM file. When all the data in a file has been processed, a CLOSE macroinstruction should be issued.

Format:

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

Positional Parameter 1:

filename
Is the label of the corresponding DTF macroinstruction in your program. Filename may contain a maximum of seven characters; the maximum number of filenames is 16.

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

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

Example:

LABEL (1)	ΔOPERATIONΔ (10)	OPERAND (16)	Δ
ENDREP	CLOSE	INPUT	

Enters the transient routine which closes the file described in the DTF macroinstruction whose label is INPUT.

3.5. ERROR AND EXCEPTION HANDLING

3.5.1. FilenameC

When certain errors or exceptions to file processing performance are detected by OS/3 data management, it will make appropriate entries in specific fields of the DTF file table, which your program may address in order to learn of these conditions and take the proper course of action on regaining control. One such field is *filenameC*, a 1-byte field which you may access by concatenating the character C to your 7-character logical filename and using the *test-under-mask* (TM) instruction.

Refer to Appendix B for the meaning of the bits in *filenameC* of the DTFCD file table which are set to binary 1 by OS/3 data management for certain error and exception conditions.

3.5.2. FilenameS

When you have specified CRDERR=RETRY on a card punch file DTFCD and six successive attempts to punch a card have failed, OS/3 data management sets the *hardware error* bit in *filenameC* (see Appendix B) and also places the image of the card which is in error in *filenameS* of the DTFCD file table. *FilenameS, which you may address in the same way as filenameC*, is an 80-byte field for all modes except the binary (image) mode, and a 160-byte field for that mode. *FilenameS* does not contain the error card image if the file is a combined file. Software punch retry applies to the 0604 card punch. For the 0605 (integrated) punch, the operator can repunch erroneous cards.

3.6. SAMPLE PROGRAMS

The following examples have been constructed to illustrate typical uses of card input, output, and combined files in BAL programs. They also provide examples of the OS/3 job control statements you need to implement your programs.

```
LABEL    OPERATION   OPERAND                  COMMENTS
1        10          16                       72                       80
// JOB   EXAMPLE
         .
         .           JOB STREAM TO EXECUTE ASSEMBLER
         .
         .
/$
INPT1    START       0
*
*  INPT1    EXAMPLE OF THE USE OF A DATA MANAGEMENT CARD INPUT FILE
*
         BALR        9,0                      LOAD COVER REGISTER
         USING       *,9
         LA          13,SAVE                  REG SAVE AREA ADDR TO R13
         OPEN        IN1                      OPEN CARD FILE IN1
B1       GET         IN1,WORK                 READ 1 CARD
         .
         .           PROCESS DATA READ INTO WORK AREA
         .
         .
         B           B1
EOF      CLOSE       IN1                      END OF FILE, CLOSE CARD FILE IN1
         EOJ                                  END OF JOB
*
*  ERROR ROUTINE
*
```

```
LABEL     OPERATION   OPERAND                    COMMENTS
1         10          16                         72        80
ERR       TM          INIC,X'80'                 DATA TRUNCATED BIT SET?
          BOR         14                         BRANCH IF SET BACK INTO PROGRAM
*                                                &CONTINUE PROCESSING
          CANCEL                                 OTHERWISE CANCEL PROGRAM
*
* DATA STORAGE AREA
*
SAVE      DS          18F                        REGISTER SAVE AREA
IO1       DS          CL80                       I/O AREA1
WORK      DS          CL80                       WORK AREA
*
* MACRO CALL FOR CARD DM SYSTEM
*
IN1       DTFCD       BLKSIZE=80,EOFADDR=EOF,ERROR=ERR,IOAREA1=IO1,MODE=STD,  X
                      TYPEFLE=INPUT,WORKA=YES
          END         INPT1
/*
                      } JOB STREAM TO EXECUTE LINKAGE EDITOR
// DVC 30             // LFD IN1                 ASSIGNMENT OF CARD READER
                      } JOB STREAM TO EXECUTE PROGRAM
/&
// FIN
```

```
LABEL     OPERATION     OPERAND          COMMENTS
1         10            16               72                    80
// JOB    EXAMPLE
          .
          .             JOB STREAM TO EXECUTE ASSEMBLER
          .
          .
/$
OUPT1     START         0
*
*  OUPT1    EXAMPLE OF THE USE OF A DATA MANAGEMENT CARD OUTPUT FILE
*
          BALR          9,0                LOAD COVER REGISTER
          USING         *,9
          LA            13,SAVE            REG SAVE AREA ADDR TO R13
          OPEN          OUT                OPEN CARD FILE OUT
B1        .
          .             PREPARE PUNCH DATA INTO WORK AREA
          .
          PUT           OUT,WORK           PUNCH 1 CARD
          CLC           WORK(12),=C'/*'    LAST DATA PROCESSED?
          BNE           B1                 IF NO,REPEAT PROCESS
          CLOSE         OUT                IF YES,CLOSE CARD FILE OUT
          EOJ                              END OF JOB
*
*  ERROR ROUTINE
*
```

```
LABEL    OPERATION   OPERAND                          COMMENTS
1        10          16                               72        80
ERR      CANCEL                                       CANCEL PROGRAM
*
* DATA STORAGE AREA
*
SAVE     DS          18F                              REGISTER SAVE AREA
IOA1     DS          CL80                             I/O AREA 1
IOA2     DS          CL80                             I/O AREA 2
WORK     DS          CL80                             WORK AREA
*
* MACRO CALL FOR CARD DM SYSTEM
*
OUT      DTFCD       BLKSIZE=80,ERROR=ERR,IOAREA1=IOA1,IOAREA2=IOA2,MODE=STD,X
                     TYPEFILE=OUTPUT,CRDERR=RETRY,RECFORM=FIXUNB,WORKA=YES
         END         OUPT1
/*
                     JOB STREAM TO EXECUTE LINKAGE EDITOR
// DVC 40            // LFD OUT
                     JOB STREAM TO EXECUTE PROGRAM
/&
// FIN
```

```
LABEL     OPERATION     OPERAND                    COMMENTS
1         10            16                         72                    80
// JOB    EXAMPLE
          .
          .             JOB STREAM TO EXECUTE ASSEMBLER
          .
          .
/$
COMB1     START         0
*
* COMB1      EXAMPLE OF THE USE OF A DATA MANAGEMENT CARD COMBINED FILE
*
          BALR          9,0                        LOAD COVER REGISTER
          USING         *,9
          OPEN          COMBINE                    OPEN CARD FILE COMBINE
B1        GET           COMBINE                    READ 1 CARD
          .
          .             PROCESS DATA READ INTO I/O AREA 1
          .
          .
          .             PREPARE PUNCH DATA INTO I/O AREA 2
          .
          PUT           COMBINE                    OVERLAP PUNCH
          B             B1
EOF       CLOSE         COMBINE
          EOJ
*
```

```
LABEL     OPERATION   OPERAND                                   COMMENTS
1         10          16                                        72        80
* ERROR   ROUTINE
*
ERR       CANCEL                                                CANCEL PROGRAM
*
* DATA STORAGE AREA
*
SAVE      DS          18F                                       REGISTER SAVE AREA
IOA1      DS          CL80                                      I/O AREA 1
IOA2      DS          CL80                                      I/O AREA 2
*
* MACRO   CALL FOR CARD DM SYSTEM
*
COMBINE   DTFCD       BLKSIZE=80,EOFADDR=EOF,ERROR=ERR,IOAREA1=IOA1,   X
                      IOAREA2=IOA2,MODE=STD,OPTION 2ON=YES,QUBLKSZ=80, X
                      TYPEFLE=CMBND,SAVAREA=SAVE
          END         COMB1
/*
                      } JOB STREAM TO EXECUTE LINKAGE EDITOR
// DVC 40  //        LFD COMBINE
                      }
                      } JOB STREAM TO EXECUTE PROGRAM
/&
// FIN
```

4. Diskette Formats and File Conventions

4.1. GENERAL

This section describes the data formats and file conventions that apply to the 8413 diskette subsystems files supported by OS/3. The 8413 diskette is a rapid replacement medium for card processing devices and provides multifile volume and multivolume file exchange capabilities.

4.2. FILE ORGANIZATION

The 8413 diskette is a single-sided, fixed-sector storage medium used for sequential file processing as a substitute for punched card equipment. The diskette subsystem handles single or multivolume input, output and combined files. A maximum of 19 files is allowed on a single diskette volume. A summary of the 8413 diskette volume characteristics follows:

Tracks	77 (0—76)
Tracks (usable)	73 (1—73)
Sectors per track	26
Sectors (records) per volume	1898
Sector size	128 bytes
Files per volume	19 maximum
Number of volumes	152 maximum per file

Figure 4—1 illustrates the track and sector organization of an 8413 diskette volume.

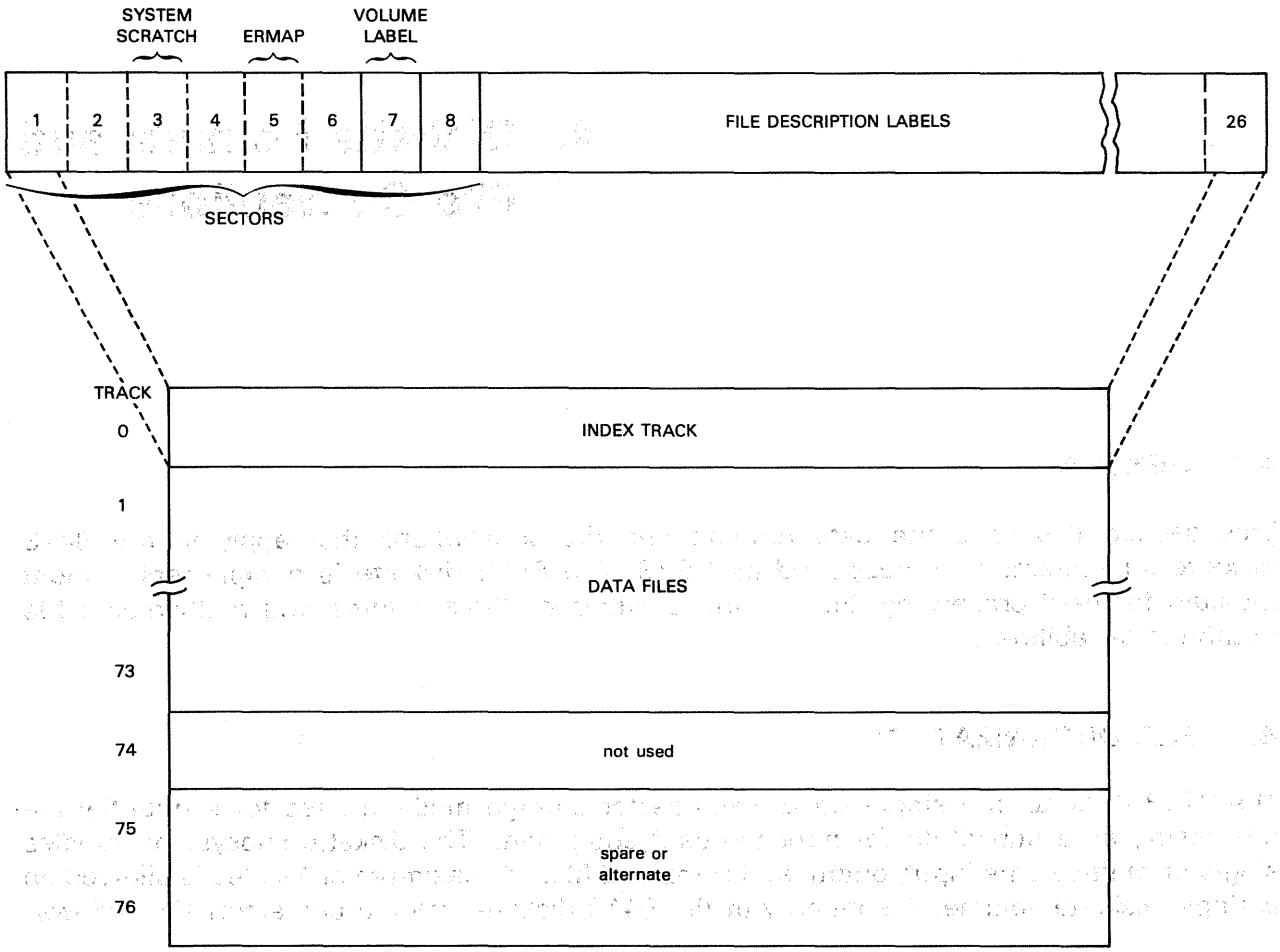
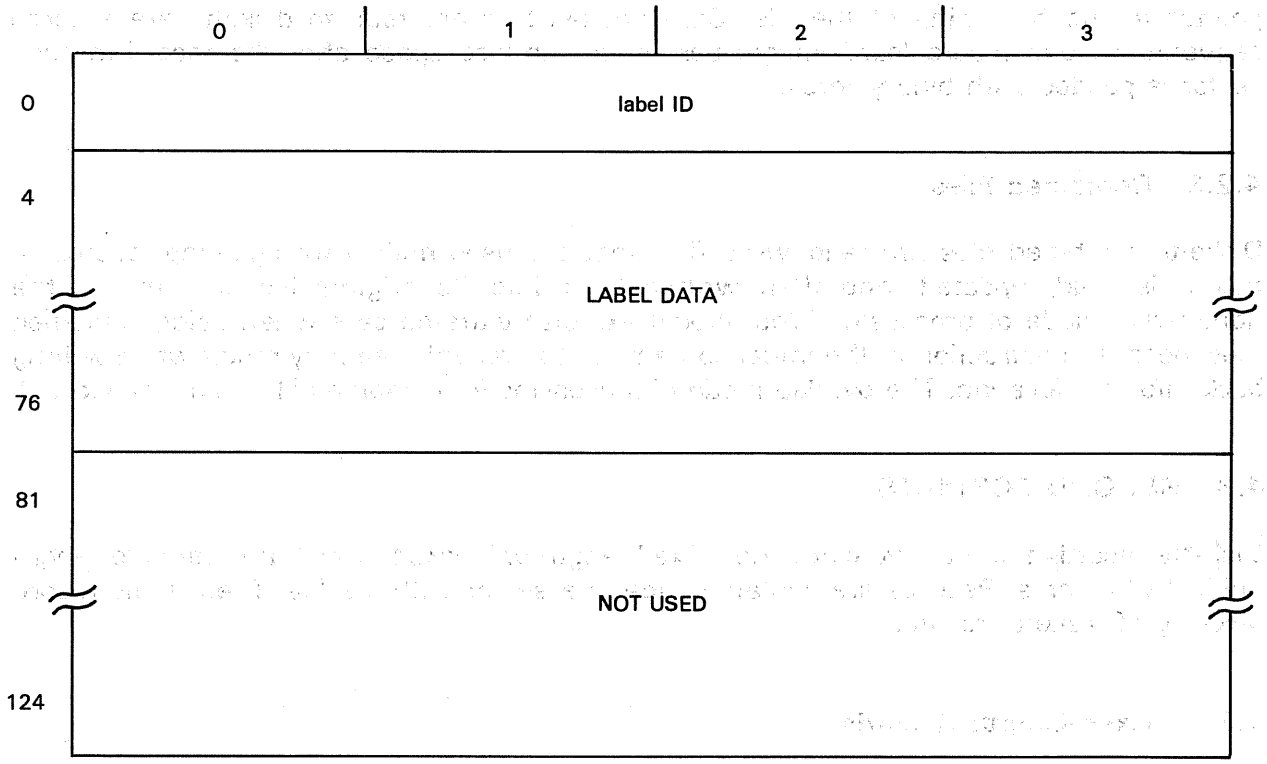


Figure 4—1. Typical Organization of a Diskette Volume

File description labels are written on the first track (track 0) of each volume of a diskette DTFCD file by data management. The maximum number of files that can be written to a volume is 19. The simple 128-byte diskette file label format is:



NOTE:

Details on the diskette file description label are presented in D.5.

4.2.1. Diskette Input Files

Diskette files can be contained on one volume or can span several volumes (multivolume file). Information on a diskette volume is organized into two areas; the index track (track 0) and the data files (tracks 1 through 73). Track 74 is not used; tracks 75 and 76 are alternates or spares (See Figure 4—1).

The index track (track 0) contains a volume label (VOL1) in sector 7. Sectors 8 through 26 are used for the file description labels. One file can be described in each sector; therefore, a maximum of 19 file description labels can be entered in the track index.

The data portion of the diskette files contain punch card images (EBCDIC) with one record to each diskette sector. Each sector is 128 bytes long, and any unused space in the sector after the record is hardware padded.

All diskette input files are read-only sequential files. Multivolume files require that the volumes be mounted in the proper sequence; standard mount messages provide prompting to ensure that the volumes are mounted in the correct order.

4.2.2. Diskette Output Files

Diskette output files are read/write sequential files allocated on a sector basis. All files must reside in a contiguous area. When the file description label is written, it includes a pointer to the beginning of the file. Card images that are read to diskette are entered sequentially, one record (card image) per sector; unused space after the record in each sector is padded with binary zeros.

4.2.3. Combined Files

Diskette combined files are read/write files that are used mainly for updating records. A record is read, updated, and then written back into its original location; this is the nonoverlap mode of processing. You should exercise extreme care when using combined files, because destruction of the initial contents of the record read may result when writing back into that location. The overlap mode of processing for combined files is not supported.

4.3. RECORD FORMATS

Diskette records fall into two categories: fixed-length unblocked records and variable-length unblocked records. Records are contained one to a sector within a file. There is no record blocking of diskette records.

4.3.1. Fixed-Length Records

Fixed-length diskette records are all of equal length for a given file. Diskette records are generally the length for a given card type image (51, 66, 80, or 96); however, the records can be any length from 1 to 128 bytes. Figure 4—2 illustrates the fixed-length record characteristics.

4.3.2. Variable-Length Records

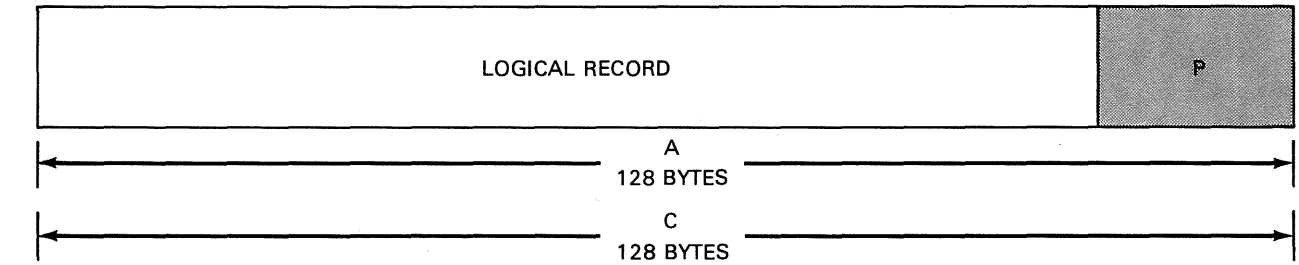
When you use variable-length records, data management preempts the first four bytes of every block (in this case, record) for use as a record descriptor word (Figure 4—2). Data management calculates the length of the record and inserts this for you in the first two bytes; the other two bytes are used by data management.

Data management, again reserves the final two bytes of the record descriptor word (RDW) for its own use, but the first two bytes must contain the length of the record of which the RDW is a part.

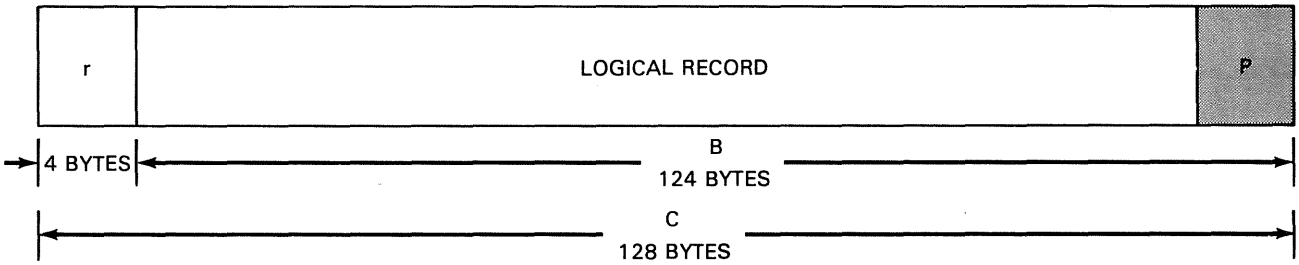
When you specify that your records are variable and unblocked, data management will write out one block for each logical record you submit, regardless of the amount of available space remaining in the I/O area.

You must not use the RECSIZE keyword parameter in your DTF for a file containing variable-length records, because data management expects to find the record size in the first two bytes of RDW.

FIXED-LENGTH



VARIABLE-LENGTH



LEGEND:

r Record descriptor word (RDW)
A Data record length plus padding to fill out the sector
B Variable record length
C I/O area layout
P Padding

NOTES:

1. For input files, data management passes to the user the record length and data portion of the record.

2. BLKSIZE=n specification on DTF macro and file label block length determines the size of record processed during OPEN processing. The maximum block size of multisector I/O is 1024 bytes.

3. Unused sector space is hardware padded.

4. Under multisector I/O, user IOAREA contains multiple records in either fixed unblocked or variable unblocked formats.

Figure 4—2. Diskette File Record Formats

5. Function and Operation of Diskette SAM

5.1. GENERAL

This section contains a brief description of the data management modules that apply to SAM operation for input files, output files, and combined files used with diskette operation. Following the functional description is a detailed explanation of the declarative macroinstructions that define the three types of files. This section concludes with the imperative macroinstructions that initiate, conduct, and terminate file processing.

NOTE:

The 8413 diskette processor does not handle compressed code translation.

5.2. FUNCTIONAL DESCRIPTION

5.2.1. Input Record Processing

Diskette input files, like punched card input files (2.2.1), use the fixed unblocked record format (RECFORM=FIXUNB). Diskette records range from a fixed length of 1 to 128 bytes per sector. In addition, diskette input files can be in variable unblocked record format (RECFORM=VARUNB).

Data management accesses diskette input files in read mode only. Once data management locates a diskette file label at open time, it saves the file label address and certain fields of information from the label such as file extent boundaries, i.e. beginning of extent (BOE), end of data (EOD), and end of extent (EOE). Data management then compares file label information with DTF specifications to determine if the file should be processed under single-sector or multisector I/O. It is the user's responsibility here to provide I/O areas of adequate size to handle the block length specified.

5.2.2. Output Record Processing

Data management accesses an opened diskette output file in both read and write modes. Using the file-id on the job control // LBL statement, data management searches the index track to locate the corresponding file label and saves the file label address to be used at close time. Files can be extended or overwritten on output only if the expiration date has been surpassed or if INIT is specified on the // LFD statement.

If specified on the // LBL statement for the output file, the new creation date is then inserted in the file label; otherwise, the system date is used as the creation date.

If INIT is specified on the // LFD job control statement for an output file, the file expiration date is ignored and the file is overwritten. If EXTEND is specified on the // LFD statement, the expiration date is checked and if still valid, the file is extended. If neither INIT or EXTEND is specified, a normal check of expiration date occurs.

If no major file errors occurred to that point, data management writes the label back to the index track in its original sector location, positioning occurs if EXTEND mode was specified, and data management marks the DTF as opened and passes control to the next instruction.

Data management writes output files via the PUT imperative macro either by single-sector or multisector I/O (determined by BLKSIZE or RECSIZE DTF specifications). When data management closes output files, it writes all necessary buffers to the diskette to avoid loss of user records, and updates the EOD field of the file label by reading the label, updating it, and writing it to its original sector location on the index track. Finally, data management resets indicators and fields in the DTF and marks the DTF closed.

5.2.3. Combined File Record Processing

Data management handles combined files (files capable of GET/PUT functions) at open time the same way it handles input files (5.2.1).

Likewise, during close operations, data management handles combined files as output files (5.2.2) with one exception. If the current EOD is less than the original EOD, i.e., partial update occurred, data management does not update the EOD field on the file label. If the current EOD is greater than the original EOD, i.e., file extension occurred, data management updates the EOD field in the file label.

Data management processes GET and PUT diskette operations for combined files under single-sector I/O. The IOAREA1 receives the input record via the GET macro. The user must move the record to IOAREA2 and then update it. The contents of IOAREA1 are superimposed on the updated record in IOAREA2. If an invalid character results, the original character in IOAREA1 will be substituted in IOAREA2. From there, the PUT macro writes it to the diskette at the sector from which the original record was read. Data management repositions back one sector before each write so that the PUT writes directly to the record's original location. When a series of PUT macros occurs, however, no backward sector repositioning occurs after the second and all succeeding PUT macros. The user must take care in moving updated records to avoid loss of existing data.

5.2.4. Multisector I/O

Multisector I/O is allowed for the diskette up to a maximum I/O byte count of 1024. This means that up to 8 full 128-byte sectors (records) can be read or written with one physical I/O. Many more sectors can be handled, if record size is less than 128 bytes. For example, if record size is 80, the maximum blocksize that can evenly process multiple sectors is 960 bytes and the number of sectors accessed in a single physical I/O is 12.

To process smaller records using multisector I/O, the BLKSIZE and RECSIZE parameters of the DTFCD declarative macro must be specified with the blocksize value being an integral multiple of the record size. To determine the actual number of sectors to the processed, divide your BLKSIZE length DTF specifications by block length field in the file label (positions 23 through 27 in the file label sector).

The result must be an integral number of sectors. There is no remainder. Records are in blocked format in I/O areas; therefore, to facilitate record by record processing, you must specifiy either the IOREG or WORKA parameters on your DTFCD macro.

In addition, the IOAREA buffer space allocation must be increased to equal the new larger blocksize specification in the DTFCD macro and reprogramming and reassembling of existing card file programs is necessary to use diskette multisector I/O.

In multisector processing, the initial logical GET brings in a block of sectors and either points to a record or moves a record to your work area. When all records from a block of sectors have been processed, another physical multisector I/O occurs and processing continues until the file is exhausted and EOD is reached. Control then passes to your end of file routine (EOFADDR=symbol).

5.2.5. Specifying 8413 Diskette Use

The following steps are required to use the 8413 diskette:

- The supervisor which supports the diskette must be generated. See the system installation user guide/programmer reference, UP-8074 (current version).

- The DSKPRP system utility routine and diskette space management must be applied to the diskette to initialize the allocate file space before user program execution. See the system service programs (SSP) user guide, UP-8062 (current version).

- The appropriate job control statements must be provided to enable diskette recognition and scheduling. The // DVC statement specifies logical unit numbers 130, 131, 132, or 133 for diskette. The ALT option of this statement allows you to specify multivolume processing (using two drives). The // VOL statement supplies the diskette volume serial number. On a first run, the // EXT statement allocates sectors. The // LBL statement identifies the file (this name should match positions 5—12 on the file label of the diskette).

Only the first eight positions of the field are used for the file name. The // LBL statement can also specify the file creation date, and file expiration date. Finally, the // LFD statement specifies the name given for the file description (DTFCD).

For further detail, see the job control user guide, UP-8065 (current version).

5.2.6. Diskette Limitations

The following limitations exist for the 8413 diskette:

- All diskette files are created and retrieved sequentially.
- Data management requires that each diskette file be opened via an OPEN imperative macro before accessing the file and closed via a CLOSE macro when file processing is finished. Data management also recognizes a virtual device on an OPEN macro.
- The CNTRL imperative macro is ignored when issued to a diskette file.
- If an error occurs during file processing in output mode, and control passes to the user's error routine, the user must close the file in error to permit data management to update the EOD pointer in the file label.
- Maximum block size allowable with multisector I/O is 1024 bytes and file processing is limited to tracks 1 through 73.
- Within the same job step, only one logical file (DTF) may access a diskette volume.
- Data management cannot process, in a normal GET/PUT sequence, combined files that contain logically deleted data records containing 'D' in the first position of the record.
- Data management PUT operations do not use multisector processing if spooling out. The output writer provides this feature.
- All data management mount messages are suppressed in a spooling environment.
- If spooling is in operation, a CLOSE macro does not attempt to access the diskette.

DTFCD (Diskette)

5.3. DEFINE A SAM DISKETTE FILE (DTFCD)

Function:

The DTFCD declarative macro used to define punched card files is also used to define diskette files (3.3). Except for the undefined record format specification (RECFORM=UNDEF) which generates an invalid DTF field message (DM61; see Table B—1), if issued for a diskette, the following DTFCD parameter specifications do not apply to diskette files and are ignored if specified for diskette files:

AUE=YES

CONTROL=YES

CRDERR=RETRY

MODE=BINARY

RECFORM=UNDEF

STUB=51

STUB=66

The format of the DTFCD macroinstruction as it applies to diskette files follows:

Format:

LABEL	Δ OPERATION Δ	OPERAND
filename	**DTFCD**	**[ASCII=YES]** **[,BLKSIZE=n]** **[,EOFADDR=symbol]** **[,ERROR=symbol]** **IOAREA1=symbol** **[,IOAREA2=symbol]** **[,IOREG= (r)]** **[,ITBL=symbol]** **[,MODE= { CC / STD / TRANS }]** **[,OPTION=YES]** **[,ORLP=YES]** **[,OTBL=symbol]** **[,OUBLKSZ=n]** **[,RECFORM= { FIXUNB / VARUNB }]**

LABEL	Δ OPERATION Δ	OPERAND
	DTFCD (cont)	[,RECSIZE= {(r) / n}] [,SAVAREA=symbol] [,TYPEFLE= {INPUT / OUTPUT / CMBND}] [,WORKA=YES]

For a complete description and summary of each keyword parameter, refer to 3.3 and Table 3—1.

5.4. IMPERATIVE MACROINSTRUCTIONS

There are four imperative macroinstructions available to you for processing diskette SAM files:

Macro instruction	Use
OPEN	File control
GET	Record processing
PUT	Record processing
CLOSE	File control

The following paragraphs describe these macroinstructions in detail and provide coding examples with explanations.

OPEN

5.4.1. Open a Diskette SAM File (OPEN)

Function:

The OPEN macroinstruction is used to open a file for processing. The transient routine called by the OPEN macroinstruction makes certain validation checks and then proceeds to access the diskette file. The OPEN transient routine accesses input and combined files in read only mode (5.2.1). It accesses output files in read and write modes (5.2.2).

Format:

LABEL	ΔOPERATIONΔ	OPERAND
[name]	OPEN	{filename-1... [,filename-n] / (1)}

Positional Parameter 1:

filename
Is the label of the corresponding DTF macroinstruction in the program. The file name may have a maximum of seven characters; the maximum number of file names is 16.

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

Example:

LABEL (1)	ΔOPERATIONΔ (10)	OPERAND (16) Δ
	OPEN	INPUT,OUTPUT

Enters the transient routines necessary to prepare the DTF macroinstructions whose labels are INPUT and OUTPUT. Checks that they are prepared to access these files with the next imperative macroinstruction (GET, PUT, etc.).

GET

5.4.2. Retrieve Next Logical Record (GET)

Function:

The GET macroinstruction makes the next logical record in the input diskette file available to you. The data is accessible either in an I/O area or in a work area you have specified. Data records must be in fixed unblocked or variable unblocked format (4.3).

If you specify only one I/O area and require single-sector I/O processing, you access records relative to the name of the I/O area. Otherwise, you must specify a register (IOREG keyword parameter) used by the card processor to supply the starting address of the current record or must specify a work area in the DTF declarative macro, WORKA=YES.

If you use multisector processing, to determine the number of sectors read by one physical I/O as a result of a GET macro, divide your DTF blocksize length by the block length field in the file label (positions 22—26). The result must be an integral number of sectors. There is no remainder. The GET macro reads a block of sectors and points to a record or moves it to a work area until reaching end of data (EOD). Data management then passes control to your EOFADDR=symbol routine indicated on the DTF macro.

Format:

LABEL	Δ OPERATION Δ	OPERAND
[name]	GET	{filename / (1)} [, {workarea / (0)}]

Positional Parameter 1:

filename
Is the label of the corresponding DTF macroinstruction in the program.

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

Positional Parameter 2:

workarea
Is the label of an area into which the current record is moved for processing.

3.4.2. Retrieve Next Logical Record (GET)

Function:

The GET macroinstruction makes the next logical record in an input file available to you. The data is accessible either in the I/O area or in a work area you have specified. The macroinstruction is used for all record types.

If you specify only one I/O area, you may directly access data relative to the name of the I/O area. Otherwise, you must specify a register (through the IOREG keyword parameter) to be used by the logical IOCS to give the starting address of the current record, or you must specify a work area in the declarative macroinstruction. More than one work area may be employed, since the address of the area is specified to the logical IOCS with each GET macroinstruction. Each GET macroinstruction may specify a different work area, if necesary.

Format:

LABEL	Δ OPERATION Δ	OPERAND
[name]	GET	{ filename / (1) / 1 } [, { workarea / (0) / 0 }]

Positional Parameter 1:

filename
Is the label of the corresponding DTF macroinstruction in the program.

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

Positional Parameter 2:

workarea
Is the label of an area into which the current record is moved for processing.

(0) or 0
Indicates that register 0 has been preloaded with the address of a work area.

If omitted, indicates the user has chosen processing either by means of a register (IOREG keyword parameter) or by directly accessing the data relative to the name of the I/O area.

NOTE:

When a work area is specified, the keyword WORKA=YES must also be specified in the DTF statement.

Example:

LABEL (1)	OPERATION (10)	OPERAND (16)
HERE	GET	INPUT,INWORK

Places the next record of the file described in the DTF macroinstruction, whose label is INPUT, into the area whose label is INWORK. The optional label HERE may be used to reference this point in the program.

(0)

Indicates that register 0 has been preloaded with the address of a work area.

If omitted, indicates the user has chosen processing either by means of a register (IOREG keyword parameter) or by directly accessing the data relative to the name of the I/O area.

NOTE:

When workarea is specified, the keyword WORKA=YES must also be specified in the DTF statement.

Example:

LABEL (1)	OPERATION (10)	OPERAND (16)
HERE	GET	INPUT,INWORK

Places the next record of the file described in the DTF macroinstruction, whose label is INPUT, into the area whose label is INWORK. The optional label HERE may be used to reference this point in the program.

PUT

5.4.3. Writing a Diskette Record (PUT)

Function:

The PUT macroinstruction delivers an output record to the card processor. In single-sector processing, each PUT macro writes a record to diskette. In multisector processing, you use the IOREG or WORKA specifications in the DTF and the PUT macro to control the release and writing of individual records from a block of sectors held in the output buffer to the output file on the diskette.

To prevent occurrence of unwanted information in the data, you must be careful to clear output record buffer areas before each use or, to supply complete records including blanks on each logical PUT.

Format:

LABEL	Δ OPERATION Δ	OPERAND
[name]	PUT	{filename / (1)} [, {workarea / (0)}]

Positional Parameter 1:

filename
Is the label of the corresponding DTF macroinstruction in the program.

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

Positional Parameter 2:

workarea
Is the label of the work area from which the record may be obtained.

(0)
Indicates that register 0 has been preloaded with the address of the work area.

If omitted, indicates you have chosen processing either by means of a register (IOREG keyword parameter) or by directly accessing the data relative to the name of the I/O area.

NOTE:

When the work area is specified, the keyword parameter WORKA=YES must be present in the DTF statement.

Example:

LABEL (1)	ΔOPERATIONΔ (10)	OPERAND (16)	Δ
	PUT	(1), OUTWORK	

Programming Considerations:

- Variable-Length, Unblocked Records

  You must determine the size of the output record and must insert the size at the beginning of the record before issuing the PUT macroinstruction. Record size includes the 4-type record length field. You may not access the first four bytes, which are reserved for block size.

CLOSE

5.4.4. Closing a Diskette File (CLOSE)

Function:

The CLOSE macroinstruction transfers control to a data management CLOSE transient routine which validates devices. If devices are diskette, a new diskette close transient receives control and determines the close processing required according to file type. It marks the DTF of an input file closed and resets indicators and fields in the DTF where applicable (2.4.3). For output files, it writes all necessary buffers to diskette, updates the EOD indicator in the file label, and resets indicators in the DTF before closing the file (2.4.4). The diskette close transient closes combined files like output files except when updating or not updating the EOD field for partial updates of files or extended writes to files (2.4.5).

Format:

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

Positional Parameter 1:

filename
Is the label of the corresponding DTF macroinstruction in your program. Filename may contain a maximum of seven characters; the maximum number of filenames is 16.

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

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

Example:

LABEL (1)	ΔOPERATIONΔ (10)	OPERAND (16) Δ
ENDREP	CLOSE	INPUT

Enters the transient routine which closes the file described in the DTF macroinstruction whose label is INPUT.

6. Printer Formats and File Conventions

6.1. GENERAL

The section describes the data formats and file conventions that apply to printer subsystem files supported by OS/3. The SPERRY UNIVAC 0773 Printer Subsystem, an integrated printer, is intended primarily for use with OS/3. However, the SPERRY UNIVAC 0768, 0770, and 0776 Printer Subsystems are also supported by OS/3.

A number of terms, used in what follows, are explained here:

line spacing
Advancing the paper (or forms) to be printed under the control of a line counter, i.e. a *specified number* of lines.

line skipping
Advancing the paper to a line on the form that is *specified by a code* placed in the vertical format buffer (VFB) by the user (or by a punch made by the user in the forms control loop).

paper advance
Vertical movement of the form or paper in the printer either after printing or without printing.

vertical format buffer
A buffer in the 0773, 0770, 0768, and 0776 printers. The buffer contains a location for each line on a form. A code may be placed in the location that corresponds to a particular line. Your program can then advance the form to that line by issuing a skip command and specifying the appropriate code. The paper tape loop on the 0768 printer is used in conjunction with the VFB.

load code buffer
A buffer located in the printer that allows the specification of any 8-bit code for any graphical symbol on the print band or drum. Thus, you can load the EBCDIC codes for the graphical characters on the print band into the load code buffer in the proper sequence and then print EBCDIC data.

6.1.1. 0773 Printer Subsystem

The 0773 printer has a standard print line of 120 print columns. You can expand this to 144 print columns through available hardware options. Line spacing (6 or 8 lines per inch) is accomplished through a switch on the printer. Paper advance (up to 15 lines) is controlled by the VFB and can be accomplished after printing or without printing.

6.1.2. 0770 Printer Subsystem

The 0770 printer allows you to use print lines of up to 160 print columns. The line spacing (6 or 8 lines per inch) and paper advance (up to 15 lines) are accomplished through the VFB. As with the 0773 printer, paper advance can be accomplished without printing or after printing.

6.1.3. 0768 Printer Subsystem

The 0768 printer allows you to print lines of up to 132 print columns. Line spacing (6 or 8 lines per inch) is controlled through a punched paper tape loop (forms control loop), and paper advance (up to 15 lines) can be accomplished without printing or after printing.

6.1.4. 0776 Printer Subsystem

The 0776 printer allows you to print lines of up to 136 print columns. Line spacing (6 or 8 lines per inch) and paper advance (up to 15 lines) are accomplished through the VFB. As with the 0773 printer, paper advance can be accomplished without printing or after printing.

6.1.5. 0778 Printer Subsystem

The 0778 printer has a standard print line of 120 print columns. You can expand this optionally to 132 print columns. Line spacing (6 or 8 lines per inch) is accomplished through a switch on the printer. Paper advance (up to 15 lines) is controlled by the VFB and can be accomplished after printing or without printing.

6.2. FILE ORGANIZATION

A print file can be best described as a collection of related data that is output to a printer device, one line at a time (band or drum printer). Line printers assemble the contents of a complete line (including blank spaces) before actual printing occurs.

Line printers are provided with the 90/30 system, and operate through OS/3. Therefore, not only are you responsible for organizing the data you want printed within each line, but you must also consider the vertical separation between lines and pages.

Print files described in this section fall into three categories:

- general text;
- tabular data; and
- data printed on forms.

6.2.1. Text

The simplest printer file to understand and use is probably one which consists of plain text. For example, assume that each input record is punched card and each output record is formed in the I/O area or in a work area you have specified. The records are output to the printer buffer by the physical IOCS. Each time that the printer buffer is full, a print command is issued and the line is printed. Figure 6—1 is a typical example of text output; the annotations point out the record where the home paper instruction should be issued and the home paper position necessary to begin printing the lower portion of the text at the top of a new page.

```
LINE TRUNCATED (BIT 0)

VERTICAL
 SPACING
          FIXED LENGTH, UNBLOCKED RECORDS: THE LINE TRUNCATED BIT IS SET
          DURING OPEN PROCESSING IF THE USER HAS SPECIFIED A BLOCK SIZE
          WHICH INDICATES A PRINT LINE LONGER THAN THE MAXIMUM PRINT LINE
6 LINES/INCH WHICH CAN BE PRINTED ON THE ASSIGNED PRINTER. OPEN PROCESSING IS
          COMPLETED AND THEN A BRANCH TO THE USER ERROR ROUTINE OCCURS. IF
          THERE IS NO ERROR ROUTINE, THE OPEN ROUTINE RETURNS TO THE USER
          AT NORMAL RETURN POINT. IF THE USER CONTINUES PROCESSING EACH
          PRINT LINE WILL CONTAIN THE MAXIMUM NUMBER OF PRINT
HOME PAPER
INSTRUCTION →
                                   HOME PAPER                          PAGE
                                   POSITION                            NUMBER
90/30 PRINTER SYSTEM/USER INTERFACE  ----↓----                          ↓
                                                                        16
 ↑
PAGE
HEADING
          POSITIONS WHICH CAN BE PRINTED ON THE ASSIGNED PRINTER.

          VARIABLE LENGTH, UNBLOCKED RECORDS AND DEFINED RECORDS:
          IF THE USER TRIES TO PRINT A LINE LONGER THAN THE MAXIMUM LINE
          LENGTH INDICATED BY THE BLOCK SIZE SPECIFICATION, A LINE EQUAL
          IN LENGTH TO THE MAXIMUM INDICATED BY BLOCK SIZE WILL BE PRINTED,
          AFTER THE LINE IS PRINTED, THE LINE TRUNCATED BIT WILL BE SET AND
          A BRANCH TO THE USER ERROR EXIT OCCURS. IF THERE IS NO ERROR EXIT,
          THE PUT ROUTINE RETURNS TO THE INSTRUCTION AFTER THE PUT MACRO
          INSTRUCTION. NOTE: THE OPEN ROUTINE ADJUSTS BLOCK SIZE IF A PRINT
          LINE LONGER THAN THE MAXIMUM PRINT LINE ON THE ASSIGNED PRINTER
          IS INDICATED BY THE BLOCK SIZE SPECIFICATION. IN THIS CASE, BLOCK
          SIZE IS ADJUSTED TO INDICATE A MAXIMUM PRINT LINE EQUAL IN LENGTH
          TO THE MAXIMUM PRINT LINE ON THE ASSIGNED PRINTER.
```

Figure 6—1. Typical Text Output Example

6.2.2. Tabular Data

Tabular data and reports generally require a more complex printer file structure since there is a more varied spacing requirement, both vertical and horizontal (Figure 6—2). Also, column headings and similar repetitive items require a more complex program if the file is lengthy. The output records are formed in the same manner as those of regular text files (in the I/O area or work area) and are output to the printer one line at a time.

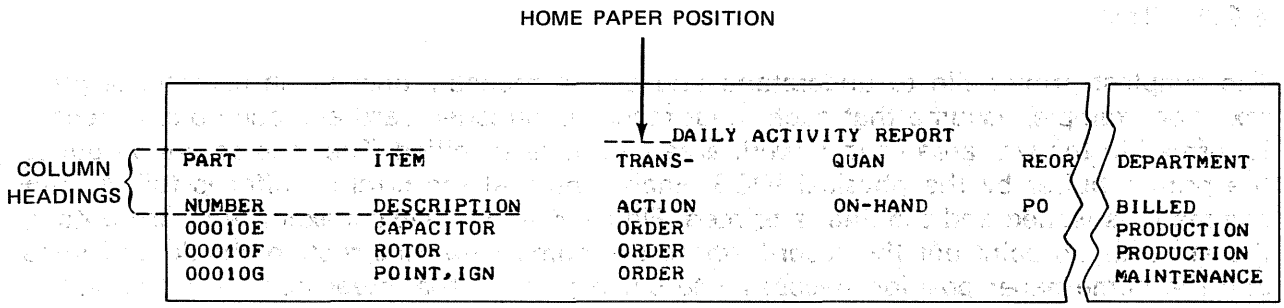


Figure 6—2. Sample Table Printout

6.2.3. Printer Forms

Printer files that complete or that are added to document forms are usually simple to use once they are organized (Figure 6—3). By using the control and overflow macroinstructions, you can achieve desired vertical positioning. By forming your records properly in the I/O area or work area, you can achieve the required horizontal positioning to place the data on the form where it belongs.

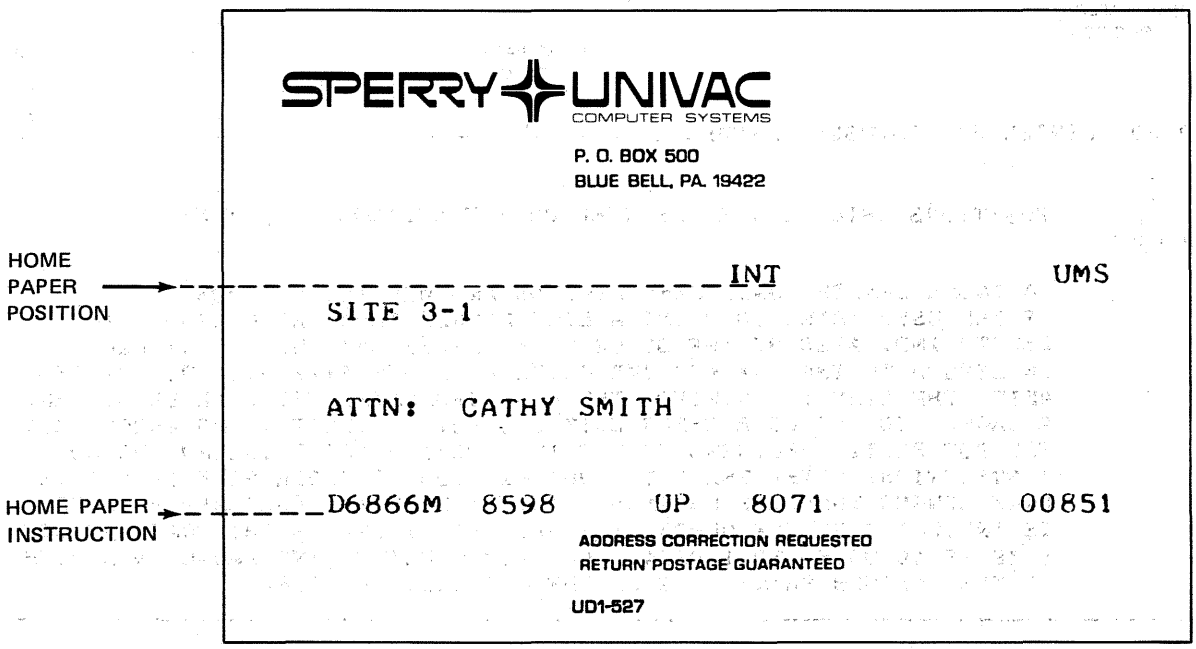


Figure 6—3. Sample Forms Printout

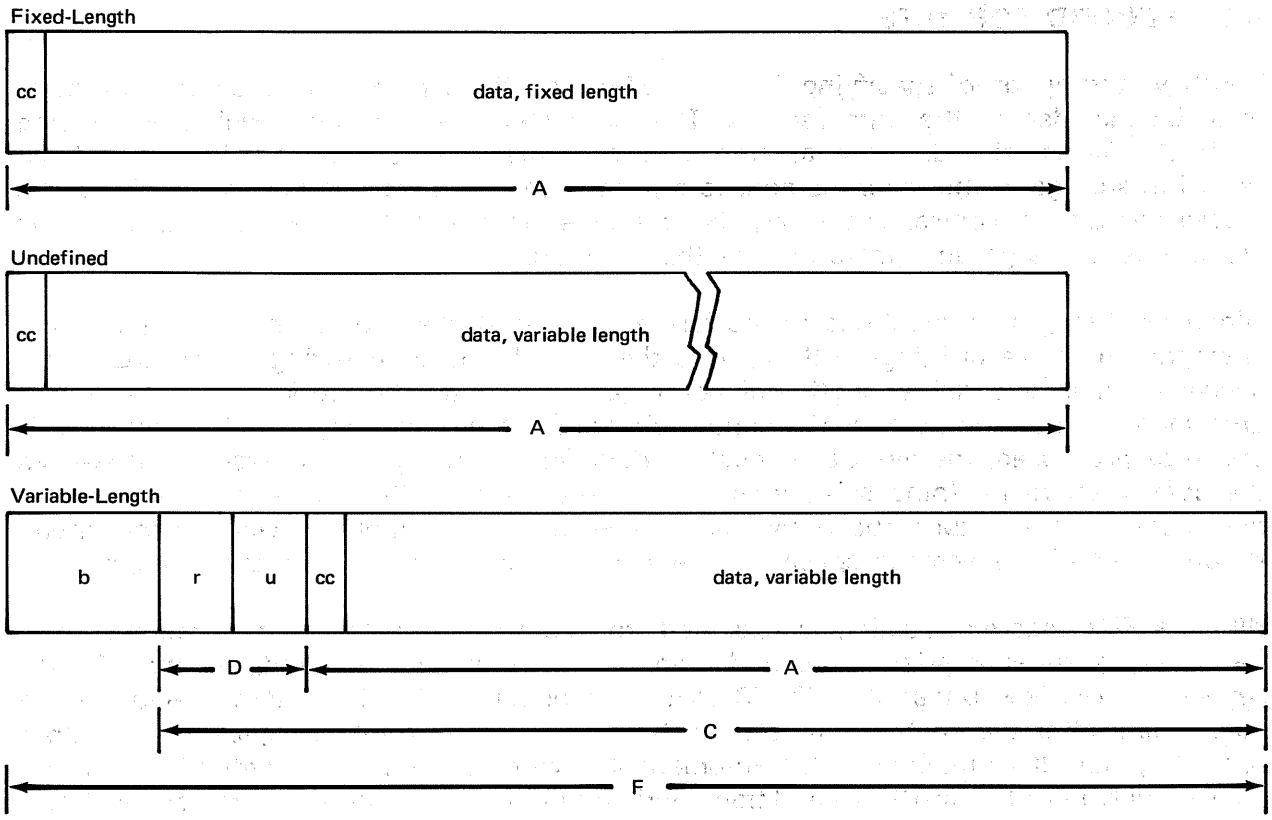
6.3. RECORD FORMATS

You have the option of specifying, in the DTFPR macroinstruction, that a control character is to be included in the data records. This character specifies line spacing or skipping when the file is printed. The character itself is normally not printed, but is a part of the record in storage. If the record is sent to a printer and the user has not specified that the record contains a control character, the character is handled as data and printed. I/O areas must be large enough to include this character.

When fixed-length or undefined record formats are used, the control character is the first character in the record (Figure 6—4). It is the first character following the record length specification in a variable-length unblocked record format. The block size of the output area must include the byte for the control character. If variable-length, unblocked records are to be processed, the block size must account for the initial eight characters as well as the control character (nine bytes total) in the output area. Although these characters do not appear in the output, the output area must be large enough to accommodate them. When a control character is specified, every record must contain a control character.

When a PUT macroinstruction is executed, the control character in the data record is translated to the appropriate command code. (For the character required, see the CTLCHR keyword parameter under the DTFPR macroinstruction, 7.3.) The first character in the output data after the control character is the first character printed. Logical input/output control system (IOCS) modules also automatically issue certain printer control instructions. These involve printer overflow conditions and vertical format control. Parameters available with the macroinstructions that call the IOCS modules into the program allow you to tailor them for each particular task. In this manner, the complex vertical movement and overflow sensing functions are made easy for you to control.

If the CNTRL macroinstruction is to be issued for the printer, the CONTROL keyword parameter is specified and CTLCHR must be omitted.



LEGEND:

b	Block size field, four bytes
cc	Control character, one byte, optional
r	Record length field, two bytes, binary
u	Reserved (two bytes); can be any two characters you choose.
D	Record size field
A	Data record length
C	Variable record length
F	I/O area layout

NOTES:

1. You must align an I/O area so that the first character to be printed falls on a half-word boundary.
2. You must place record length, as a binary number, in the first two bytes of the record length field (r) before printing a variable-length, unblocked record. The record length includes the 4-byte record length field and the control character, if any.

3. You should allocate an even number of bytes for data in I/O areas, even though an odd number of columns are to be printed. To print an odd number of columns, allocate data areas one byte larger than the number of columns to be printed.

Figure 6—4. Printer Record Formats

6.4. VERTICAL FORMAT AND LOAD CODE BUFFERS

In OS/3, you use the job control LCB statement to specify the load code buffer (LCB) and the VFB statement to specify the vertical format buffer (VFB) of the following printer subsystems:

- SPERRY UNIVAC 0768 Printer Subsystem
- SPERRY UNIVAC 0770 Printer Subsystem
- SPERRY UNIVAC 0773 Printer Subsystem
- SPERRY UNIVAC 0776 Printer Subsystem
- SPERRY UNIVAC 0778 Printer Subsystem

Refer to Table A—3 for operational characteristics of these printers, and to the job control user guide, UP-8065 (current version).

6.4.1. Load Code Buffer Interchangeability

There is no interchangeability of printer load code buffers across devices; an LCB job control statement you have specified for a particular printer and print band or drum cannot be used for any other.

6.4.2. LCB Statement Specification

You specify the codes to be assigned to each graphic symbol on the print band or drum by using the X (hexadecimal) or the C (character) positional parameters of the *LCB statement.* You must specify a character code or a hexadecimal specification for each symbol on the band or drum, and you may intermix X and C specifications. Each X or C specification must be complete on a single card. As many specifications as are necessary to specify an entire band or a single repeated font may be made.

The *space* or nonprinting *code* should be specified through the SPACE keyword parameter, and not included in the sequence of codes specified through the positional parameters.

If the number of characters is specified with the NUMBCHAR keyword, it should include only the number of codes specified for graphic symbols and should not include the *space code.*

If the CARTNAME keyword is specified, the operator receives a message to mount the specified band, and program execution is suspended until the operator replies to the message. If the CARTNAME keyword is not specified, no operator message is issued.

6.4.2.1. LCB Specification for the 0773 and 0778 Printers

You may specify 48, 63, 64, or 256 characters for the 0773 and 0778 printers; however, any band having more than 64 characters requires the specification of 256 characters. A 128-character cartridge requires the specification of 256 characters. A 128-character cartridge requires that the 128 characters be specified twice on the LCB statement.

Dualing applies to 48-character bands only; you specify dualing with the DUAL keyword of the LCB statement. Four *dualing characters* may be specified for the 0773 and 0778 printers; these correspond to the 39th, 40th, 44th, and 47th characters on the band.

The CARTID specification is optional for the 0773 and 0778 printers.

6.4.2.2. LCB Specification for the 0770 and 0776 Printers

You may specify from 24 to 384 characters for the load code buffer of the 0770 and 0776 printer. For repeating fonts ranging from 24 to 192 symbols, you need only specify the characters for a single font: for example you would specify only 128 characters through the LCB statement for a repeating font of 128 characters.

Dualing for the 0770 and 0776 printers involves specifying up to four pairs of codes with the DUAL parameter. Each pair consists of one code that has been specified for the load code buffer, followed by one code that has not. Assuming, for example, that a band contains the question mark symbol (?), but not the vertical bar (|), you could substitute ? in your printout for | by specifying DUAL = C?|'. Every time your program outputs the EBCDIC code for a vertical bar to be printed, a question mark appears on the printed listing.

For the 0770 and 0776 printer, you must specify the CARTID parameter, and the code you specify must be the correct one for the cartridge you intend to use.

You may also specify a *mismatch character* for the 0770 or 0776 printer: that is, you may specify what character, other than blank (space), is to be printed whenever a *character mismatch* occurs.

6.4.2.3. LCB Specification for the 0768 Printer

You need only specify the string of codes for the load code buffer and the MISM, SPACE, and TYPE parameters. You may also specify the optional NUMBCHAR parameter, but the other parameters of the LCB statement do not apply to the 0768 printer.

6.4.3. Vertical Format Buffer Interchangeability

Table 6—1 summarizes the conditions under which a properly specified VFB statement for one printer may be used with other devices. There is no difference in the appearance of the printed results if the same VFB statement is used from machine to machine under these conditions.

6.4.4. VFB Statement Specification

Specifying a VFB job control statement involves visualizing the form with numbered lines. An 11-inch form to be printed at a density of 8 lines per inch has 88 lines. At 6 lines per inch, an 11-inch form has 66 lines. The first printable line on a form is line 1. The last line on an 11-inch form, printed at 8 lines per inch, is line 88.

The vertical format buffer can be specified and the program designed so that most printing occurs between the home paper code position and the overflow code position on the form. The position of the home paper code determines the amount of unprinted space at the top of the form, and the overflow code position approximates the amount of unprinted space at the bottom of the form.

Because lines may be printed (and the form advanced) beyond the overflow position, you must provide enough space between the overflow code position and the bottom of the form for any lines (and form advances) that must fit on a page. Note that you must provide at least four lines between the overflow code position and the bottom of the form. This is particularly important for VFBs that are used to print dumps, librarian runs, assemblies. etc.

6.4.4.1. Specifying Home Paper Position

The HP code specified on the VFB job control statement gives the lines number location of the *home paper* position: The specification HP=5 places the home paper position on the fifth line of the form.

6.4.4.2. Specifying Forms Overflow Position

You use the OVF keyword of the VFB job control statement to specify the *forms overflow* position to printer SAM. You should not specify the OVF keyword if you do not intend to use it.

When an overflow code is placed in the buffer, a space form operation (advance paper *n* lines) which would move the form to or beyond the overflow position causes forms overflow to be detected. On detecting forms overflow, printer SAM takes action according to your specifications of the PRINTOV keyword parameters in the DTFPR declarative macro.

No indication of overflow is returned to you, except that printer SAM transfers control to your overflow routine if you have so specified. In this routine, you may take such actions as skipping to the top of the next page, printing page numbers, printing subtotals, and so forth.

You must specify the overflow code position so that enough space is left between the overflow position and the bottom of the form to print and space all of the lines that are to appear on the page. Printer SAM may print a line on or below the overflow code position and perform spacing before branching to your overflow routine.

If a PUT appears in the overflow routine, the effect will be to perform spacing and/or print a line between the overflow position and the bottom of the form.

If the user does not specify the PRINTOV keyword in the DTF, data management takes no overflow action; in this event, the BAL programmer who is not counting lines in his program runs some risk of tearing the form by printing on or too near the perforations.

Table 6—1 summarizes the combinations of device-independent control character codes permitted when using each type of printer with or without the TYPE parameter specification on the VFB job control statement. Because it describes the allowable use of device-independent control character codes, Tables 6—1, 7—1, and 7—2 should be used conjunctively. Table 7—1 interprets the control character codes associated with each of four printer functions: print and space, print and skip, spacing, and skipping. Table 7—2 interprets overflow and home paper control character codes.

Note that the 0770 printer has two overflow codes (9 and 12) that the data management PRTOV imperative macro can detect selectively. You should specify a secondary overflow code (hexadecimal code 9, specified through the OVF2 keyword) only with the 0770 printer and only if you are using the PRTOV macro, or (if you are not using data management) its PIOCS equivalent.

6.4.4.3. Specifying Special Forms

If you specify the FORMNAME keyword in the VFB job control statement, the operator is issued a message to mount the specified form, and program execution is halted until the operator replies.

6.4.4.4. Paper Tape Loop, 0768 Printer

For the *0768 printer,* you must provide *both* a paper tape loop and a VFB job control statement. The paper tape should be punched to agree exactly with the VFB statement, with the following exceptions:

1. A 7 should not be punched on the tape. Home paper should be punched either as 15 (for 8 lines per inch spacing) or as 14 (for 6 lines per inch). Only one home paper code may be punched on the tape.

2. Channel 1, 2, 3, or 12 should not be punched on the tape.

Table 6—1. VFB Statement Specification and Interchangeability

Specification of TYPE Keyword	Statement May be Used With Printer Types (Note 2)	Other Keywords that the User May Specify (Note 1):																	
		HP	OVF	OVF2	CD1	CD2	CD3	CD4	CD5	CD6	CD7	CD8	CD9	CD10	CD11	CD12	CD13	CD14	CD15
=0773 (or keyword omitted)	0768, 0770, 0776 TYPE keyword omitted; 0773; 0778	X	X			X	X	X	X	X									
=0768	0768; 0770, 0776 TYPE keyword omitted	X	X			X	X	X	X	X		X		X	X		X	X	X
=0770 Note 3	0770 Note 4	X	X	X		X	X	X	X	X		X		X	X		X		
=0776	0768, 0770 TYPE keyword omitted; 0776	X	X			X	X	X	X	X		X		X	X		X		

LEGEND:

X Keyword may be specified.

▒ Keyword may not be specified.

NOTES:

1. This table is concerned with only the keywords shown; the user may always specify the LENGTH, DENSITY, FORMNAME, and USE keywords.
2. The TYPE keyword should be specified only if a particular printer type must be used. A VFB statement designed for a particular printer (using the permitted keywords shown for that printer in Table 6—1) may be used with other printers only if the TYPE keyword is omitted.
3. The user should specify TYPE=0770 only if he specifies a secondary overflow code (OVF2) or if he specifies multiple home paper positions.
4. If the user does not specify the OVF2 keyword, he may use the VFB statement (TYPE keyword omitted) with the 0768, 0770, and 0776 printers.
5. The secondary overflow code (OVF2) should be specified only by a data management user who issues the PRTOV imperative macro. Refer to the PRINTOV keyword parameter of the DTFPR and PRIO declarative macros.

6.4.4.5. Vertical Format Buffer Statement Example

The following example might be a typical VFB statement used to set up the forms spacing and skipping required for a printed report.

```
1  LABEL     ΔOPERATIONΔ      OPERAND                                    Δ        COMMENTS
           10          16
//  VFB  LENGTH=66,DENSITY=6,FORMNAME=TESTPR,TYPE=0770,HP=1,OVF=52,
//1  CD2=6,CD3=46
```

Line 1 specifies that the operator use a form called TESTPR. The total number of lines per page is 66 at 6 lines per inch. The 0770 printer is being used. The home paper position is on line 1 and the overflow area of each page begins on line 52. Note that this includes the 4-line space between the overflow code position and bottom of page used if a dump, librarian run, or assembly is executed. The number 52 is used by data management to test for page overflow conditions. Then, according to your PRTOV imperative macro specifications, data management skips to the overflow routine or register to handle your overflow routine address.

Line 2 specifies a channel code of CD2 with a line number of 6. This means that whenever a CNTRL filename,SK,2 imperative macro is issued in the program, the printer immediately skips 6 lines on the page. Here a detail line of a report might be printed. On the other hand, if a CNTRL filename,SK,,2 macro is issued in the program, the printer skips 6 lines after printing the detail line. The first macro use illustrates immediate skipping and the second, delayed skipping.

A second channel code of CD3 indicates 46 lines. Similarly, the CNTRL macro could specify an immediate or delayed skip of 46 lines where a final total might be printed.

It is important to realize that the CD number (VFB parameter) relates to a channel code and the value indicated on the right of the equal sign indicates the line number to which the printer skips.

7. Function and Operation of SAM Printer Files

7.1. GENERAL

The OS/3 includes data management modules that can be used to move and manipulate sequential access method (SAM) printer files. These modules can function with five different printer subsystems:

- SPERRY UNIVAC 0773 Printer Subsystem
- SPERRY UNIVAC 0770 Printer Subsystem
- SPERRY UNIVAC 0768 Printer Subsystem
- SPERRY UNIVAC 0776 Printer Subsystem
- SPERRY UNIVAC 0778 Printer Subsystem

This section contains a brief functional description of printer file SAM operation. This is followed by a detailed description of the declarative macroinstruction that is used to define a printer file and of the imperative macroinstructions that initiate, conduct, and conclude file processing.

7.2. FUNCTIONAL DESCRIPTION

At system installation time, the system macro library (YMAC) is loaded with source code modules that are common to several machine operations. These modules include data management modules that are common to several device types and access methods.

When assembling the program, you define the characteristics of printer file involved in the operation, using the define the file (DTFPR) declarative macroinstruction. This macroinstruction creates a table of file characteristics, in main storage, that is referenced by your program.

The source code modules that are required for your program are called in from the system macro library at program assembly time by using imperative macroinstructions. These imperative macroinstructions are included in the program you are assembling and result in the creation of inline code at the point where the assembler encounters the macroinstruction. Positional parameters contained in the imperative macroinstructions allow you to modify the basic assembled module so that it meets your particular requirements.

When the file is opened, the characteristics in the file control table set up by the DTFPR are examined to determine that they are valid and, if required, that they are present. The output records are formed either in the I/O area or a work area. A form of overlap processing can be achieved by assigning either two I/O areas or an I/O area and a work area. In this way, records can be constructed in one area while others are simultaneously being output to the printer from the other area.

Logical input/output control system (IOCS) modules also automatically issue certain printer control instructions. These involve printer overflow conditions and vertical format control. Parameters available with the macroinstructions that call the IOCS modules into the program allow you to tailor them for each particular task. In this manner, the complex vertical movement and overflow sensing functions are made easy for you to control.

A typical printer SAM operating sequence is described in the following example, which show the sequence and function of each macroinstruction. The macroinstructions are discussed in detail in 7.3 and 7.4.

Example:

LABEL (1)	OPERATION (10)	OPERAND (16)	COMMENTS
// JOB	EXAMPLE		
*			Job stream to execute assembler
// EXEC	ASM30		
/$			Start of data
TAPEIN	DTFMT		Keyword parameters to define magnetic tape input file and the printer output file, including options and physical characteristics
* PRTOUT	DTFPR		
			Storage definition
	OPEN	TAPEIN	Initiates transient routines to check DTFMT and DTFPR that all necessary parameters are supplied and are valid. The physical IOCS then reads the input record into the I/O area.
*	OPEN	PRTOUT	
*			
	GET	TAPEIN, WRKAREA	Makes the next logical record available to user by placing it in the work area, labeled WRKAREA.
*			
*			Processing steps

LABEL (1)	OPERATION (10)	OPERAND (16)	COMMENTS
	CNTRL	PRTOUT,SK,7	Macro specifies an immediate skip to the home paper position (code 7), before printing.
*			
*			
	PUT	PRTOUT,WRKAREA	Delivers an output record from the work area labeled WRKAREA to the output area.
*			Physical IOCS then outputs the record to the printer.
*			
	PRTOV	PRTOUT,12	
*			
*			Specifies that forms overflow for the 0770 Printer being used will be indicated by VFB code 12. As the third positional parameter has been omitted, an automatic skip to home paper position will occur when forms overflow is detected.
*			
*			
*			
*			
	CLOSE	TAPEIN	Initiates file processing termination procedures for the input and output files.
*			
	CLOSE	PRTOUT	
	END		Ends assembly
/*			End of data
// DVC	20	// LFD PRNTR	(Required for SNAPs and DUMPs)
// DVC	20	// LFD PRTOUT	
// DVC	90	// LFD TAPEIN	
// EXEC			Executes program
/&			End of job
// FIN			

DTFPR

7.3. DEFINE A SAM PRINTER FILE (DTFPR)

Function:

The DTFPR declarative macroinstruction is required to define each printer file processed in the program. Following the format is a listing, in alphabetical order, of the required and optional keyword parameters which may appear in the operand of the DTFPR macroinstruction. A description of each keyword parameter follows the format. A summary of the keyword parameters is given in Table 7—3.

A comma is shown preceding each keyword parameter except the first, to remind you that all keywords coded in a string must be separated by commas. However, a comma must neither be coded in column 16 of a continuation line, nor follow the last keyword in the string. Refer to the coding example that follows.

Format:

LABEL	Δ OPERATION Δ	OPERAND
filename	DTFPR	[BLKSIZE=n] [,CONTROL=YES] [,CTLCHR=DI] [,ERROR=symbol] ,IOAREA1=symbol [,IOAREA2=symbol] [,IOREG=(r)] [,OPTION=YES] [,PRAD=n] [,PRINTOV= { SKIP / symbol / YES }] [,RECFORM= { FIXUNB / UNDEF / VARUNB }] [,RECSIZE=(r)] [,SAVAREA=symbol] [,UCS= { OFF / ON }] [,WORKA=YES]

Keyword Parameter BLKSIZE:

BLKSIZE=n
Specifies the length of the I/O area in bytes. If the record is variable length or undefined, n specifies the length of the longest block, including block size and record size bytes for the variable-length unblocked records.

If omitted, the block size (120, 121, 128, or 129) is determined from the CTLCHR and RECFORM keyword parameters.

With RECFORM=FIXUNB specified and CTLCHR not specified, block size is set to 120. With RECFORM=VARUNB and CTLCHR specified, block size is set to 129.

The minimum block size is two bytes. With CTLCHR=DI, RECFORM=VARUNB, and maximum print position options installed on the 0773, 0778 and 0770 printers, block size can be a maximum of 141 bytes for the 0768 printer, 145 bytes for the 0776 printer, 169 bytes for the 0770 printer, and 153 bytes for the 0773 and 0778 integrated printers. If these optional features are not installed on your printers, the maximum block sizes for the 0773 and 0778 printer are 129 bytes and for the 0770 printers are 141 bytes.

Keyword Parameter CONTROL:

CONTROL=YES
Specified if spacing or skipping lines on the printer is controlled by your program through the CNTRL macroinstruction.

The CONTROL keyword parameter and the CTLCHR keyword parameter are mutually exclusive. If they are both used in the same DTF, an error flag appears in the output listing and the control specification is ignored.

Keyword Parameter CTLCHR:

This keyword parameter is specified when you wish to use a control character with data records.

CTLCHR=DI
Specifies the device-independent, 2-digit, hexadecimal control character code listed in Table 7—1.

The use of device-independent characters allows a single character for each function to be used with any printer, even if the hardware opcode varies with printer type. With this set of characters, some substitutions have to be made to compensate for device characteristics (Table 7—2).

Table 7—1. Device-Independent Control Character Codes (Part 1 of 2)

Function	DI Code (Hex.)	Printer			
		0773 and 0778	0770	0768	0776
No-op	00				
Print and space n lines (Note 8)		Note 6			
n = 0	10				
1	01				
2	02				
3	03				
4	04				
5	05			Note 1	
6	06			Note 1	
7	07			Note 1	
8	08			Note 1	
9	09			Note 1	
10	0A			Note 1	
11	0B			Note 1	
12	0C			Note 1	
13	0D			Note 1	
14	0E			Note 1	
15	0F			Note 1	
Print and skip to code n (Note 7)					
n = 1 (OV)	11		Code 12 (OV)	Chan 9 (OV)	Code 12 (OV)
2	12			Note 5	
3	13			Note 5	
4	14				
5	15				
6	16				
7 (HP)	17		Note 4 (OV)	Chan 15 Note 3	Code 7 (HP) Note 4
8	18	Code 2			
9	19	Code 1 (OV)	Note 2 (OV)		Code 12 (OV)
10	1A	Code 3			
11	1B	Code 4			
12	1C	Code 1 (OV)	Note 2 (OV)	Chan 9 (OV)	Code 12 (OV)
13	1D	Code 5			
14	1E	Code 7 (HP)	Code 7 (HP)	Chan 15 Note 3	Code 7 (HP)
15	1F	Code 7 (HP)	Code 7 (HP)	Chan 15 Note 3	Code 7 (HP)
Space n lines (Note 8)		Note 6			
n = 1	51				
2	52				
3	53				
4	54			Note 1	
5	55			Note 1	
6	56			Note 1	
7	57			Note 1	
8	58			Note 1	
9	59			Note 1	
10	5A			Note 1	
11	5B			Note 1	
12	5C			Note 1	
13	5D			Note 1	
14	5E			Note 1	
15	5F			Note 1	

Table 7—1. Device-Independent Control Character Codes (Part 2 of 2)

Function	DI Code (Hex.)	Printer			
		0773 and 0778	0770	0768	0776
Skip to code n (Note 7)					
n = 1 (OV)	21		Code 12 (OV)	Chan 9 (OV)	Code 12 (OV)
2	22			Note 5	
3	23			Note 5	
4	24				
5	25				
6	26				
7 (HP)	27		Note 4	Chan 15 (Note 3)	Code 7 (HP) Note 4
8	28	Code 2			
9	29	Code 1 (OV)	Note 2		Code 12 (OV)
10	2A	Code 3			
11	2B	Code 4			
12	2C	Code 1 (OV)	Note 2	Chan 9 (OV)	Code 12(OV)
13	2D	Code 5			
14	2E	Code 7 (HP)	Code 7 (HP)	Chan 15 (Note 3)	Code 7 (HP)
15	2F	Code 7 (HP)	Code 7 (HP)	Chan 15 (Note 3)	Code 7 (HP)

LEGEND:

OV Overflow code

HP Home paper code

NOTES:

1. Line spacing of 4—15 lines on the 0768 printer is accomplished by issuing multiple I/O commands. The commands are issued by data management.

2. Code 12 is the primary forms overflow code on the 0770 printer. Code 9 can also be detected as forms overflow code, using the PRTOV macro. If the PRTOV macro is not used, however, code 12 should be used as overflow code, and code 9 should not be placed in the VFB.

3. On the 0768 printer, you must use both a vertical format buffer and a paper tape loop. Using DI codes 27, 2E, or 2F as control characters has no direct effect on line spacing; this is controlled by what is punched on the paper tape loop. The actual selection of six or eight lines-per-inch spacing occurs when the operator sets up your form on the 0768 printer to register at home paper position and pushes the HP button twice.

   If channel 14 is used as home paper code on the paper tape loop, this causes printing at six lines per inch; using channel 15 results in eight lines-per-inch spacing. These two codes should not be intermixed. When a DI code for skip to code 7 is issued, a skip is issued to channel 15 on the 0768 printer — this causes an advance to either channel 14 or 15, whichever in punched on the paper tape loop.

4. Code 7 must be used as the home paper code on the 0770 and 0776 printers.

5. For the 0768 printer, the software (physical IOCS) provides codes 2 and 3.

6. Line spacing (print density) is switch-controlled on the 0773 and 0778 printers. You issue instructions to the operator via the job control VFB statement by using its FORMNAME and DENSITY parameters, and he sets the line rate when the vertical format buffer is loaded.

   For the 0770 and 0776 printers, line spacing is software-controlled via the DENSITY parameter of the VFB statement.

7. Code n specifies channel code CD1 through CD15. (See 7.4.3.)

8. Code n specifies number of lines to be spaced. (See 7.4.3.)

Table 7—2. Overflow and Home Paper Control Character Codes

Code	Printers			
	0773 and 0778	0770	0768	0776
Overflow	Code 1	Code 9 Code 12*	Code 9	Code 12
Home paper	Code 7	Code 7	Code 14 Code 15	Code 7

*Code 12 is the primary code; code 9 should not be used with the 0770 printer unless the PRTOV macro is used.

Keyword Parameter ERROR:

ERROR=symbol

Specifies the address of a special error handling routine to which you may have control transferred when a fatal hardware or detectable logic error occurs on your file. Information concerning reasons for the error will be contained in *filenameC* (see B.4).

Keyword Parameter IOAREA1:

IOAREA1=symbol

Defines the address of the I/O area. Each input or output file must have an area reserved for its individual use. The I/O area must be aligned so that the first byte of data (the character to be printed in column 1) is on a half-word boundary.

The I/O area must provide space for everything included as part of the block length. You must allocate I/O areas that contain an even number of bytes, excluding the control character, if any. Data management inserts a nonprinting character at the end of any user print line which contains an odd number of characters. This extra character is placed in an I/O area before printing the line.

Examples:

```
   LABEL      ΔOPERATIONΔ        OPERAND                    Δ
   1          10         16
1. |          |DS      |OH      |                Half-word alignment
   |IOA1      |DS      |DCLnn   |                IOAREA1
   |IOA2      |DS      |DCLnn   |                IOAREA2
   |          |        |        |
2. |          |DS      |OH      |                Half-word alignment
   |          |DS      |CL1     |                Extra byte
   |IOA1      |DS      |DCLnn   |                IOAREA1
   |          |DS      |CL1     |                Extra byte
   |IOA2      |DS      |DCLnn   |                IOAREA2
```

1. Without control character: nn must be even and greater than or equal to BLKSIZE.

2. With control character: nn must be odd and greater than or equal to BLKSIZE.

   Note, in this case, that it is necessary to reserve one byte after the half-word alignment, because the first character to be printed must be on a half-word boundary, and a control character is being used. Remember that the block length always includes one byte for the control character when one is used.

Keyword Parameter IOAREA2:

IOAREA2=symbol
Provides a second I/O area to allow overlapped processing and speed I/O operations.

The same conditions that apply for IOAREA1 also apply for IOAREA2. If IOAREA2 is specified and WORKA is not, you must specify the IOREG keyword parameter. No additional processing speed is obtained when both the IOAREA2 and WORKA keyword parameters are specified. Most efficient processing is obtained with either of the following:

IOAREA1, IOAREA2, and IOREG
or
IOAREA1 and WORKA

Keyword Parameter IOREG:

IOREG=(r)
Specified when a general register (2 through 12) is used to reference current data. If SAVAREA is specified, register 13 is also available. The register must be specified if two output areas are used and records are not to be processed in a work area.

After each line is printed, and before returning to your program, data management loads the register specified by IOREG with one of the following, depending on your record format:

- The address where the first character of the next record to be output should be placed when fixed-length, unblocked or undefined records are used. This is either a control character (if used) or the character in print position 1.

- The address of the location where the 4-byte record length field, followed by the control character, if any, and data to be printed, should be placed for variable-length unblocked records.

The IOREG and WORKA keyword parameters are mutually exclusive. If both are specified, the WORKA keyword parameter is ignored and an error flag appears in the DTF listing.

Keyword Parameter OPTION:

OPTION=YES
Allows you to specify an optional file: one which you anticipate will not invariably be required to be printed every time your program is executed.

When the OPTION keyword parameter is used, the PUT, CNTRL, and PRTOV imperative macroinstructions are disabled:

- if an OPT positional parameter is included in the DVC job control statement and the device is not available at execution time; or
- when no device is assigned to the file by your job control statements, (i.e., no DVC-LFD sequence).

If the OPTION keyword parameter is used under these conditions, the occurrence of a PUT, CNTRL, or PRTOV macroinstruction results in a branch back to your program, and no I/O is performed.

If the OPTION keyword parameter is not specified and one of the two previously stated conditions exists, the file is not opened; an error bit is set in the file table and the program branches to your error routine. If you have not provided an error routine, control returns to you inline.

Keyword Parameter PRAD:

PRAD=n
Allows you to specify a standard form advance of from 1 to 15 lines, where *n* is the number of lines the form is to be advanced and ranges from 1 through 15. The form advance takes place after the line is printed.

On the 0768 printer, spacing of 4 through 15 lines is accomplished by issuing multiple I/O commands because this device can advance only three lines under one I/O command. If the PRAD and CTLCHR keyword parameters are not specified, PRAD=1 is assumed; if both are specified, the control character will determine the line advancement.

A delayed CNTRL macro instruction, which spaces or skips lines after printing, overrides the PRAD keyword parameter specification for one print operation only.

Keyword Parameter PRINTOV:

This keyword parameter specifies the action to be taken when the forms overflow code is detected during a space or print and space command. The forms overflow code cannot be detected on skip or print and skip commands.

There are three PRINTOV option specifications that can be made:

PRINTOV=SKIP
Specifies an automatic skip to the home paper position.

PRINTOV=symbol
Specifies that control is transferred to your overflow routine. When this option is specified, the form will not be automatically advanced to the home paper position.

In the overflow routine, you may print total lines, skip to home paper, and print page headings. To branch back to the point in the program where processing would have continued (if overflow hadn't occurred) you may use the address in register 14. If imperative macros (CNTRL, PUT) are issued in the overflow routine, you should store register 14 before issuing the macro and restore it after the instruction is executed.

If overflow is detected during the issuance of a CNTRL macro, control will be transferred to your overflow routine.

PRINTOV=YES
Specifies that the PRTOV imperative macroinstruction will be used in the program to control overflow detection and actions.

To use the forms overflow features, you must load the proper forms overflow code in the VFB for the 0770, 0776, 0768, and 0773 printers. If the PRTOV imperative macroinstructions are to be used with the 0770 printer, each code required must be loaded in the VFB. The VFB is loaded through job control statements. The 0768 printer also requires a paper tape loop which must agree with the VFB statement.

OS/3 data management will execute one user-issued CNTRL or PUT macroinstruction after the immediate CNTRL or PUT instruction on which forms overflow was detected. If the imperative macro issued after the macro on which overflow was detected results in a forms advance to a skip code (VFB or paper tape code), your overflow options will not be executed.

Keyword Parameter RECFORM:

One of the following three options describing the record format should be specified:

RECFORM=FIXUNB
Fixed-length records for print files are assumed by the logical IOCS when this keyword parameter is omitted.

RECFORM=UNDEF
Used for undefined records. You must specify the RECSIZE keyword parameter when this format is used and enter the size of each record into the RECSIZE register before issuing each PUT macroinstruction. See Figure 6—4.

RECFORM=VARUNB
Used for variable-length, unblocked records.

Before issuing a PUT macroinstruction, you must include a valid record length value (a binary number) in the record. This record length, inserted into the first two bytes of the 4-byte record size field, must include the control character, if provided, as well as four bytes for the record size field itself. See Figure 6—4.

Keyword Parameter RECSIZE:

RECSIZE=(r)
For output files with undefined record format, specifies the number (2 through 12) of the general register that holds the size of the output record. If SAVEAREA is specified, register 13 may be used as the RECSIZE register. The record size must be entered into the general register before the PUT macroinstruction is issued.

Keyword Parameter SAVAREA:

SAVAREA=symbol
If you have a program written for a SPERRY UNIVAC 9200/9300 System or similar system (in which register 13 was used), you may convert the program to run under OS/3 by adding a 72-byte labeled save area (aligned on a full-word boundary) by adding this keyword parameter.

This keyword parameter should be specified by you for each DTF in a program. Only one register save area is needed per program. Refer to 1.4 for the content of this area.

If this keyword is not present in a DTF, logical IOCS will assume that register 13 has been loaded with the address of a 72-byte save area, aligned on a full-word boundary.

Keyword Parameter UCS:

This keyword parameter is specified to determine whether character mismatches are to be ignored or not. A mismatch occurs whenever the printer attempts to print a bit configuration which is not present in the printer's load code buffer. This parameter has two formats:

UCS=OFF
Character mismatches are ignored by the program. All unprintable characters are printed as the nonprinting code (NP) in the load code buffer. When the standard load code is used, a blank (40_{16}) is printed.

UCS=ON
The operator is notified of character mismatches. If an error routine has been provided, control will be transferred to that routine and the registers restored. If no error routine has been provided, a message will be issued and control will return to the program as if no error had occurred.

Keyword Parameter WORKA:

WORKA=YES
Specifies a work area for preparation of output records. The address of the current work area must be specified with each PUT macroinstruction.

The WORKA and IOREG keyword parameters are mutually exclusive; if both are specified, the WORKA keyword parameter is ignored and an error flag appears in the DTF listing. Best efficiency is obtained with either of the following combinations:

IOAREA1, IOAREA2, and IOREG
or
IOAREA1 and WORKA

Example:

```
LABEL     ΔOPERATIONΔ          OPERAND                    Δ
1         10        16                                    72
*         PRINTER FILE DEFINITION - SAMPLE
STAMP     DTFPR     IOAREA=GREEN,                         X
                    BLKSIZE=132,                          X
                    RECFORM=VARUNB
```

Table 7—3. Summary of Keyword Parameters for DTFPR Macroinstruction

Keyword	Specifications	OUTPUT File	Remarks
BLKSIZE*	n	X	The maximum block size in bytes
CONTROL	YES	X	Specified if CNTRL macro instruction is issued to line spacing or skipping
CTLCHR	DI	Y	Device-independent control characters
ERROR	symbolic label	X	Address of your unrecoverable error routine
IOAREA1	symbolic label	R	Address of output area
IOAREA2	symbolic label	X	Address of alternate output area
IOREG	(r)=general register	X	General register (2 through 12) that contains the address of the current record each time a PUT is issued. Register 13 may be used when SAVAREA keyword parameter is used.
OPTION	YES	X	Specifies an optional file that has not been allocated by job control
PRAD	n	X	Number of lines (1 to 15) to be spaced
PRINTOV	SKIP	X	Automatic skip to home paper position on printer
	symbolic label	X	Address of your overflow routine
	YES	X	Specifies execution of PRTOV imperative macros in program
RECFORM*	FIXUNB	Y	For fixed-length records
	UNDEF	Y	For undefined records
	VARUNB	Y	For variable-length, unblocked records
RECSIZE	(r)=general register	X	General register (2 through 13) that contains the length of each record when RECFORM=UNDEF
SAVAREA	symbolic label	X	Specifies 72-byte register save area
UCS	OFF	Y	Character mismatches will be ignored.
	ON	Y	Operator is notified of character mismatches.
WORKA	YES	X	Process records in work area.

LEGEND:

R Required
X Optional
Y One option required
Value assumed if keyword is not specified
* Parameter may be changed on DD job control statement

7.4. IMPERATIVE MACROINSTRUCTIONS

There are five imperative macroinstructions available to you for processing SAM printer files:

Macroinstruction	Use
OPEN	File control
PUT	Record processing
CNTRL	File control
PRTOV	File control
CLOSE	File control

The following paragraphs describe the macroinstructions and their related parameters in detail and provide you with coding examples and explanations, when required, to clarify use.

OPEN

7.4.1. Open a Printer File (OPEN)

Function:

Before a file can be accessed by the logical IOCS, an OPEN macroinstruction is issued. The transient routine called by the OPEN macroinstruction performs certain validation checks and initiates file processing. A check is made to determine if all the necessary keyword parameters defining the file have been supplied. The device allocation performed by the job control program is determined, and the I/O register is set up if one is specified.

Format:

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

Positional Parameter 1:

filename
Is the label of the corresponding DTF macroinstruction in the program. The filename may have a maximum of seven characters. The maximum number of filenames is 16.

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

Examples:

	LABEL (1)	ΔOPERATIONΔ (10)	OPERAND (16) Δ
1.	DMO1	OPEN	PRNT1,PRNT2
2.		LA	1,PRNT4
	DMO2	OPEN	(1)

1. Open printer files labeled PRNT1 and PRNT2.

2. Open printer file labeled PRNT4.

PUT

7.4.2. Output a Record (PUT)

Function:

The PUT macroinstruction delivers an output record to the logical IOCS in either an output area or a work area you have specified. This output record could be a line to be printed or, if the record comprises only control characters, could cause carriage movement alone.

Format:

LABEL	Δ OPERATION Δ	OPERAND
[name]	PUT	{filename / (1) / 1} [, {workarea / (0) / 0}]

Positional Parameter 1:

filename
Is the label of the corresponding DTF macroinstruction in the program.

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

Positional Parameter 2:

workarea
Is the label of the work area from which the record may be obtained.

(0) or 0
Indicates that register 0 has been preloaded with the address of the work area.

If omitted, indicates that you have chosen processing either by means of a register (IOREG keyword parameter) or by directly accessing the data relative to the name of the I/O area.

NOTE:

When the work area is specified, the keyword parameter WORKA=YES must be present in the DTF statement.

Examples:

	LABEL (1)	OPERATION (10)	OPERAND (16)
1.		PUT	PRNT1
2.		PUT	PRNT2,WORK1
		LA	0,WORK2
3.		PUT	PRNT4,(0)

1. Print a line, or space the paper. The record is located in an output area. The file characteristics are contained in the DTF table labeled PRNT1.

2. Move the record in the work area WORK1 to an output area and print, space paper, or both.

3. Move a record from the work area whose address is stored in register 0 and print, space paper, or both.

Programming Considerations:

When WORKA=YES is used, a work area must be specified with each PUT macroinstruction. When only one I/O area is specified, you may move the records to be printed directly into the I/O area. If you specify two I/O areas, you must use the I/O register (IOREG keyword parameter) to move records to be printed into one of the I/O areas before issuing each PUT macroinstruction.

When printing variable-length, unblocked records, you must place the record length (as a binary value) in the first two bytes of the record size field. Undefined-length records require that you place the record length in the required register (specified by the RECSIZE keyword parameter) before issuing each PUT macro instruction.

When you want to use control characters to position the form only (no printing) for variable-length, unblocked, or undefined record formats, you can indicate that the record contains no characters to be printed.

When you are printing records in undefined or variable, unblocked format, printer SAM checks whether the number of columns being printed is greater than zero every time. If you try to print zero columns, data management normally issues error message DM18 (RECORD SIZE INVALID), sets the *record size invalid* error flag (byte 0, bit 7) in *filenameC,* and branches to your error routine. Refer to Appendix B.

However, this error processing does not take place when you are performing an advance-only operation using control characters; you may then indicate that there is no data to be printed. With either undefined or variable, unblocked records, you must place the record size in either the RECSIZE register or in the record size field in the record before issuing each PUT macro. To indicate that you have no data to be printed when your records are variable, unblocked, you may place a record size of five bytes in the record size field — provided that you limit this to advance-only operations.

When you are printing a record containing an odd number of characters to be printed, data management places a nonprinting character in the I/O area after the last byte of user-supplied data.

7.4.3. Control Printer Forms (CNTRL)

Function:

The CNTRL macroinstruction controls printer forms spacing and skipping. Forms motion can occur before, after, or both before and after a line is printed.

Format:

LABEL	ΔOPERATIONΔ	OPERAND
[name]	CNTRL	{filename / (1) / 1}, {SK / SP} [,m] [,n]

Positional Parameter 1:

filename
Is the label of the corresponding DTF macroinstruction in the program.

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

Positional Parameter 2:

SK
Indicates that forms skipping is desired.

SP
Indicates that forms spacing is desired.

Positional Parameter 3:

m
Specifies either the number of lines (0 through 15) for immediate spacing, or the channel code (1 through 15) for immediate skipping. The PUT macro performs spacing after printing. The number of lines spaced is determined by the PRAD keyword parameter (default causes advance of one line). Immediate CNTRL spacing is in addition to any spacing performed on a previous PUT macroinstruction. It should be noted that forms overflow processing can be initiated after CNTRL processing (immediate or delayed) is performed.

Positional Parameter 4:

n

Specifies either the number of lines (0 through 15) for delayed spacing, or the channel code (1 through 15) for delayed skipping.

Examples:

```
   LABEL      ΔOPERATIONΔ      OPERAND
   1          10               16                    Δ
1. DMO7       CNTRL            PRINT2,SP,3,5
                               PRINT2,WORK1
2. DMO8       CNTRL            PRINT2,SK,7
                               PRINT2,WORK2
3.            CNTRL            PRINT2,SK,,7
```

1. Space three lines before printing the next line in file PRINT2 and five lines after printing the next line.

2. Skip to the home paper position.

3. After printing the next line, skip to the home paper position.

Programming Considerations:

Because of differences between the 0770, 0773, 0776, 0778, and 0768 printers, substitutions have to be made for some skip codes on each type of printer. Table 7—4 lists these substitutions. On the 0768 printer, data management accomplishes spacing greater than three lines by issuing multiple I/O commands. If you issue several control commands in succession, specifying delayed spacing or skipping, only the last delayed spacing or skipping option is executed.

Table 7—4. Device Skip Code Table (Part 1 of 2)

Function	Printer Code Substitution			
	0773 and 0778	0770	0768	0776
Skip to code n, m				
n, m = 1 (OV)		Code 12 (OV)	Chan 9 (OV)	Code 12 (OV)
2			Note 6	
3			Note 6	
4				
5				
6				
7 (HP)		Note 2	Chan 15 (Note 4)	Code 7 (HP) Note 2
8	Code 2			
9	Code 1 (OV)	Note 5 (OV)		Code 12 (OV)

Table 7—4. Device Skip Code Table (Part 2 of 2)

Function	Printer Code Substitution			
	0773 and 0778	0770	0768	0776
10	Code 3			
11	Code 4			
12	Code 1 (OV)	Note 5 (OV)	Chan 9 (OV)	Code 12 (OV)
13	Code 5			
14	Code 7 (HP)	Code 7 (HP)	Chan 15 (Note 4)	Code 7 (HP)
15	Code 7 (HP)	Code 7 (HP)	Chan 15 (Note 4)	Code 7 (HP)

LEGEND:

OV Overflow code or channel

HP Home paper code

NOTES:

1. A blank in the code substitution column indicates that no substitution is made; data management skips to the code you have specified.

2. Code 7 must be used as the home paper code on the 0770 and 0776 printers.

3. A skip to code 7 control causes a skip to the home paper code on all printers. On the 0768 printer, the skip is to channel 14 or 15 (skip to channel 15 is issued to the printer). On the 0768 printer, the home paper code used on the tape loop sets the number of lines printed per inch. Channel 14 results in 6 lines/inch line spacing and channel 15 results in 8 lines/inch spacing.

4. A skip to channel 15 issued to the 0768 printer causes an advance to the home paper position, regardless of whether channel 14 or 15 is punched in the paper-tape forms control loop.

5. Code 12 is the primary forms overflow control code for the 0770 printer. Code 9 can also be detected as forms overflow code, using the PRTOV macro. If the PRTOV macro is not used, however, code 12 should be used as overflow code, and code 9 should not be placed in the VFB.

6. Data management accomplishes spacing greater than three lines on the 0768 printer by issuing a number of I/O commands. If the user issues several control commands in succession, specifying delayed spacing or skipping, only the last delayed spacing or skipping option is executed.

7. For the 0768 printer, the software (physical IOCS) provides codes 2 and 3.

PRTOV

7.4.4. Print Overflow Action (PRTOV)

The PRTOV macroinstruction specifies the action to be taken when a forms overflow condition occurs.

Format:

LABEL	Δ OPERATION Δ	OPERAND
[name]	PRTOV	$\left\{\begin{matrix}\text{filename}\\(1)\\1\end{matrix}\right\}\left[,\left\{\begin{matrix}9\\12\end{matrix}\right\}\right]\left[,\left\{\begin{matrix}\text{over-flowname}\\(0)\\0\end{matrix}\right\}\right]$

Positional Parameter 1:

filename
Is the label of the corresponding DTF macroinstruction in the program.

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

Positional Parameter 2:

9
Indicates that forms overflow is to be indicated by channel 9.

12
Indicates that forms overflow is to be indicated by channel 12 (0770 printer only).

Positional parameter 2 is ignored for all printers except the 0770 printer. If omitted, channel 9 is assumed for the 0768 printer, and channel 1 is assumed for the 0773 and 0778 printers. Channel 1 is the only overflow code recognized by the 0773 and 0778 printers. Channel 12 is the only overflow code recognized by the 0776 printer.

Positional Parameter 3:

overflowname
Is the label of your overflow routine. When overflow occurs and this option is specified, the logical IOCS transfers control to this address.

(0) or 0

Indicates that register 0 has been preloaded with the address of your overflow routine.

If omitted, the logical IOCS provides an automatic skip to home paper in the paper tape loop.

Examples:

	LABEL (1)	OPERATION (10)	OPERAND (16)
	DM17	PUT	FILE1,WRKA
1.		PRTOV	FILE1,,OVFLO
		CNTRL	FILE1,SP,5
2.		PRTOV	FILE1
		PUT	FILE2,WRKA
3.	DM19	PRTOV	FILE2,9
4.		PRTOV	FILE2,12,OVFLOB
	DM20	CNTRL	FILE3,SP,,2
		PUT	FILE3
5.		PRTOV	FILE3,9

1. If an overflow condition occurred on a previous PUT or immediate CNTRL macroinstruction execution on file FILE1, branch to routine OVFLO. Overflow codes detected are:

   0773 printer — code 1
   0770 printer — code 12
   0768 printer — code 9
   0776 printer — code 12
   0778 printer — code 1

2. If an immediate overflow condition occurred on a previous PUT or immediate CNTRL macroinstruction execution issued on FILE1, skip to home paper position.

(Lines 3 and 4 of the example show selective overflow detection, which is possible only on the 0770 printer.)

3. If an overflow code 9 was detected on a previous PUT or immediate CNTRL macroinstruction execution, skip to the home paper position.

4. If an overflow code 12 (0770 printer only) was detected on a previous PUT or immediate CNTRL macroinstruction execution, branch to routine OVFLOB.

5. If an overflow code is detected, a skip to home paper position will occur.

Programming Considerations:

When you use the PRTOV macroinstruction in your program, you must specify the PRINTOV=YES keyword parameter in the DTF macroinstruction for the file. The PRTOV macroinstruction can be used with the 0770 printer to selectively check for a code 9 or code 12 forms overflow condition.

When the PRTOV macroinstruction is used, the logical IOCS performs either a skip to the home paper channel or a branch to your forms overflow routine when a forms overflow condition is detected on the preceding print or space command. The forms overflow code is not recognized during a skip operation.

OS/3 data management executes one user-issued CNTRL or PUT macroinstruction after the immediate CNTRL or PUT instruction on which forms overflow is detected. If the imperative macro issued after the macro on which overflow was detected results in a forms advance to a skip code (VFB or paper tape code), your overflow options will not be executed.

The PRTOV macroinstruction may be issued after each CNTRL or PUT macroinstruction, or, it can be issued after only a PUT macroinstruction. It is also permissible to issue a CNTRL, PUT, PRTOV macroinstruction sequence.

If an overflow routine is specified, the printer carriage is not automatically restored to the home paper position.

The address of the instruction after the PUT macroinstruction, which detected the overflow condition, is stored in register 14. If imperative macroinstructions are issued in your overflow routine, register 14 should be stored before issuing the instruction, and restored after execution of the instruction.

CLOSE

7.4.5. Close a Printer File (CLOSE)

Function:

The CLOSE macroinstruction is issued when all the data in a file has been processed. This macroinstruction calls a transient routine which checks for errors that may have occurred in the final output operation and then prevents further access to the file.

Format:

LABEL	△OPERATION△	OPERAND
[name]	CLOSE	{ filename-1[,...,filename-n] / (1) / 1 / *ALL }

Positional Parameters:

filename
Is the label of the corresponding DTF macroinstruction in your program. The maximum number of filenames is 16.

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

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

Examples:

	LABEL (1)	△OPERATION△ (10)	OPERAND (16) △
1.	DM87	CLOSE	PRNT2,PRNT4
2.		LA	1,PRNT1
	DM88	CLOSE	(1)

1. Close print files labeled PRNT2 and PRNT4.

2. Close print file labeled PRNT1.

7.5. ERROR AND EXCEPTION HANDLING

7.5.1. FilenameC

When certain errors or exceptions to file processing performance are detected by OS/3 data management, it will make appropriate entries in specific fields of the DTF file table, which your program may address in order to learn of these conditions and take the proper course of action on regaining control. One such field in the DTFPR file table is *filenameC,* a 1-byte field which you may access by concatenating the character C to your 7-character file name and using the assembler language *test-under-mask* (TM) instruction.

Refer to Appendix B for the meanings of the bits in *filenameC* of the DTFPR file table which are set to binary 1 by OS/3 data management for certain error and exception conditions.

7.5.2. Truncation of Print Line

When OS/3 data management detects that the execution of a PUT macroinstruction has resulted in the printing of a truncated line, it sets the *line truncated* flag in *filenameC* (bit 0, byte 0) (Appendix B). However, it does not pass control to your error routine until after the printer file has been closed, by which time other truncated lines may have also been printed. Depending on your requirements, you may find it useful in your program to test for setting of the *line truncated* bit after each execution of the PUT macro and provide for a flag character to be printed with the next line to speed visual location of all truncation errors on your printout.

7.6. SAMPLE PROGRAM

The following sample program, constructed to illustrate a typical use of the OS/3 data management printer system in a BAL program, also indicates where to place the OS/3 job control statements needed to implement it.

```
LABEL     OPERATION     OPERAND                    COMMENTS
1         10            16                         72                    80
//JOB      EXAMPLE
          .
          .             JOB STREAM TO EXECUTE ASSEMBLER
          .
          .
/$
EX1       START         0
*
*         EX1           EXAMPLE OF THE USE OF THE DATA MANAGEMENT PRINTER
*                       SYSTEM.
*
          BALR          9,0                        LOAD COVER REGISTER
          USING         *,9
          LA            13,SAVE                    REG SAVE AREA ADDR TO R13
          OPEN          OUT1                       OPEN PRINTER FILE OUT1
          CNTRL         OUT1,SP,,2                 DELAYED SPACE OF 2 LINES
          PUT           OUT1,WORK1                 PRINT HEADING & SPACE 2 LINES
          LA            5,25                       NUMBER OF LINES TO REG 5
E1        EQU           *
*         .                          CODE TO GENERATE 25 PRINT LINES
*         .
*         .
*         .
          PUT           OUT1,WORK2                 PRINT & SPACE 1 LINE
          BCT           5,E1                       IF NOT DONE BRANCH TO E1 FOR
*                                                  NEXT PRINT LINE
          CNTRL         OUT1,SK,7                  SKIP TO HOME PAPER
          CNTRL         OUT1,SP,,3                 DELAYED SPACE OF 3 LINES
          PUT           OUT1,WORK3                 PRINT HEADING & SPACE 3 LINES
          PUT           OUT1,WORK4
```

```
LABEL    OPERATION   OPERAND                  COMMENTS
1        10          16                       72        80
         PUT         OUT1,WORK5               PRINT SUMMARY OF DATA
         PUT         OUT1,WORK6
         CLOSE       OUT1                     CLOSE PRINTER FILE OUT1
         EOJ                                  END OF JOB
*
*        ERROR ROUTINE
*
ERR      CANCEL                               CANCEL JOB
*
*        DATA STORAGE AREA
*
SAVE     DS          18F                      REGISTER SAVE AREA
IO1      DC          CL120' '                 I/O AREA 1
WORK1    DC          CL120'FIRST PAGE HEADING'
WORK2    DC          CL120'DATE:          WIDGETS PRODUCED:          GIDGETS PRODUCX
                     ED:'
WORK3    DC          CL120'SUMMARY OF DATA'
WORK4    DC          CL120'TOTAL WIDGETS PRODUCED:'
WORK5    DC          CL120'TOTAL GIDGETS PRODUCED:'
WORK6    DC          CL120'CURRENT INVENTORY:  WIDGETS=        GIDGETS='
OUT1     DTFPR       BLKSIZE=120,CONTROL=YES,ERROR=ERR,IOAREA1=IO1,PRAD=1,   X
                     PRINTOV=SKIP,RECFORM=FIXUNB,WORKA=YES
         END         EX1
/*
                     JOB STREAM TO EXECUTE LINKAGE EDITOR
```

LABEL (1)	OPERATION (10)	OPERAND (16)	COMMENTS (72–80)
// DVC	20,P	// LFD OUT1	ASSIGNMENT OF PRINTER TO FILE
		JOB STREAM TO EXECUTE PROGRAM	
/&			
// FIN			

PART 4. DISK FILES

10. ISAM Formats and File Conventions

10.1. GENERAL

The *indexed sequential access method* (ISAM), one of the five methods that OS/3 data management provides for handling your disk files, may be used with all available disk subsystems. These are: SPERRY UNIVAC 8411 Disk Subsystem; SPERRY UNIVAC 8414 Disk Subsystem; SPERRY UNIVAC 8415 Disk Subsystem; SPERRY UNIVAC 8416 Disk Subsystem; SPERRY UNIVAC 8418 Disk Subsystem; SPERRY UNIVAC 8424 Disk Subsystem; SPERRY UNIVAC 8425 Disk Subsystem; SPERRY UNIVAC 8430 Disk Subsystem; and SPERRY UNIVAC 8433 Disk Subsystem. The 8415, 8416, and 8418 disk subsystems are fixed-sector disks, on which data records are written in fixed physical sectors of 256-byte lengths. All data transfers must be multiples of this length. The others are variable-sector disks. Each OS/3 data management processing module handles all disk types, in order to give you as much device-independence as possible. You seldom need to be concerned with most of the details of the way these disk subsystems operate. Those functional characteristics you will find useful, however, are presented in Appendix A. ISAM does not support the 8413 diskette.

ISAM is the only method that gives you the capability of building a hierarchy of index blocks to support a search of your files by key; its *search-by-key* function allows you to perform random retrieval in a relatively short time. In OS/3 ISAM, a *key* is a character string that you specify within each logical record, to uniquely identify that record. Its minimum length is 3 bytes; its maximum is 253. One restriction on the content of keys is that no byte of any key may contain the hexadecimal value FF (11111111 in binary). A key that contains FF_{16} may produce erratic results during retrieval by key. Another restriction on the content of keys is that you may not have a key constituted entirely of binary 0's; this key is reserved for a dummy record that data management creates and inserts at the start of every ISAM file. The reasons for these restrictions on key size and content are developed later in the manual.

As in other OS/3 data management systems, a *logical record* is simply what you tell data management it is — a character string that you define and that data management handles as an entity in storage and retrieval.

To use ISAM's *search-by-key* function, you must first present your records serially with keys and load them sequentially onto disk. You present your records to data management, unblocked and one at a time, in ascending order of these keys. Data management blocks your initial records into an area of your file, termed the *prime data area*, and builds the index structure used to effect random retrieval by key. After initially creating your ISAM file, you may add new records, presented in any key order. OS/3 ISAM does not place these in the prime data area; it places your new records in an *overflow area* of your file and logically chains them to the proper points in the pre-existing series of records, maintaining the effect of one long, orderly series. (Other ISAM systems insert new records on the original prime data tracks, and existing records are pushed down; OS/3 avoids this inefficient process.)

You may retrieve and update your records in any of the following ways:

- sequentially, progressing from any record toward the end of the series;
- randomly, by presenting a key; or
- randomly, by presenting a relative address.

Another significant point of difference between this and other similar access methods is that OS/3 ISAM also allows you the option of side-stepping the formation of the ISAM index. When you do so, your records need not be keyed, and the key-based functions of the ISAM repertoire are inoperative, yet you retain full abilities for sequential access, direct access, and inserting records.

The advantages of using OS/3 ISAM without an index (ASAM) are these:

- Although your ASAM file is, in effect, a sequential file, under ISAM you have direct addressing capabilities that are not available to you under the OS/3 sequential access method (SAM).
- Your ability to add records to an ISAM file gives you the capability of building a header-trailer file and forming trailers dynamically.

Another advantage of OS/3 ISAM over earlier systems is that a program referencing several disk files, one of which is an ISAM file with index, may employ the same processing module to reference the other files; that is, you may use an OS/3 ISAM module to process ASAM files sequentially or directly.

Usually, an OS/3 ISAM file will be one of several files occupying the same disk pack. Every pack is provided with a *volume table of contents* (VTOC), which facilitates system control of disk space and file location. (The VTOC and the system standard disk labels used in common by all OS/3 data management access methods are described in Appendix D of this manual. But it is worth noting at this point that ISAM does not support user labels for disk files.)

10.2. ISAM FILE ORGANIZATION

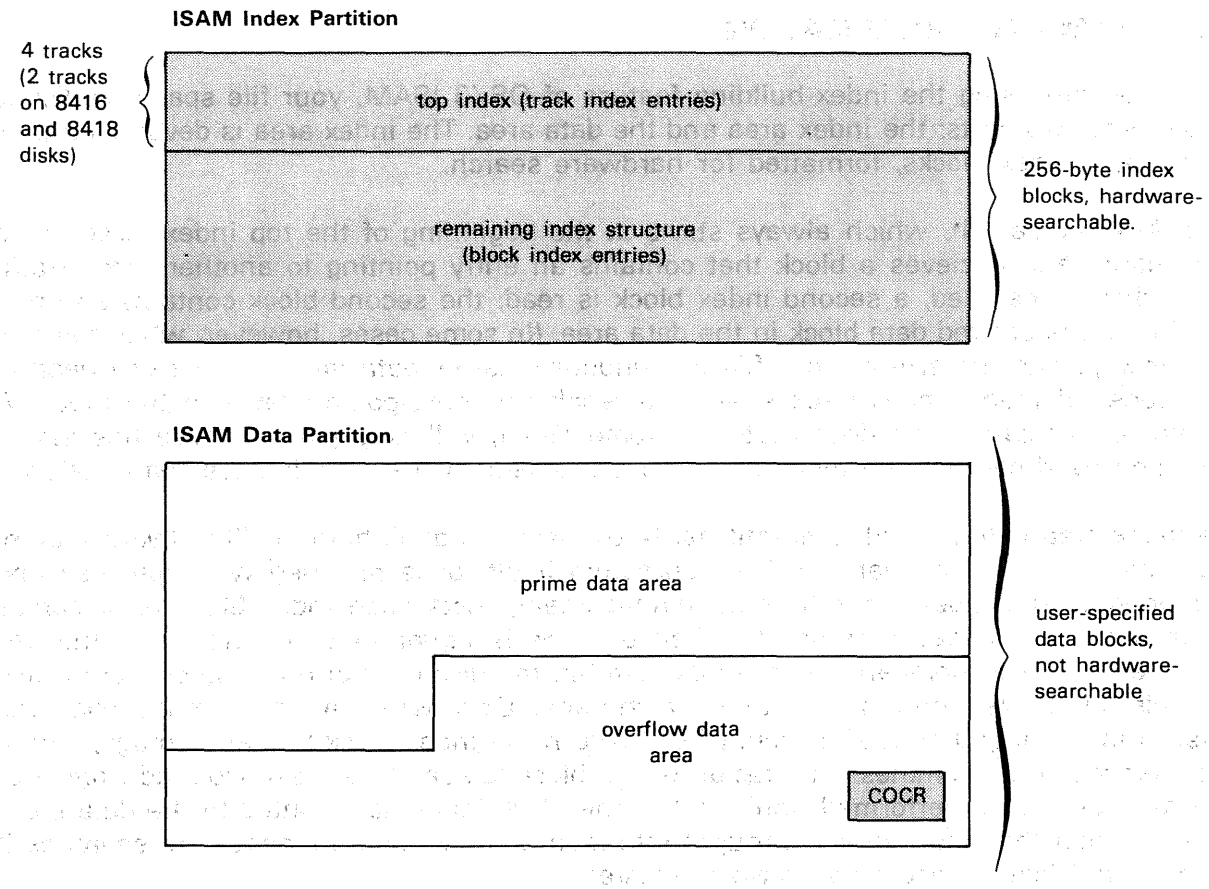
When you are using the index-building feature of OS/3 ISAM, your file space on disk is divided into two parts: the index area and the data area. The index area is devoted entirely to 256-byte index blocks, formatted for hardware search.

An ISAM key search, which always starts at the beginning of the top index, executes a disk search and retrieves a block that contains an entry pointing to another index track. When this is searched, a second index block is read; the second block contains an entry pointing to the desired data block in the data area. (In some cases, however, when the size of your keys and the size of your file are unusually large, data management will need to make one additional index-track search to reach this data-pointer level in the index. A number of measures, developed later in some detail, will help you minimize this index-searching overhead; for the moment, however, remember that you do have some options.)

The index area is formatted to permit hardware search, equal/high. By "hardware search, equal/high", we mean that the disk subsystem itself, once provided with your key and positioned to the index track by data management, tests each index block as it comes under the head to see whether the key on disk is equal to or greater than the key presented to it. Key comparisons are made within the disk control unit and do not involve transmission of key information to main storage. Only when a hit is made does the physical input/output control system (IOCS) return the index block to main storage, where data management examines it to decide which block to search for next. No read from your data area on disk is performed until a hit in the block index has pointed to the data block sought. When this data block is brought into main storage, data management searches it there for the logical record you have requested.

This exploitation of the hardware search capability of the disk subsystems makes for speed, but a hardware search of the entire index can be avoided when you place part of it in main storage. This point is developed further in 10.2.4 and 10.2.5. See also the discussion of the INDAREA keyword parameter (11.4.5).

The data area of your file contains only data bocks, which are not formatted for hardware search, and cylinder overflow control records (COCR). The data blocks are either *prime* data blocks filled during your initial file load, or *overflow* data blocks, filled by your subsequent additions to the file. These blocks are identical in form; the only difference is their location, in either the prime data area or the overflow area of the cylinder. You will specify the percentage of each cylinder of your file that data management is to reserve for overflow when you first design and describe the file, using a certain keyword parameter in the declarative macro that defines it to OS/3 ISAM. Figure 10—1 shows the two partitions of an indexed OS/3 ISAM file.



LEGEND:

COCR Cylinder overflow control record; written by data management in last block of cylinder and points to the location of remaining overflow space on this cylinder.

Supplied by OS/3 data management.

Data supplied by you; you also specify to OS/3 data management the percentage of cylinder area that is to be assigned to receive overflow records. The breakpoint between prime and overflow need not fall at a track boundary.

Figure 10—1. The Two Partitions of an Indexed OS/3 ISAM File: Cylinder Formats of the Index Partition and the Data Partition

Because it is unlikely that you will use the overflow areas at the same rate in all your cylinders, some may be filled while others are hardly touched; this is where the COCR comes in. When a cylinder's overflow area can take no more, OS/3 ISAM will place records ordinarily destined for that cylinder onto other cylinders having overflow space available. The COCR is in the last data block of the cylinder, and is used to keep track of available overflow space.

During its original loading of your file into data blocks on disk, data management inserts a 5-byte data pointer field after each prime data record. It will use these fields later, whenever you present new records for insertion, to set up the required logical sequence and to keep it current. (You have a use for this data pointer, too, for retrieval of records without keys; this point is developed later.)

New records are never actually *inserted*, although a rewritten record, which is an update of a record already on disk, is written back into the location of the original. As stated previously, data management places new records in the overflow area and chains them into logical sequence. All records remain at the locations where they were originally placed, making it easy for you to point to the records of one ISAM file from the records of some other file. Because an updated record is always written back to its original spot, you must never alter either the original key or record length of a record when you update it.

The most significant three bytes of the 5-byte data pointer contain the record's block number, relative to the start of the data area; its least significant two bytes contain the byte position of the record within this block. You may address all ISAM data records directly by their 5-byte relative addresses of this format, which you will express in binary form.

10.2.1. ISAM Record Formats

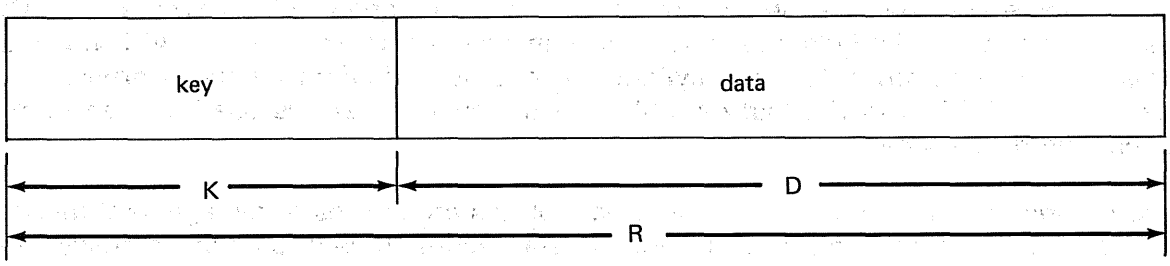
Your OS/3 ISAM file may contain either fixed-length or variable-length blocked records — this is a difference between OS/3 and some earlier systems, which allowed only fixed record lengths. You present your records one by one, and ISAM blocks them.

If your records are variable, you must insert into the leading two bytes of every logical record the length of that particular record in binary form. Fixed records do not require this 2-byte field. (It is worth noting, if you are familiar with other systems using a 4-byte record length field at the head of each variable record and reserving two of these bytes for system use, that OS/3 ISAM does not do so — only two bytes are used for record length; the rest of the record is yours).

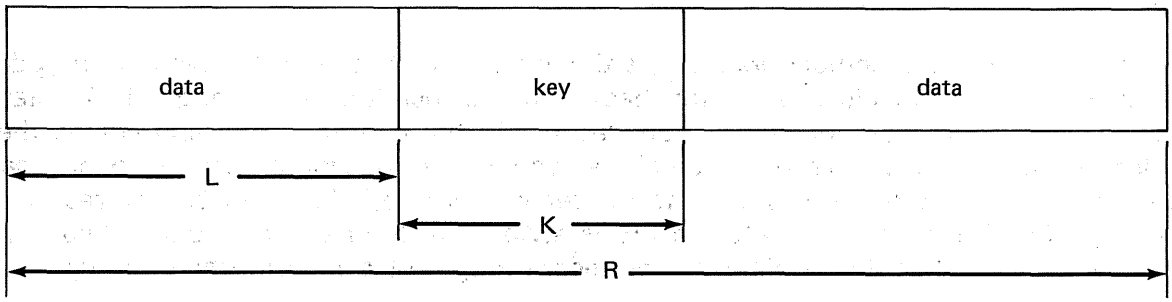
When you submit *keyed* records to OS/3 ISAM (which you must do when you are using the index feature and *may* do even when you are not), all logical records in the file must have keys. Each key must be unique in content, equal in size to the other keys, and must be located at the same distance from the start of its record. As noted before, a key may always be embedded in a record; its length may not be less than 3 bytes nor exceed 253 bytes.

Figures 10—2 and 10—3 show the formats of OS/3 ISAM logical records, keyed and unkeyed.

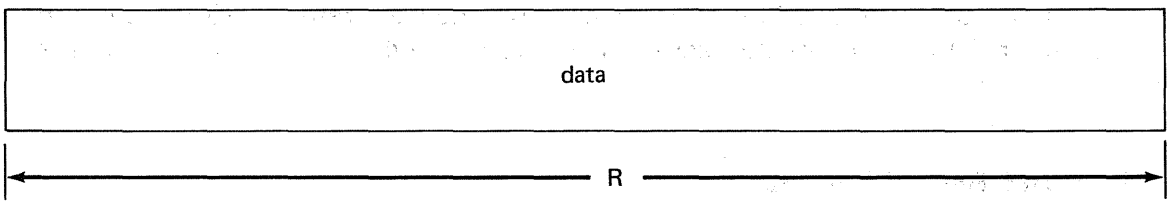
Key at Head of Record



Key Internal to Record



Without Key



LEGEND:

K Record key. All keys in a keyed file must have the same length; each record in a keyed file must have one (and only one) unique key; and the starting location of the key must be the same in each record. You specify the length of the key with the KEYLEN keyword parameter; minimum length is 3 bytes, maximum is 253.

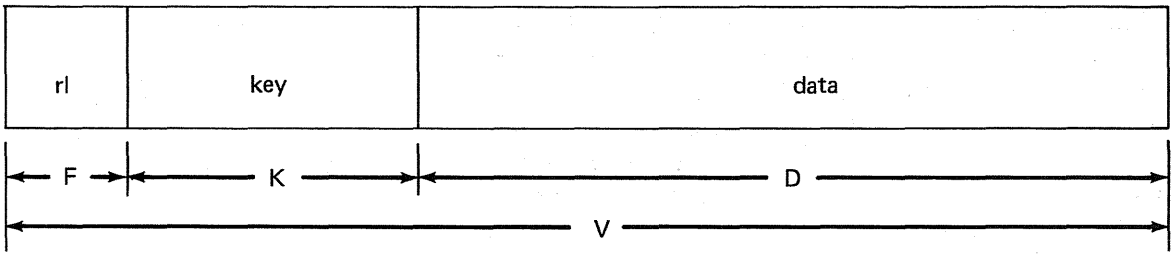
L Key location. The starting location of the key must be the same in each record. You may specify the number of bytes of data preceding the key with the KEYLOC keyword parameter. If keyword is omitted, ISAM assumes the key starts in the first byte of a fixed record.

D Data portion of your logical record

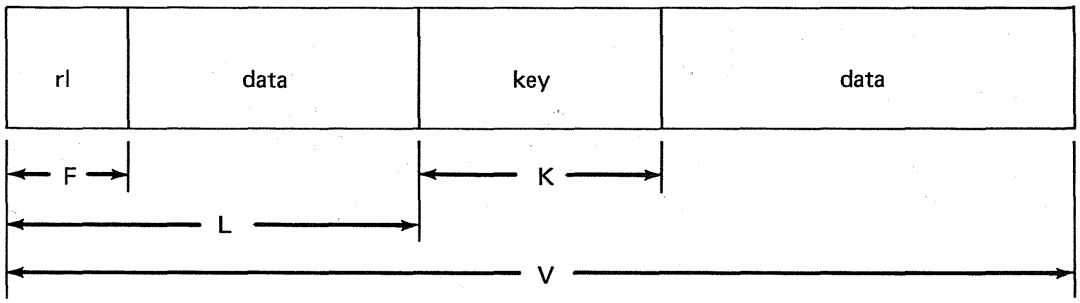
R Length of logical fixed-length record (key plus data). You specify this length, measured in bytes, with the RECSIZE keyword parameter; it must never exceed the value of data block size, less seven bytes.

Figure 10—2. Fixed-Length ISAM Records with and without Keys

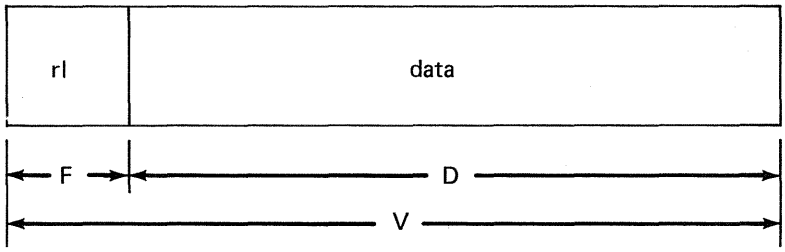
Key at Head of Record



Key Internal to Record



Without Key



LEGEND:

F A 2-byte record length field. You insert the length of each variable record into this field in binary; the length includes this 2-byte field and is equivalent to V in this figure.

D Data portion of your logical record

V Length of a variable record. Includes key plus data, plus two bytes for the record length field. You never specify this length with a keyword parameter but place it in the leading two bytes of each variable record (in the field represented by F in this figure).

K Record key. All keys in a keyed file must have the same length; each record in a keyed file must have one (and only one) unique key; and the starting location of the key must be the same in each record. Minimum key length is 3 bytes; the maximum is 253. You specify the length of the key with the KEYLEN keyword parameter.

L Key location. The starting location of the key must be the same in each record. You may specify the number of bytes that precede the key with the KEYLOC keyword parameter. If you omit the keyword, ISAM assumes the key begins in the third byte of a variable record.

Figure 10—3. Variable-Length ISAM Records with and without Keys

10.2.2. ISAM Data Block Format

When OS/3 ISAM writes your records to the disk, it blocks them into *data blocks,* the size of which you specify. (This data block length is the same as the length of your I/O area, or data buffer, and may contain unused space, for reasons discussed later.) All data blocks begin with a 2-byte field called the *block header,* which states the number of bytes in the block that are currently occupied. Because every logical record is accompanied by the 5-byte data pointer just mentioned, the maximum size of any logical record is thereby limited to data block size minus *seven* bytes. And, because the effective data length of a variable record is further reduced by its own 2-byte record length field, the number of bytes of data that your longest variable record may contain is no more than data block size less *nine* bytes.

Although you should use as large a block as you can afford to have in main storage, data buffer size is not entirely up to you. It may never be less than 256 bytes, for example, because this is the length of the index blocks, and ISAM handles these through the I/O buffer (whose length you specify when you specify data block size to data management). Furthermore, you should set data block size at some multiple of 256 bytes if you are using the fixed-sector 8415, 8416, or 8418 disks at your installation. If you do not specify a multiple of 256 bytes, ISAM increases your specification to the next higher multiple. If your data block size is less than 256 bytes, then you must specify a BLKSIZE length of 256 bytes or more. Finally, whether you are using this or the variable-sector 8411, 8414, 8424, 8425, 8430, or 8433 disks, the block size you specify must not exceed the track size for these devices:

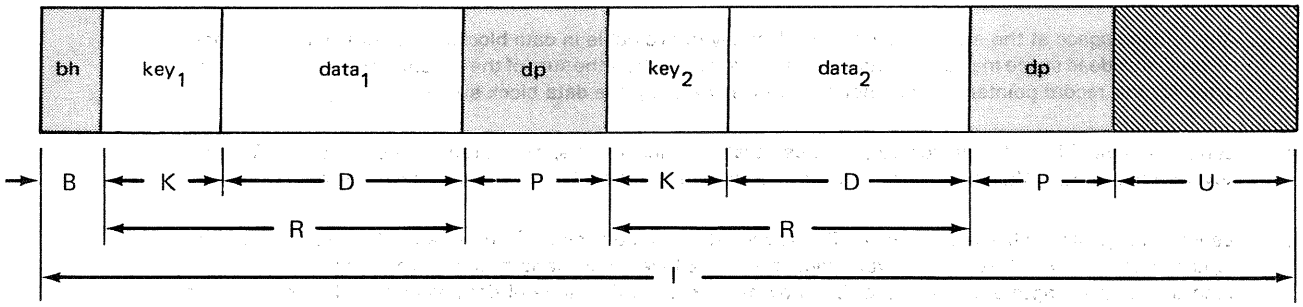
SPERRY UNIVAC Disk Subsystem	Track Size, in Bytes
8411	3625
8414	7294
8415	10,240
8416	10,240
8418	10,240
8424	7294
8425	7294
8430	13,030
8433	13,030

Figure 10—4 illustrates the layout of two ISAM data blocks on disk, one containing two fixed-length records, and one containing two variable-length records. The legend relates the segments shown to various keyword parameters you will use in the DTFIS *declarative macroinstruction* to define your file to OS/3 ISAM. (The DTFIS macro and its keyword are described in detail in Section 11.)

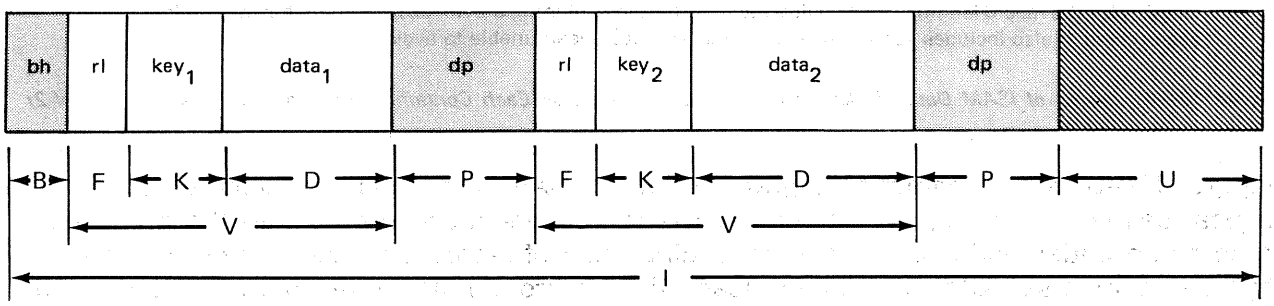
Note that Figure 10—4 shows some unused space at the end of each data block. You should avoid this (if you can) when you specify data block size for a file containing fixed records, by ensuring that the sum of your logical record size, plus five bytes for the record pointer following each record, divides evenly into the block size — not forgetting that block size must always include the 2-byte block header field.

On the other hand, it is generally not possible for you to avoid some unused space in the data block when your ISAM file contains variable records.

Fixed Records



Variable Records



LEGEND:

B Block header, written by data management in the buffer. This is always two bytes long and contains (in binary) the number of bytes in the data block which hold usable information. In the example shown, this figure would equal (I) minus (U); it includes the total space occupied by the records themselves and the 5-byte data pointers, one of which follows each of the records. Because data management appends the block header in the buffer, and it is placed in the buffer with the block when it is retrieved, you must allow for this 2-byte header in calculating the data block size, which you specify with the BLKSIZE keyword parameter. However, it is *not* moved to your record work area (the *address* of which you specify with the WORK1 keyword parameter) where data management presents your records, one by one.

K Record key. This may be internal to the record instead of being located (as shown) at the head of the record. All keys in a file must have the same length; each record in a keyed file must have one and only one unique key; and the starting location of the key must be the same in each record of the file. You specify the starting location of the key with the KEYLOC keyword parameter, and its length with the KEYLEN parameter; minimum key length is 3 bytes and maximum is 253. (When you present a key that you want ISAM to match by search you load it in an area of your program specified by the KEYARG keyword parameter.)

D Data portion of your logical record

F A 2-byte record length field of a variable record

P Record pointer, a 5-byte divider which follows every record that data management writes into the data block. The record pointers are written by data management, but you must allow space for them in calculating your I/O area length, which you specify with the BLKSIZE keyword parameter. Data block size and I/O buffer size may differ; the buffer must be at least the size of the data blocks, but may be greater. The record pointer contains, in binary, the block number and byte position in that block of the next sequential logical record in the file, in the form **rrrbb**, where: **rrr** is the block number, relative to the data partition; and **bb** is the displacement, measured in bytes, of the record into that block (a record preceded by 125 bytes will have a displacement value of 125). You will use this 5-byte "address" when you retrieve records directly, rather than by key. (This address is always returned to you by data management after each record is loaded or added to your file; it is your responsibility to access it and store it for later use, if you plan to use it. Data management places this 5-byte value in the field of your DTF called *filenameH*.)

Figure 10—4. Layout of ISAM Data Blocks (Prime or Overflow) on Disk Each Containing Two Logical Records (Part 1 of 2)

LEGEND (cont):

U Any unused space at the end of a data block. Usually unavoidable in data blocks containing variable-length records, this dead space may also occur in files of fixed records, if the sum of the logical record size (R) plus five bytes for the record pointer (P) does not divide evenly into (I), the data block size you specify.

R Length of logical fixed-length record (key plus data). You specify this, measured in bytes, with the RECSIZE keyword parameter. This must never exceed the value of the data block size, *less* seven bytes.

V Length of logical variable-length record (2-byte record length field, plus key, plus data). You never specify this length with a keyword parameter; you place it in the 2-byte record-length field at the head of each logical record. Like the length of a fixed record, it may never exceed the value of data block size, less *seven* bytes.

I Length of I/O area, which you specify with the BLKSIZE keyword parameter. It includes the 2-byte block header length, plus the record length of each logical record in the block, plus the 5-byte record pointer that must follow each record. It also includes such unused space as you have been unable to avoid.

Figure 10—4. Layout of ISAM Data Blocks (Prime or Overflow) on Disk Each Containing Two Logical Records (Part 2 of 2)

Figure 10—5 is a schematic diagram of OS/3 ISAM's method of chaining records in logical sequence. The upper tier of records represents those placed into the prime data area by an initial load of the file in ascending order of record keys (keys being represented by the numbers). After the initial load, the record pointer that follows each record contained the hexadecimal pattern F0 AA AA AA AA, indicating that the next record in physical sequence was also the next in logical sequence in the file. (The record pointer after the record whose key is 750 still points in this way to the start of the one whose key is 809, for example.) After the lower tier of four records was added (to the overflow area), data management rewrote some record pointers as necessary to chain the new records into their logical sequence. The record pointer following record 827 originally contained the pattern pointing to record 902, which is the next record in sequence in the original file and happens to be the first record in the next data block. After the addition of record 901, data management rewrote this to point to the new record; the new record pointer following 901 is now the one pointing to 902.

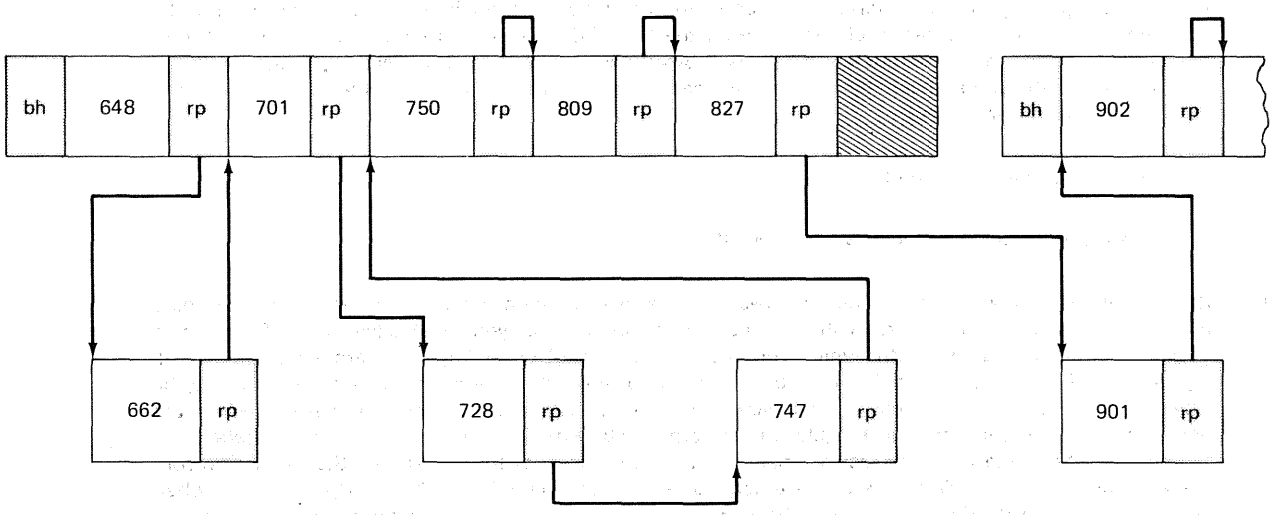


Figure 10—5. Schematic Diagram of ISAM Records Chained into Logical Sequence after Adding Records to the File (Part 1 of 2)

LEGEND:

bh Block header, a 2-byte field at the head of each physical data block. Written by data management to indicate the number of bytes in the block that are devoted to usable data.

rp Record pointer, a 5-byte field inserted by data management following each logical record written to a data block. Originally, after initial load, contains hexadecimal pattern F0 AA AA AA AA, pointing to next sequential record in the file. Rewritten by data management as necessary to point to records "inserted" by addition after initial load.

Logical records submitted by you in ascending order of keys. The top tier represents prime data records submitted by an initial load; the lower tier represents four records submitted by a subsequent operation, placed in overflow, and chained into logical sequence by OS/3 ISAM.

Supplied by OS/3 ISAM

Unused space in physical data block

Figure 10—5. Schematic Diagram of ISAM Records Chained into Logical Sequence after Adding Records to the File (Part 2 of 2)

10.2.2.1. Calculating Space Requirements for the File

To calculate the number of cylinders required for data in an ASAM or ISAM file, you may proceed as follows:

1. Calculate r, the number of logical records per data block (r is an integer):

$$r = \frac{\text{(block size)} - 2 \text{ bytes}}{\text{(average record size)} + 5 \text{ bytes}}$$

2. From this, calculate b, the number of prime data blocks required for the initial ISAM load:

$$b = \frac{\text{number of records to be loaded}}{r}$$

3. Then calculate d, the total number of data blocks (prime plus overflow):

$$d = \frac{100\ b}{100 - \text{(percent overflow)}}$$

4. Calculate the cylinder capacity of the disk subsystem you are using:

cylinder capacity = (number of surfaces per disk unit) (track capacity)

The number of surfaces and the track capacity for the various disk subsystems are shown in Table A-4.

5. Calculate the number of data blocks of the desired size each cylinder can hold:

$$\text{number of data blocks each cylinder can hold} = \frac{\text{cylinder capacity}}{\text{block size}}$$

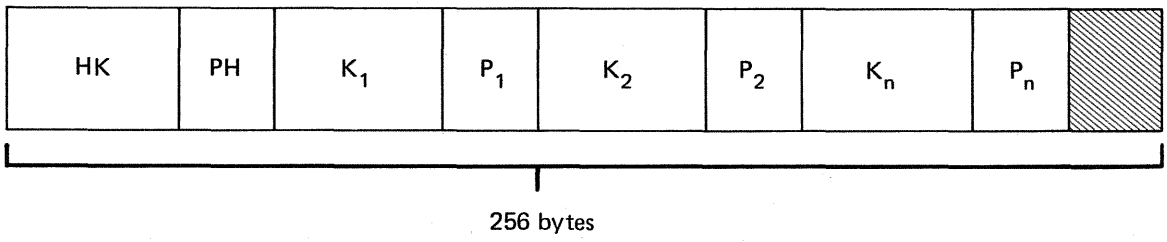
6. Finally, convert d into C_d, the number of cylinders required for data:

$$C_d = \frac{d}{\text{(number of data blocks each cylinder can hold)}}$$

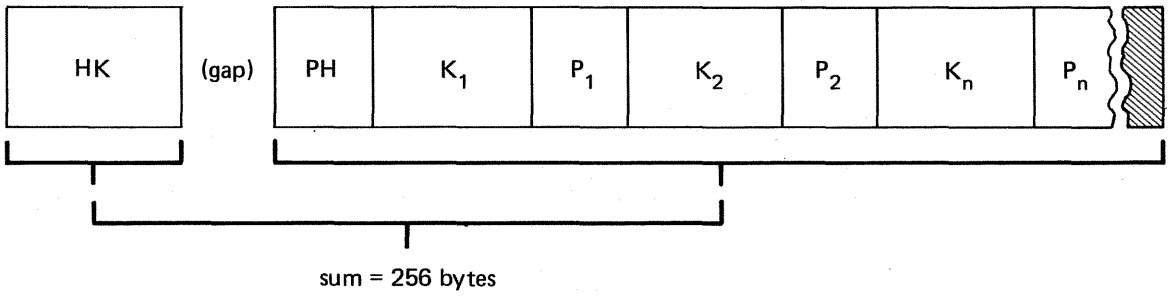
10.2.3. ISAM Index Blocks

The index area of your ISAM file is, as we stated previously, entirely devoted to 256-byte index blocks, which are written for you on the index tracks by OS/3 data management. The format of these blocks is shown in Figure 10—6; they contain keys and 3-byte pointers to the prime data blocks and are hardware-searchable.

ISAM Index Block on a Fixed-Sector 8416 Disc



ISAM Index Block on a Variable-Sector Disc



LEGEND:

HK High key of the block.

PH Pointer following the high key of the block — points to next level below

P A 3-byte pointer to next level below

Unused space in index block

$K_1, K_2, \ldots Kn$ are ascending keys, < HK

Figure 10—6. Format of Full OS/3 ISAM Index Blocks

Note that all the entries are fixed-length and that the index block has no block header field. If the final block of an index level happens to be a full block, it will have the form shown in Figure 10—6, and the high key of the block will contain all 1 bits. If the final block is *not* full, its form will differ in that the top entry with a key of all 1 bits will be repeated between the last record pointer and the unused space. As a result, a scan of a final index block in main storage will search only valid information; there is no possibility of its running on into the spurious information contained in the unused space.

Each scan of index blocks in main storage begins at the K_1 position. In the ordinary full block, the result may be a hit on or before K_n; if it is a miss, the search uses the PH pointer to locate the next index block to scan.

Note that, like the ISAM data blocks, the index blocks may also contain unused space if the sum of the key length (measured in bytes and uniform for every key in the file), plus three bytes for the track or block pointer that follows each key, does not divide evenly into 256. If your keys are 40 bytes long, for example, you can fit only five of them (and their accompanying 3-byte pointers) into 256 bytes and will have 41 unusable bytes left over. If you are designing a file, therefore, and have some latitude in choice of key length, it is wiser not to establish it arbitrarily, but to choose a key length that will give you the least wasted space in the index blocks. Figure 10—7 recapitulates much of what has been discussed so far: it shows the layout of an index partition and a data partition of an indexed ISAM file as they appear on disk:

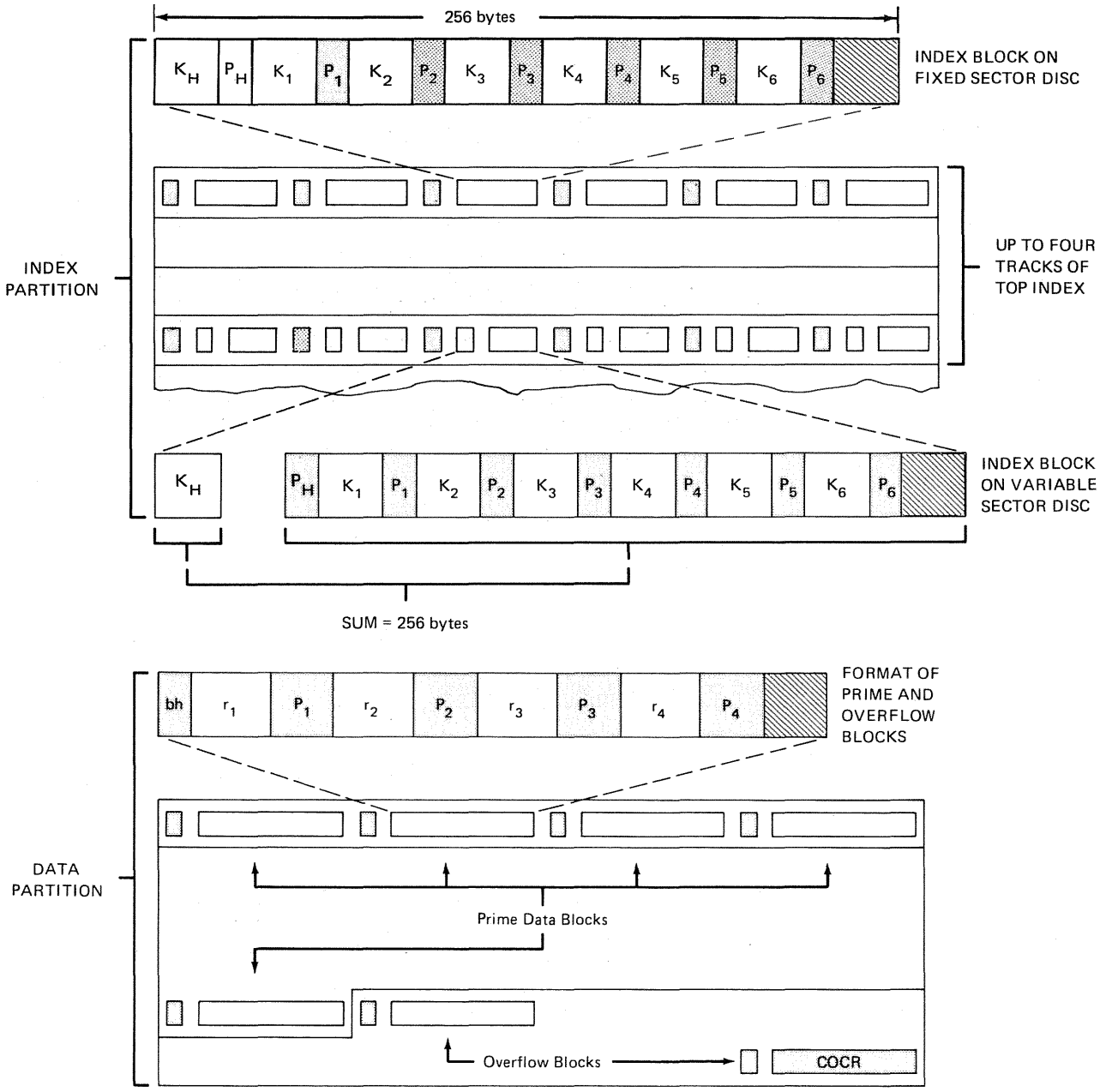


Figure 10—7. OS/3 ISAM File Structure (Part 1 of 2)

LEGEND:

K Record key

P Pointer

bh Block header

r Logical record

Unused space at block end

COCR Cylinder overflow control record; written by data management

System-supplied pointers, headers, and count fields on disk

Figure 10—7. OS/3 ISAM File Structure (Part 2 of 2)

10.2.4. Calculating Space for the ISAM Index Area

You may calculate your disk space requirements for the block index of your ISAM file by the process shown below. Because the space needed to hold your index is inversely proportional to the size of your data blocks, you should favor larger over smaller data blocks when you design your file, to avoid excessive index space.

→ To calculate the number of cylinders (C_i) that your block index will need, you start with your block size, the average size of your records (if they are variable), the total number of records in your file, and the size of keys in the file. These are the five steps we suggest you take; they can, of course, be compressed into one calculation:

1. Calculate r, the number of logical records per data block (r is an integer, naturally):

$$r = \frac{\text{blocksize} - 2}{\text{average record size} + 5}$$

2. Calculate b, the number of prime data blocks you require:

$$b = \frac{\text{total number of records to be loaded}}{r}$$

3. Calculate e, the number of entries data management will make per index block (e is also an integer):

$$e = \frac{256}{\text{keysize} + 3}$$

4. The next calculation gives i, the number of index blocks you will have:

$$i = \frac{b}{e}$$

5. Dividing this by the number of index blocks each cylinder may hold (a constant depending on the disk subsystem) gives C_i, the number of cylinders your block index will require:

$$C_i = \frac{i}{\text{number of index blocks per cylinder}}$$

These are the numbers of 256-byte ISAM index blocks that each cylinder may hold on the disk subsystems used by OS/3:

SPERRY UNIVAC Disk Subsystem	Number of 256-byte Index Blocks per Cylinder
8411	100
8414	340
8415 fixed	120
8415 removable	80
8416	280
8418	280
8424	340
8425	340
8430	551
8433	551

The foregoing calculations hold good for the *second* level of your ISAM index: the block index. For a useful approximation of the disk occupied by the entire index (when there is no intermediate index), you need add only the number of tracks that ISAM sets aside for the first level, which is the top index: two tracks for an ISAM file residing on an 8415, 8416, or 8418 fixed sector disk, and four tracks for the variable sector 8411, 8414, 8424, 8425, 8430, and 8433 disks supported by OS/3.

The calculation of the amount of disk space *actually* occupied by your top index (without the repetition pattern mentioned in 10.2.3) is more complex. Of course, you can readily figure the maximums:

Disk Subsystem on Which ISAM File Resides	Maximum Number of Tracks for Top Index	Track Capacity in 256-byte Top Index Blocks	Maximum Number of Bytes for Top Index
8411	4	10	10,240
8414	4	17	17,408
8415	2	40	20,480
8416	2	40	20,480
8418	2	40	20,480
8424	4	17	17,408
8425	4	17	17,408
8430	4	29	29,696
8433	4	29	29,696

However, these maximums are rarely reached in practice; few files require more than 3K bytes for the top index.

The most convenient procedure for learning the size of the top index is to access *filenameS* after issuing the ENDFL imperative macro that terminates the initial file load or each subsequent extension of it (11.5.2.3). ISAM places the number of bytes required to hold the top index of a file in main storage in this 2-byte field, which is half-word aligned and addressed by concatenating the character "S" to your 7-byte file name. This is the figure you need to know for subsequent random operations on your file, because you can improve the speed of your keyed search operations (for add and retrieval) by providing main storage space for some or all of the top index.

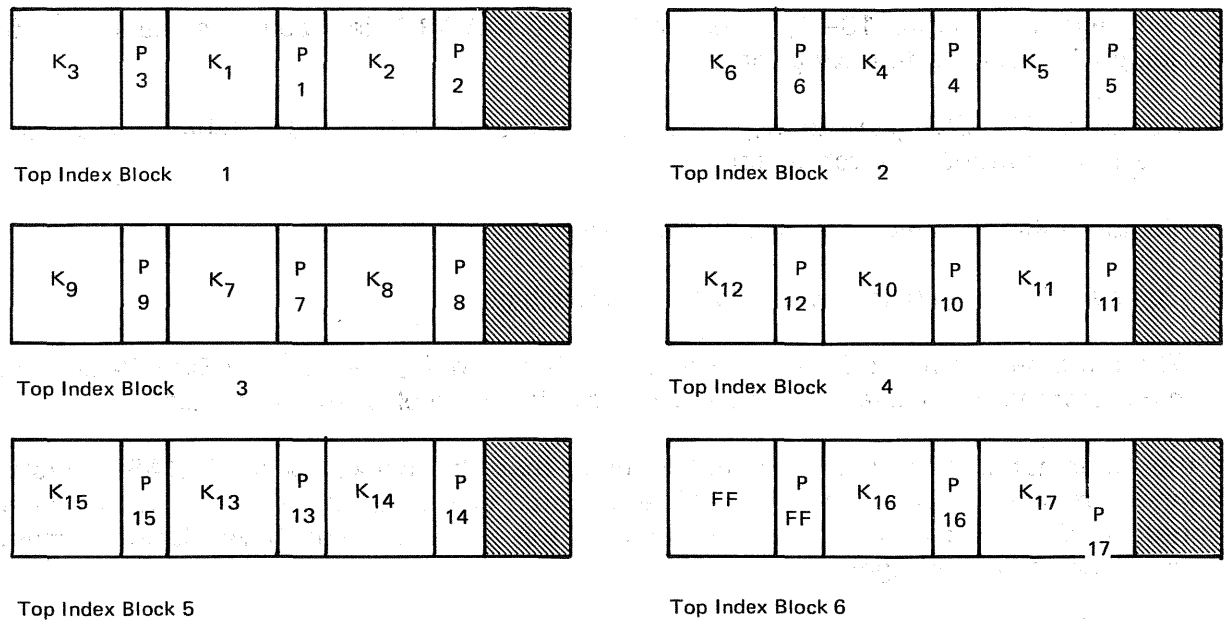
10.2.5. Loading the Top Index into Main Storage

To speed random retrieval or record insertion operations in an ISAM file, you may specify that data management is to place as much as possible of your top index in the index buffer (INDAREA, 11.4.5) for subsequent search there. Doing so minimizes the hardware search of this part of your index on disk; if you can allow enough space in main storage for the *entire* top index, a search of it on disk is eliminated altogether. Only top index entries may be brought into main storage.

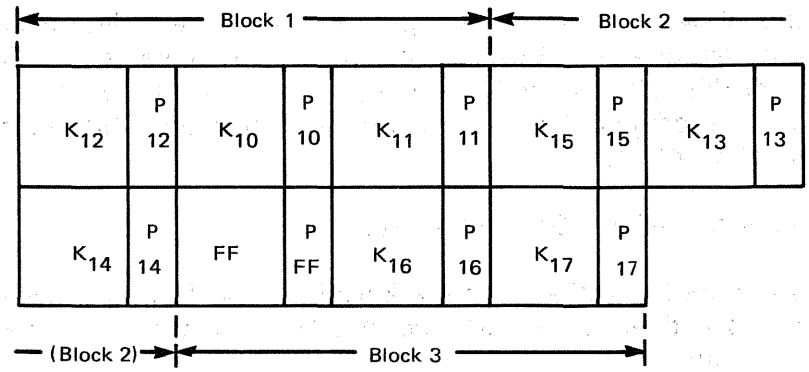
As explained in 11.4.5, when you specify random retrieval operations, or record insertion operations, or both, you may direct ISAM to bring your top index into main storage by proper specification of the INDAREA and INDSIZE keywords.

When your ISAM file is first opened for random retrieval or record insertion, if you have specified that your top index is to be brought into main storage, the OPEN transient computes the number of top index blocks that can be held in a table of the size you have specified with the INDSIZE keyword, reads this number of top index blocks (commencing with the *trailing* block and working toward the *start* of the index), and transfers these blocks to the INDAREA buffer. Although the top index blocks are not reformatted, ISAM eliminates any unused space in each 256-byte index block. The appearance of your top index in main storage is then as depicted in Figure 10—8. Notice that the part of your top index that is in main storage always includes at least the *last* block, which contains the high key (hexadecimal FF). Including this key guarantees that an equal/high comparison will result when the index is searched in main storage.

The search of the index in main storage is initiated by a READ, KEY; ADD; or WRITE, NEWKEY macro issued to the file. First, the search compares the key argument (KEYARG, 11.4.9) to the low key of the top index segment in the INDAREA table. If the argument is equal to or greater than the low key of the INDAREA table, then a comparison is made against the high key of each *block* of the top index in the table until an equal/high comparison results. Then each entry in the block thus located is searched on an equal/high basis, to isolate the index entry that corresponds to the key argument. The search next turns to the block index on the disk (via the intermediate index, if one is present on disk) and from there to direct access of the prime data block sought.



Corresponding INDAREA Index Table in Main Storage,
Assuming INDSIZE Specification Accommodates Three Blocks:



LEGEND:

K Record key

P 3-byte pointer to next index level below

Unused space in top index block on disc

Figure 10—8. Blocks of an ISAM Top Index on Disk and Corresponding INDAREA Table in Main Storage

Referring again to Figure 10—8, consider that the KEYARG field contains the key 13.5. The search follows the following logic:

1. Compare KEYARG (KA) to K_{10}, low key of the INDAREA index table: KA $>$ K_{10}. Therefore continue search in table.

2. Compare KA: K_{12}, high key of block 1: result low.

3. Compare KA: K_{15}, high key of block 2: result high.

4. Therefore search block 2, each entry: result is hit on K_{14}. Search of INDAREA table in main storage is complete; P_{14} points to next lower level of index, on disk.

On the other hand, if the key argument is lower than the low key of the INDAREA table, the search moves the disk, where a hardware search of the top index begins in the usual manner, as described in 10.2. The part of your top index that you have placed in main storage is not searched again on disk.

10.3. ALTERNATE SEQUENTIAL ACCESS METHOD (ASAM)

OS/3 ISAM provides the usual ISAM capabilities of retrieving sequentially or by key; and provides the additional capability of retrieving by address. The coding necessarily includes the simpler coding required for SAM processing and relative record DAM processing. Therefore, it was not difficult to provide handling for unkeyed sequential files wherein the index structure is omitted and keyed functions are avoided. This alternate sequential access method is called ASAM.

Some of the reasons for using ASAM files are:

1. A program is required to retrieve from a keyed file and from a sequential file. If these are specified respectively as ISAM and ASAM files, the same data management coding handles both.

2. Direct addressing into an essentially sequential file is desired, and ASAM handling is found to fulfill the need.

3. It is desired to insert new data between records of a previously formed file, and to retain direct access capabilities. The inserted data may be additional records of the same nature as the original records, or they may be addenda to the record from which they are chained.

The important things to remember about ASAM files are these:

- No index or index space is provided; hence records may not be retrieved by key search.

- Records may be retrieved randomly (as in ISAM) by providing the record address that was returned to you at the time the record was placed in the file. You must make your own arrangements for retention of addresses; alternatively, you may calculate addresses, if your records are of fixed size. In this case, remember that there is one dummy record at file start.

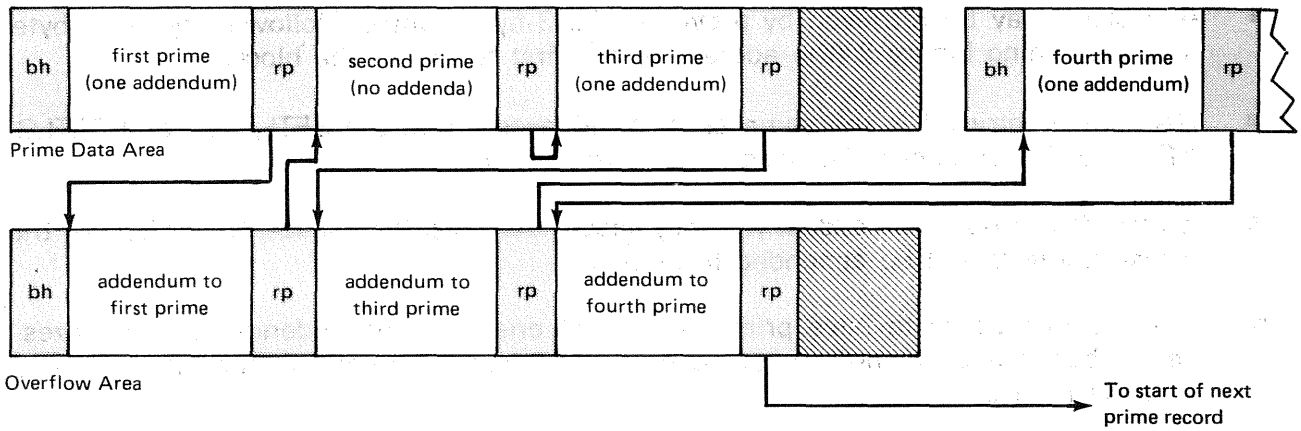
- Any block may be retrieved by providing the 3-byte number, followed by the 2-byte field containing binary 2 (the address of the first record of the block).
- Sequential retrieval is the same as in ISAM except that the SETL, KEY and SETLG, KEY imperatives cannot be used to start the sequence.
- If records are not to be added, you may specify zero overflow space. Nevertheless, the 5-byte pointers will be appended to records.
- It may be desirable to have prime records of one size and addenda of other sizes. Remember that all records must have variable record formats, if there is to be any variance in size.
- In order to add records, you must provide ASAM with the address of the record that is to be followed by the new record.
- ASAM files will usually be defined as unkeyed. If keys *are* specified, ASAM will reject duplicate and out-of-sequence keys during LOAD. When such checking is necessary, you may save some coding by having ASAM do it.
- As in ISAM, you must use the SETL and ESETL imperatives to start and end a sequential progression. While you are in sequential mode you may not perform random functions.

Figure 10—9 shows the logical aspect of an ASAM file in which not more than one addendum record in overflow has been chained from the prime record to which it relates.

Because you may logically insert new records at any point in the ASAM file, it is possible to subordinate or connect several addendum records in overflow to each prima data record. A file you may structure this way may be of more interest or use to you than one based on the one-to-one relationship suggested so far. If you add your new records to overflow and provide ASAM with the address of the same prime data record for each of a group of, say, three records that you want related to it, these are chained in the sequence shown in Figure 10—10. The same chaining can be obtained by successive adds on three separate occasions — a point to remember, therefore, is that when you retrieve these sequentially, the order of retrieval is inverted. The last record of a string added to overflow, chained from a given record, is the first retrieved.

This "last in, first out" (LIFO) retrieval sequence results automatically when you chain a series of overflow records from one prime record and is satisfactory for some applications. However, when you want some other order in sequential retrieval, you may set up the file for this when you provide ASAM, for each new record, with the address of the record (which may be overflow or prime) from which it is to be chained.

Note that there is no pushdown or relocating of records added to the overflow area; the physical location of a new record blocked into overflow is not determined by the position of the prime record from which it is chained nor by the relative location of other records chained from the same prime record.



LEGEND:

bh Block header, a 2-byte field written by data management at the head of each physical data block to indicate the number of bytes of usable data in the block.

rp Record pointer, a 5-byte field inserted by data management following each logical record in the data block. Those inserted following the prime records initially loaded into the ASAM file were originally dummied and contained the hexadecimal pattern F0 AA AA AA AA. This pattern indicated that no related record had been added in overflow, chained from this prime record, and that the next record in logical sequence was therefore the next in physical sequence. When the overflow records were added, data management rewrote the record pointers following those prime records from which an overflow record was chained. The pointer following the first prime record, for example, points to the start of its addendum record in overflow; the pointer after the second prime data record (for which no overflow record was added) still contains the dummy pattern pointing to the next sequential record, in the prime data area.

Logical data records, prime and overflow, provided by the user.

Supplied by OS/3 ASAM.

Unused space in physical data block.

Figure 10—9. Logical Aspect of an ASAM File Containing Not More than One Record Chained in Overflow from Any One Prime Data Record

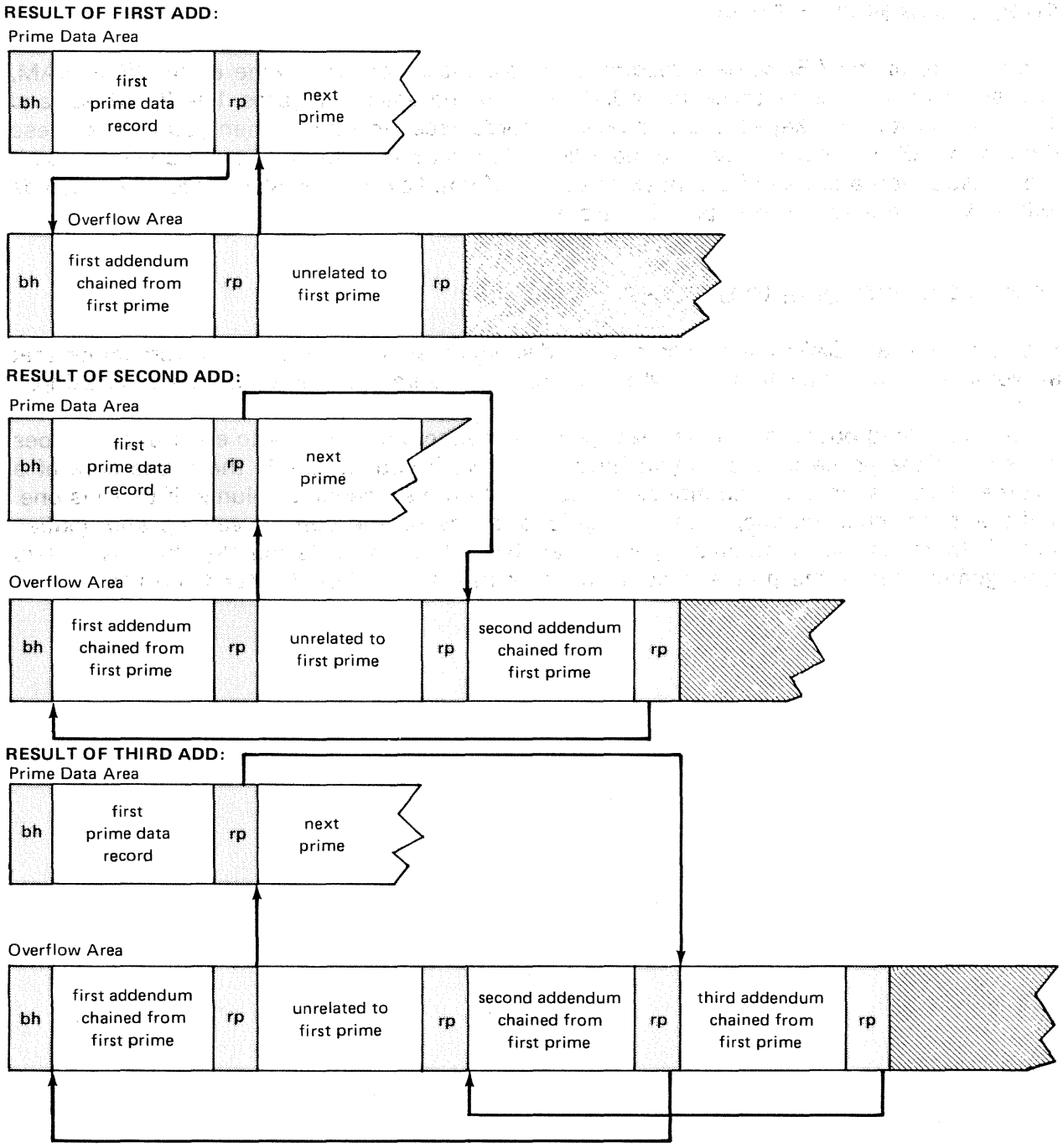


Figure 10—10. Logical Effect of Successively Adding Three Records in Overflow, Chained from Same Prime Data Record of an ASAM File

10.3.1. ASAM Data Formats

The formats of the ASAM data records and data blocks are the same as in OS/3 ISAM, and separate figures of those for ASAM are not necessary. Figures 10—2, 10—3, and 10—4 illustrate both keyed and nonkeyed records. Bear in mind, when you review these figures for ASAM, that you will be most likely to omit record keys when you want to omit the construction and use of an index structure. If you key in your ASAM file, your records will look like the keyed examples illustrated.

10.4. MULTIVOLUME ISAM FILES

You may split an ISAM file across several disk volumes. It is important to remember that all volumes of a multivolume ISAM file must be mounted online for all file processing.

During your load operations, data management monitors continually to ensure that it does not exceed the prime data area you have allocated. If your load fills the prime area of a volume, data management continues the load onto the succeeding volume, if there is one. If there is no other volume, it seeks additional space on the current volume and notifies you if there is no additional space available. When you terminate the load, data management saves the progress point for later use if you should extend the file.

11. Functions and Operation of ISAM

11.1. GENERAL

In the previous section, the discussion of the OS/3 indexed sequential access method (ISAM) aimed primarily at describing file organization, data record and data block formats, and the format of the index blocks in the index structure that is characteristic of the traditional ISAM file. The salient feature of processing an ISAM file that you have constructed with a directory or index structure is the search-by-key function, which is used to retrieve your records by key.

We also pointed out the option for creating a nondirectory ASAM file, in which your records need not be keyed, and for which data management builds no index structure. Accessing records directly without use of an index is accomplished by providing record pointers, rather than keys. Sequential processing of either type of ISAM file is essentially the same. In our discussion of the structure and content of OS/3 ISAM files, we gave you a certain insight into the means OS/3 provides for processing them.

This section builds on that foundation, presenting first an overview of OS/3 ISAM functions. It then presents a declarative macroinstruction, DTFIS, of a specific type, with which you define your file to OS/3 data management and outline to OS/3 ISAM some of your intentions for processing it. The DTFIS macro establishes a file control table that data management uses, during its processing of your file, to keep you and itself informed of the characteristics of the file and the results of processing. (Both the macro and the file table it creates are frequently termed the "DTF," from the initials of the phrase *define the file* we use to distinguish this type of declarative macro from others.)

You describe significant characteristics of your ISAM file, and provide OS/3 data management with certain particulars it needs for processing it, by means of some 27 keyword parameters that you specify as operands of the DTFIS declarative macro. Two of these keywords, which provide the size and symbolic name of the input/output area (or I/O buffer) that every OS/3 data management file needs for its exclusive use, are always required. Most of the other keywords are mandatory under certain conditions or for certain processing functions. Others you will specify entirely at your option. The greater part of the DTFIS macro discussion concerns its keyword parameters.

Following the detailed descriptions of the keywords, this section then presents the set of 12 imperative macroinstructions that make up the OS/3 data management repertoire for processing your ISAM files. Each macro is detailed individually, and the discussion of each one points out any way in which your use of it will change when you use it with the indexed form or the nondirectory form of OS/3 ISAM.

Next, we explain the methods you have of linking your program to the OS/3 ISAM processing modules. An explanation of OS/3 ISAM's system for handling error and exception conditions follows this, and recapitulates certain points made during descriptions of the imperative macros. A number of programming examples conclude the section.

11.2. FUNCTIONAL DESCRIPTION, OS/3 ISAM

Perhaps the best way to obtain a quick overview of the functions for which OS/3 ISAM has been designed is to look at the imperative macros you will issue in your basic assembly language (BAL) program to process your ISAM files. But, before studying each of these macros in detail (they are described fully in 11.5), it may be more useful to consider their functional groupings; these are discussed in the next two paragraphs.

11.2.1. Processing an Indexed ISAM File

As you have noted in Section 10, OS/3 ISAM is provided primarily to enable you to organize and process disk files from which you will have frequent need to retrieve records directly by key, but for which sequential processing is also important to you. ISAM facilitates the search-by-key function by providing a key-based index structure; it implements this function through a repertoire of key-related imperative macros. OS/3 ISAM also provides you with a set of imperatives for sequential processing. Table 11—1 lists the imperatives available for processing an indexed ISAM file. The imperative macro calls are grouped in sets according to functions and are repeated to point out that some are used in more than one setting. Note that those you would expect to call many times in your BAL program have been indented in the list to identify them. (The name of the ISAM file in the illustration is "EMPLMST" - possibly a master file of employees at some installation.)

Table 11—1. Imperative Macro Calls for Processing an OS/3 ISAM File with an Index Structure, Listed by Functions

Functions	Imperative Macro Calls
Initiating and Terminating Processing of the File	OPEN EMPLMST CLOSE EMPLMST
Loading or Extending the File	SETFL EMPLMST WRITE EMPLMST, NEWKEY ENDFL EMPLMST
Inserting New Records	WRITE EMPLMST, NEWKEY WAITF EMPLMST
Random Processing. Retrieval and Updating if WRITE is used	READ EMPLMST, KEY WAITF EMPLMST READ EMPLMST, ID WAITF EMPLMST WRITE EMPLMST, KEY WAITF EMPLMST
Sequential Processing. Retrieval and Updating if PUT is used	SETL EMPLMST, {BOF, KEY, GKEY, ID} GET EMPLMST, (0) PUT EMPLMST ESETL EMPLMST

NOTES:

1. The ADD imperative macro is equivalent to the WRITE,NEWKEY macro for inserting new records.

2. The UPDT imperative macro is equivalent to the WRITE,KEY macro for random updating.

11.2.2. Processing an ISAM File without an Index Structure

When you specify a nondirectory ISAM file, you can no longer locate records by presenting a key to data management. Three of the standard imperative macros are no longer available to you:

READ, KEY
SETL, KEY
SETL, GKEY

Otherwise, the function repertoire is the same. Despite the fact that keys are not used, some macro calls retain the KEY operand.

The WRITE, KEY macro, for example remains available to you for inserting a trailer record in overflow, but you provide this macro with the relative address of the header record from which the trailer depends, not its key.

Sequential processing is the same as for indexed files, except that no index is available for selecting the starting point for a sequential retrieval sequence. Here again, you will provide a relative record address to the SETL macro.

Table 11—2 lists the calls for the imperatives available for processing a nondirectory ISAM file. The calls are grouped in sets according to functions and, as in Table 11—1, are repeated to indicate that some operate in more than one setting. Those calls that you should expect to use many times in your BAL program are indented to identify them. (Again, the 7-character file name of the nondirectory ISAM file being processed in the illustration is "EMPLMST.")

Table 11—2. Imperative Macro Calls for Processing a Nondirectory OS/3 ISAM File without an Index Structure, Listed by Functions

Functions	Imperative Macro Calls
Initiating and Terminating Processing of the File	OPEN EMPLMST CLOSE EMPLMST
Loading or Extending the File	SETFL EMPLMST WRITE EMPLMST, NEWKEY ENDFL EMPLMST
Adding Trailer Records	WRITE EMPLMST,NEWKEY WAITF EMPLMST
Random Processing. Retrieval and Updating if WRITE is used	READ EMPLMST, ID WAITF EMPLMST WRITE EMPLMST, KEY WAITF EMPLMST
Sequential Processing. Retrieval and Updating if Put is used	SETL EMPLMST, {BOF / ID} GET EMPLMST, (0) PUT EMPLMST ESETL EMPLMST

NOTES:

1. The ADD imperative macro is equivalent to the WRITE,NEWKEY macro for inserting new records.

2. The UPDT imperative macro is equivalent to the WRITE,KEY macro for random updating.

11.2.3. Deleting Records from an ISAM File

A function for which OS/3 ISAM does not provide you a specific imperative macroinstruction is record deletion. However, to help you in this aspect of managing a file, OS/3 ISAM establishes a 2-byte field in the DTFIS file table (half-word-aligned) in which your program may keep a count of the number of records you have tagged for deletion. You may address this field, which is available for the life of the file, by concatenating the character "T" to your 7-character file name; this field will be termed *filenameT* in this manual.

You, of course, must establish your own convention for tagging records in your file that are to be deleted; there is no field in the OS/3 ISAM data formats dedicated to this use (nor, if you recall, does OS/3 ISAM provide for user file labels in which you might record deletion statistics instead of using *filenameT).*

Because tagging a record for deletion will most likely be part of a rewrite operation (for which you issue the WRITE, KEY macro), it would be logical also to increment the count in *filenameT* either before or after the rewrite operation. Because your count of records you have tagged for deletion is always available to you in the DTF, you can access it as an aid in deciding when to reorganize your ISAM file.

There is no way to avoid retrieving records you have tagged for deletion; they will always be read by a READ or GET imperative macro. For this reason, to tag records that you have decided should be removed from the file is an important part of your processing them. You should, of course, also provide in your program for checking against the presence of this flag whenever you retrieve a record.

DTFIS

11.3. DEFINING AN OS/3 ISAM FILE (DTFIS)

In order to process a file by OS/3 ISAM, you must define it to data management by issuing the DTFIS declarative macro in your BAL program. (The macro call is derived from the phase *define the file for indexed sequential.*)

The symbolic name of your file **(filename)** may contain no more than seven characters and must begin with an alphabetic character. This file name is also used by the OS/3 ISAM imperative macros to identify the file to be processed. It must be the same as the file name you have included in the LFD statement in the device assignment set of OS/3 job control statements by which you allocate the file. (See the job control user guide, UP-8065 (current version) for the details of OS/3 job control statements.)

The DTFIS declarative macro generates a file table that data management uses to keep itself and you informed of the characteristics of the file and of the results of your processing it with the ISAM imperative macros. The DTF generator assures that the file table is automatically aligned on a full-word boundary. The size of this file table varies according to the processing to be performed (which you may specify via the IOROUT keyword parameter, 11.4.8):

DTF File Table Size, in Bytes	IOROUT Specification
332	LOAD
372	RETRVE
396	ADD
396	ADDRTR

As you execute each imperative macro, data management places an informative reply, indicating normal completion or exceptions (including unrecoverable error conditions) in a special field of the DTF file table. If you have not provided an error exit, you must remember to access this field when control returns inline to you after execution of each imperative.

You address this field (called *filenameC*) by concatenating the character "C" to the name of your file. Some of the imperative macros also pass information to you in other special fields of the DTF file table, which you may address similarly by concatenating a specific character to your file name (details are given for each macro separately in 11.5, and these are recapitulated in 11.7). Because the maximum length for symbolic names throughout OS/3 is eight characters, data management therefore limits you to a maximum of seven for file names.

In addition to the file table just mentioned, the DTFIS declarative macro also generates certain references so that you may link a BAL program with a DTFIS file table you have generated in a separate assembly:

- An ENTRY definition for **filename**. You must specify a corresponding EXTRN definition for **filename** within your program.

- An EXTRN definition for each symbolic name you have supplied with certain of the DTFIS keyword parameters described in what follows. You must specify a corresponding ENTRY definition for each of these symbolic names.

Following is a listing of the required and optional keyword parameters you will specify as operands of the DTFIS declarative macroinstruction. These are listed here in alphabetic order, but you may specify them in any convenient order, just so you separate them with commas. The paragraphs following this format statement discuss the use of each keyword parameter, and a table summarizing the keywords follows the descriptions.

Refer to the preface of this manual for OS/3 data management format statement conventions and to 1.6.3 for rules concerning continuation of statements. A comma is shown preceding each keyword parameter except the first, to remind you that all keywords coded in a string must be separated by commas. However, a comma must neither be coded in column 16 of a continuation line, nor follow the last keyword in the string. Refer to the coding examples which follow.

Format:

LABEL	Δ OPERATION Δ	OPERAND
filename	DTFIS	[ACCESS= {EXC, EXCR, SRD, SRDO}]
		BLKSIZE=n
		[,ERROR=symbol]
		[,INDAREA=symbol]
		[,INDEXED=NO]
		[,INDSIZE=n]
		,IOAREA1=symbol
		[,IOAREA2=symbol]
		[,IOREG=(r)]
		[,IOROUT= {LOAD, ADD, RETRVE, ADDRTR}]
		[,KEYARG=symbol]
		[,KEYLEN=n]
		[,KEYLOC=n]
		[,LOCK=NO]
		[,PCYLOFL=nn]

LABEL	Δ OPERATION Δ	OPERAND
filename (cont)	DTFIS	[,RECFORM={FIXBLK / VARBLK}]
		[,RECSIZE=n]
		[,SAVAREA=symbol]
		[,TYPEFLE={RANDOM / RANSEQ / SEQNTL}]
		[,UPDATE=NO]
		[,VERIFY=YES]
		[,WORK1=symbol]
		[,WORKS=NO]

11.4. DTFIS KEYWORD PARAMETERS

The following is a discussion of the use of each of the keywords that you may specify as operands of the DTFIS declarative macroinstruction; the discussions are arranged alphabetically for ease of reference. Table 11-3, following these descriptions, summarizes all of the keywords.

11.4.1. Specifying File Accessing Options (ACCESS)

In a multitasking environment, the file lock feature can prevent the loss or destruction of data. This feature allows you to control the sharability (specify the read/write requirements) of a file while you are using it. This feature only applies to those files you specified as lockable. The procedure for specifying which files are lockable is described in 16.1.4.

You can specify the sharability of a lockable file by using either the ACCESS or LOCK keyword parameter (11.4.1). It is recommended that you use the ACCESS parameter because it provides greater flexibility (more read/write options available) than the LOCK parameter. If both the ACCESS and LOCK parameter are specified in the same DTF macroinstruction, the ACCESS specification will override the LOCK specification.

Keyword Parameter ACCESS:

ACCESS=EXC
Specifies exclusive read/update/add use of the file. No other jobs can access the file while it is being used.

NOTE:

This specification is equivalent to the default value for the LOCK parameter; that is, LOCK=NO is omitted.

ACCESS=EXCR
Specifies read/update/add use of the file and also allows other jobs to read from the file while it is being used. (Only ACCESS=SRD can be specified for other jobs.)

ACCESS=SRD
Specifies that only the read function is allowed for the file and allows other jobs read/update/add use of the file. (Only ACCESS=EXCR, SRD, or SRDO can be specified for other jobs.)

ACCESS=SRDO
Specifies that only the read function is allowed for the file and also allows other jobs to read from the file. Writing to the file is not allowed from the job associated with this DTF or from other jobs. (Only ACCESS=SRD or SRDO can be specified for other jobs.)

11.4.2. Specifying Size of Data Blocks (BLKSIZE)

To specify the size of the physical data blocks in your ISAM file, you must supply this required keyword parameter in your DTFIS declarative macro. (See 10.2.2 for a discussion of ISAM data block formats; these are shown in Figure 10–4.)

Keyword Parameter BLKSIZE:

BLKSIZE=n
Specifies the size of the blocks in the file, where *n* is the size in bytes. This keyword is always required. Size may not be less than 256 bytes nor exceed the track size for the disk subsystem on which the file resides:

SPERRY UNIVAC Disk Subsystem	Track Size, in Bytes
8411	3625
8414	7294
8415	10,240
8416	10,240
8418	10,240
8424	7294
8425	7294
8430	13,030
8433	13,030

When you specify fixed-length records (by default, or by specifying RECFORM=FIXBLK), your BLKSIZE specification *n* should equal record size + 5 bytes, multiplied by the number of records per data block, adding two bytes for the block header. The block size need not be the same as your I/O buffer allocation, which may be specified with a *define storage* (DS) statement elsewhere in your program, but it must not exceed the length of the buffer.

When 8415 disks are used, the ISAM blocksize is restricted. For each cylinder, ISAM requires at least two blocks of space reserved for overflow. The following algorithm calculates b, the total number of overflow blocks per cylinder:

b = blocks per track x tracks per cylinder

A minimum of one block per track may be specified and two tracks per cylinder (8415 removable) or three tracks per cylinder (8415 fixed). The number of blocks per cylinder is then multiplied by the percentage of overflow (maximum 80%) and rounded up to the next integer value. If less than two, the resulting overflow block count is set to 2. The overflow block count is then subtracted from the total number of blocks per cylinder to calculate d, the data blocks per cylinder, a value which must be nonzero.

d = b — overflow block count

When applied to the 8415 disk (fixed or removable) the algorithm for calculating number of overflow blocks per cylinder (b) can result in a value identical to the total number of blocks per cylinder, thus leaving no space for data. If this condition occurs, the OPEN imperative macro generates the message: DM61 INVALID DTF FIELD: PARAMETER, OR PARAMETER COMBINATION. This restriction condition can occur only with large block sizes (one or two blocks per track) and large overflow percentages (70 to 80%).

11.4.3. Specifying Your Error Exit (ERROR)

When a fatal hardware or detectable logic error occurs on an ISAM file, or an exception is detected to exact performance of the function you have requested, data management returns control to your error-handling routine, if you have coded one and provided its address with the ERROR keyword parameter. If you have no error routine, control returns to you inline, at the instruction in your program next after the imperative macroinstruction that initiated the transfer of control.

It is your responsibility to interrogate the error/status codes and take appropriate action. If you choose to continue processing, it is useful to know that data management provides you an inline return address in register 14; this inline return is to the instruction in your program next following the imperative macro that initiated the transfer of control to your error routine.

When OS/3 data management transfers control, the addressable field *filenameC* in the DTFIS file table contains information on the reasons for the error. (See 11.7 and Appendix B.)

Keyword Parameter ERROR:

ERROR=symbol

Specifies the address of your error-handling routine, to which data management transfers control for all conditions of error or exception to exact performance of the function you have requested. When data management transfers control, *filenameC* contains information on the reasons for the error.

If omitted, control returns to you inline.

11.4.4. Describing an Index Area in Main Storage (INDAREA, INDSIZE)

This pair of keyword parameters is *required* when you are *loading* an indexed file but optional at other times. (It is not used when your file is a nondirectory ASAM file and has no index.) You specify a main storage location into which ISAM may load part of your top index by the INDAREA keyword, and you specify the length of this area, in bytes, with the INDSIZE keyword. Like the I/O area, your index area must always be half-word aligned.

For loading operations, the length of the index area must always be at least 256 bytes because ISAM uses this space to create the 256-byte index blocks as loading proceeds. (See 10.2.3.)

An optional OS/3 ISAM facility, by which ISAM places all or part of the top index in main storage to speed random retrieval or record insertion, may be invoked by specifying the INDAREA and the INDSIZE keywords under the following conditions:

- The KEYARG and KEYLEN keywords are also specified (11.4.9, 11.4.10).
- IOROUT=ADD, IOROUT=ADDRTR, or IOROUT=RETRVE is also specified (11.4.8).
- TYPEFLE=RANDOM or TYPEFLE=RANSEQ is also specified (11.4.15).

The INDSIZE, INDAREA, and KEYLEN parameters define the size and address of the index buffer and the amount of the top index that may be brought into main storage automatically when the file is opened. ISAM determines the size of the top index table entries from the length of keys specified by the KEYLEN keyword and ensures that the length of the INDAREA table (specified by the INDSIZE keyword) will accommodate at least one block of top index entries. If your INDSIZE specification is less than this minimum, your attempt to invoke this facility is negated, and appropriate diagnostic messages are printed in the DTFIS macro expansion in your assembly listing. ISAM automatically rounds your INDSIZE specification *down* to an integer multiple of one block of top index entries.

To ascertain the total number of bytes actually required to hold the entire the entire top index in table form in the INDAREA buffer, you may access *filenameS* in the DTF on completion of a file load sequence; refer to 10.2.4.

For a description of the operation of the ISAM index-in-main-storage facility, refer to 10.2.5.

Keyword Parameter INDAREA:

INDAREA=symbol

Specifies location in main storage in which ISAM builds index blocks during load operations, or where ISAM places top index table for random retrieval or record insertion, where *symbol* (address) is the location. Must be half-word aligned. Required for load operations; optional for others. Length of area specified by INDSIZE keyword.

Keyword Parameter INDSIZE:

INDSIZE=n
Specifies length of index area in main storage, where *n* is the length, in bytes. For load operations, the length must be at least 256 bytes; excess space is unused. For random processing, length greater than 256 bytes is optional; ISAM ensures that INDAREA accommodates at least one block of top index entries (three bytes greater than KEYLEN specification).

11.4.5. Eliminating the Index Structure (INDEXED)

This keyword is used when you want to eliminate construction of the ISAM index structure.

Keyword Parameter INDEXED:

INDEXED=NO
Specifies that you have elected the option to create a nondirectory ISAM file and to reference its records by relative addresses, rather than by keys. Data management does not construct the index; key-based processing functions are inoperative.

If omitted, the index structure is provided.

11.4.6. Specifying I/O Buffers (IOAREA1, IOAREA2)

All ISAM operations require at least one I/O area, half-word-aligned and at least 256 bytes in length. You specify its address with the IOAREA1 keyword. You may specify a second area, of equal size, which must also be aligned on a half-word boundary, with the optional IOAREA2 keyword. To do so will increase the speed of your processing; you will notice that the benefits are more pronounced for load and sequential retrieval than for random operations.

Keyword Parameter IOAREA1:

IOAREA1=symbol
Required to specify location of input/output area, where *symbol* (address) is the location. Must be half-word aligned. Length must equal or exceed the number of bytes specified with BLKSIZE keyword.

Keyword Parameter IOAREA2:

IOAREA2=symbol
Specifies location of optional additional input/output area, where *symbol* (address) is the location. Must also be half-word aligned and will be same size as the required area specified by the IOAREA1 keyword. You may improve the speed of sequential I/O operations if you specify this keyword.

11.4.7. Specifying Current Record Pointer (IOREG)

When you are referencing your records in the I/O areas rather than in the work areas, you need to specify a general register to be used to point to the current I/O area. Registers 2 through 12 are always available; if you have specified the SAVAREA keyword parameter, general register 13 is also available. (See 11.4.18 for details on the use of record work areas.)

Keyword Parameter IOREG:

IOREG=(r)
Required to specify the general register to be used to point to the current I/O area when you are not referencing records in the work areas, where r is the number of the general register. Registers 2 through 12 are available, and register 13 is also available if you have specified the SAVAREA keyword.

11.4.8. Specifying the Type of File Processing (IOROUT)

You may specify the type of processing to be performed on your file with the optional IOROUT keyword; this parameter has four forms. Note that you may also specify the type of *retrieval* with the TYPEFLE keyword (11.4.15).

A file-loading operation (IOROUT=LOAD) may end with the final data block only partly full. When this occurs, data management stores the exact location of unused file space in the disk volume table of contents (VTOC) for later recovery and use as needed.

You may add records (IOROUT=ADD) with keys higher than the final prime data record, which is contained in the data block marked as the logical end of file. As data management places these in overflow, they do not affect the location of unused prime space.

If you should introduce new records by resuming load operations (IOROUT=LOAD), these must be submitted in ascending order of keys; all must have higher keys than the current highest key (whether in a prime record or an overflow record already on disk).

If you add enough records to an ISAM file, some cylinder oveflow space will become filled, and data management will be unable to place added records on the cylinder where they belong. When this occurs, data management resorts to using space on the earliest cylinder having available overflow and adds records at the earliest possible point of the following sequence:

1. On the cylinder reached by prime search
2. On the cylinder having the current COCRS*
3. On a cylinder newly set as having the COCRS (discovered by progressing through COCR blocks to one showing space available)

**The COCRS is a special cylinder overflow control record that data management writes and maintains in the DTF and VTOC to control remaining overflow space on the cylinder that is currently used to accept overflow records from other cylinders.*

Keyword Parameter IOROUT:

IOROUT=ADD
Specifies that new records are to be inserted into a file. You may not specify the ADD form of this keyword unless you have allocated an overflow area with the PCYLOFL keyword parameter (11.4.12).

IOROUT=ADDRTR
Specifies that new records are to be inserted in the file and that records will also be retrieved and updated randomly or sequentially. You may not specify the ADDRTR form of this keyword unless you have allocated an overflow area with the PCYLOFL keyword (11.4.12).

IOROUT=LOAD
Specifies that either a new file is to be created or an existing file is to be extended. ISAM assumes that the LOAD form has been specified if you omit the IOROUT keyword parameter.

IOROUT=RETRVE
Specifies that your records are to be retrieved or updated randomly or sequentially.

If omitted, IOROUT=LOAD is assumed. Type of retrieval may be specified with the TYPEFLE keyword (11.4.15).

11.4.9. Specifying Location of Retrieval Search Argument (KEYARG)

Whether you are referencing records by key (as when you are using ISAM with the index structure) or by relative address (as with the nondirectory form of ISAM), you need a field in which to present data management either with the key it is to match by the *search-on-key* function or with the relative disk address at which it is to access your record directly. You specify this field with the KEYARG keyword parameter.

You should avoid presenting ISAM with either the address or the all-zero key of the dummy record at file start as a search argument. The dummy record is not available for you to use, and efforts to retrieve or overwrite it result in the error processing described under the various imperative macros involved. You may, on the other hand, safely specify an all-zero key in the KEYARG field when you issue the SETL, GKEY imperative: data management prepares to give you the first record after the dummy, and no error processing results.

The length of the KEYARG area will be five bytes when you use it only for relative addressing, but it must equal the actual length of the keys in your records (specified by the KEYLEN keyword) when you are using the indexed form of ISAM. You may omit the KEYARG keyword parameter when you are not retrieving records.

Keyword Parameter KEYARG:

KEYARG=symbol
Specifies the field in your program where you will place addresses or keys to effect retrieval of your records, where *symbol* (address) is the location in this field. The length of the KEYARG area is five bytes when you use it for relative addressing and equal to key length (KEYLEN keyword parameter) otherwise. May be omitted when you will not retrieve records.

11.4.10. Specifying Length and Location of Records Keys (KEYLEN, KEYLOC)

When your records have keys (as they must when you use the indexed form of ISAM, and may when you are using the nondirectory form), all keys in the file must have the same length, and every record must have a key. The key need not begin at the head of the record; it may be internal to the record, in fact, but the starting place for the key must be the same in every record. The minimum length for a key is 3 bytes; the maximum is 253. Each key must be unique. No byte of any record's key may contain the hexadecimal value FF, nor may any of your keys duplicate the all-zero key or the dummy record that ISAM creates and inserts at the start of the file.

You specify the length of the key in bytes with the KEYLEN keyword parameter and the number of bytes of data that precede the key with the KEYLOC keyword. You *must* specify the KEYLEN parameter for an indexed ISAM file; you need *not* specify it for a nondirectorty file unless you want ISAM to check the sequence of your keys during the load operation (remember that sequence checking is done automatically only when you are loading your records for an indexed ISAM file, to which you present them in ascending order of keys).

You need not specify the KEYLOC keyword if the key is at the head of the record. When you omit this keyword, ISAM assumes that you have specified KEYLOC=0 for fixed records or KEYLOC=2 for variable records (each of which, as you recall, contains its record length in the leading two bytes).

Keyword Parameter KEYLEN:

KEYLEN=n
Specifies the length of keys in an ISAM file, where *n* is the length in bytes. All keys in an ISAM file must have the same length; the minimum length is 3 bytes, and the maximum is 253. Required for indexed ISAM files; optional otherwise. If specified for a nondirectory ISAM file, causes data management to check sequence of keys during a record load.

Keyword Parameter KEYLOC:

KEYLOC=n
Specifies the number of bytes that precede the key of an ISAM record, where *n* is this number. The location of the keyfield must be the same within all records of the file.

If omitted, ISAM assumes that you have specified a value of zero for fixed records or two for variable records.

11.4.11. Suppressing a File Lock (LOCK)

As we mentioned earlier, there are two ways to specify the sharability of your disk files. We have already discussed the ACCESS keyword parameter and now we will discuss the other way: that is, the LOCK keyword parameter.

Two options are available with the LOCK parameter:

1. The file is exclusively locked when it is opened during the execution of your program. You have exclusive use of the file. You can read, update, and add to the file. No other user can open the file until you close it.

2. A read-only lock is applied to the file when it is opened. You can only read from the file and all other uses can only read from the file.

Keyword Parameter LOCK:

LOCK=NO
This is equivalent to specifying ACCESS=SRDO. It should not be used in the same DTF as the ACCESS keyword parameter. If it is, the ACCESS keyword parameter will override it.

If you omit both LOCK=NO and the ACCESS keyword parameter, this is equivalent to specifying ACCESS=EXC.

11.4.12. Providing Cylinder Overflow Area (PCYLOFL)

For both the indexed and the nondirectory ISAM file, you specify the *percentage* of the space of each cylinder that data management is to reserve for overflow with the PCYLOFL keyword parameter. You will recall that it is to the overflow area that data management writes records added after your initial load. The maximum percentage is 80.

If you specify a zero value or if you omit the PCYLOFL keyword altogether, you may not add records to the file; any you attempt to insert will be rejected.

Keyword Parameter PCYLOFL:

PCYLOFL=nn
Specifies the percentage of each cylinder that data management is to reserve for overflow, where *nn* is the percent. The value of *nn* may range from 00 through 80. If you specify PCYLOFL=00, records presented later for insertion will be rejected.

If omitted, data management assumes that you have specified PCYLOFL=00.

11.4.13. Specifying Record Size and Format (RECFORM, RECSIZE)

Records in an ISAM file are fixed or variable in length and are blocked by data management. You may specify the format with the RECFORM keyword parameter; if your records are fixed length, you must specify this length with the RECSIZE keyword. All fixed records must have the same length in an ISAM file.

Keyword Parameter RECFORM:

RECFORM=FIXBLK
Specifies that your records are fixed-length, blocked. You must also specify the RECSIZE keyword.

RECFORM=VARBLK
Specifies that your records are variable-length, blocked. You do not specify the RECSIZE keyword.

If omitted, data management assumes that you have specified RECFORM=FIXBLK, and you must specify the RECSIZE keyword parameter.

Keyword Parameter RECSIZE:

RECSIZE=n
Specifies the length of fixed records, where *n* is this length, measured in bytes. Required for fixed-length records only; must be specified if you omit the RECFORM keyword. Not used for variable records.

11.4.14. Specifying a Save Area for Contents of General Registers (SAVAREA)

Before you issue an imperative macro for processing any OS/3 data management file, you may first load general register 13 with the address of a 72-byte labelled save area — always aligned on a full-word boundary — in which data management will expect to save the contents of your registers. If you do not want to provide this information with every call, you may place the location of the save area in the DTF file table by specifying the SAVAREA keyword. Refer to 1.4 for the content of this area.

Keyword Parameter SAVAREA:

SAVAREA=symbol
Specifies the address of a 72-byte labeled save area for the contents of general registers, full-word-aligned, where *symbol* (label) is the address. Used only when register 13 is not loaded with save area address before each issue of imperative macros to the file.

If omitted, data management assumes that you have preloaded register 13 with address of a save area before issuing each imperative.

11.4.15. Specifying the Type of Retrieval (TYPEFLE)

When you are performing retrieval operations on your ISAM file, indexed or nondirectory (and therefore have specified IOROUT=RETRVE or IOROUT=ADDRTR (11.4.8.)), you may specify the type of processing (random or sequential) with the TYPEFLE keyword parameter. You do not use the TYPEFLE keyword unless you are retrieving records.

Keyword Parameter TYPEFLE:

TYPEFLE=RANDOM
Specifies that random (direct) retrieval operations are to be performed.

TYPEFLE=SEQNTL
Specifies that sequential retrieval operations will be performed.

TYPEFLE=RANSEQ
Specifies that both random and sequential retrieval will be peformed.

If omitted, data management assumes that you have specified TYPEFLE=SEQNTL. Not used unless records are to be retrieved. IOROUT=RETRVE or IOROUT=ADDRTR must also be specified (11.4.8).

11.4.16. Forestalling Use of Update Functions (UPDATE)

When you want to avoid the possibility of inadvertently writing to an ISAM file that you intend to be a read-only file, you may specify the UPDATE keyword parameter in the DTFIS macro. This keyword sets a bit in the DTF file table that causes data management to set the *invalid macro* error flag (byte 0, bit 6) in *filenameC* if you should issue a PUT or WRITE imperative macro to this file, and you then may take no action on the file other than to close it. (See Appendix B.)

Keyword Parameter UPDATE:

UPDATE=NO
Specifies that data management is to flag a subsequent issue of the PUT or WRITE macro as an *invalid macro* (byte 0, bit 6 of *filenameC*); no further reference to the file, other than to close it, is possible.

If omitted, the possibility of inadvertent updating of the file is not forestalled.

11.4.17. Specifying Parity Check of Output Records (VERIFY)

When you need data management to make a parity check of your records after it has written each one to disk you must specify the VERIFY keyword parameter; otherwise, no check reading will be done. You should remember that specifying a parity check will necessarily increase the execution time required for the PUT and WRITE macros by about one rotation period per block. You must weigh this overhead against the advantage of immediate notification of problems within your file.

Keyword Parameter VERIFY:

VERIFY=YES
Specifies that data management is to check parity of output records after they have been written to disk. Necessarily increases execution time for PUT and WRITE macros. If bad parity is detected, data management sets *output parity check* flag (byte 2, bit 2) in *filenameC* and transfers control to your error routine or to you inline.

If omitted, no output parity verification will be done.

11.4.18. Specifying Location of Record Work Areas (WORK1, WORKS)

For loading and adding functions (that is, when you have specified IOROUT=LOAD, IOROUT=ADD, or IOROUT=ADDRTR, 11.4.8), you must provide ISAM with the location of a record work area by specifying the WORK1 keyword parameter. Furthermore, unless you have selected a general register to be the current record pointer (IOREG keyword 11.4.7), you must also specify a record work area for random retrieval.

For *sequential* retrieval, you do *not* specify the location of a record work area in the DTF, and data management ignores the WORK1 keyword if it is present. You have two other options for sequential retrieval:

1. Working in the input buffer. To do this, you specify the IOAREA1 and IOAREA2 keywords and both the IOREG and the WORKS keyword parameters.

2. Transferring records to the work area. To do this, you do not specify either the IOREG or the WORKS keyword parameter; you specify the location of the work area in the second positional parameter of each GET imperative macro you issue (11.5.5.2).

An important point to remember is that the record work area, however you specify it, holds one record at a time. An obvious point is that its size is governed by your record length; if your records are variable, the size must accommodate the largest of these.

Keyword Parameter WORK1:

WORK1=symbol
Provides the location of a record work area, where symbol (address) is the location. Required for load and add functions (when IOROUT=LOAD, IOROUT=ADD, or IOROUT=ADDRTR has been specified). Also required for random retrieval unless you have specified the IOREG keyword parameter. Is ignored for sequential retrieval.

Keyword Parameter WORKS:

WORKS=NO
When IOREG keyword is also specified for sequential retrieval, indicates that you will process records in the current input buffer.

If omitted, and IOREG keyword is not specified, data management expects to transfer sequentially retrieved records (one at a time) to the work area you specify as an operand of each GET macro you issue (11.5.5.2).

11.4.19. Nonstandard Forms of the Keyword Parameters

When the assembler is preparing your DTF, it uses a specific list of keywords. Discovery of a keyword that is not on the list results in an assembly error. In order that existing programs may require minimal changes for assembly in OS/3, the list of acceptable keywords has been expanded beyond those listed as standard. The acceptable variations are:

OS/3 Standard	Acceptable
BLKSIZE	BKSZ
ERROR	ERRO
INDAREA	INDA
INDSIZE	INDS
IOAREA1	IOA1, IOAREAL, IOAREAR, IOAREAS
IOAREA2	IOA2
IOREG	IORG

OS/3 Standard	Acceptable
IOROUT	IORT
KEYARG	KARG
KEYLEN	KLEN
KEYLOC	KLOC
PCYLOFL	PCYL, CYLOFL, CYL
RECFORM	RCFM
RECSIZE	RCSZ
SAVAREA	SAVE
TYPEFLE	TYPF
UPDATE	UPDT
VERIFY	VRFY
WORK1	WRK1, WORKL, WORKR

Note that there are no acceptable variations for the INDEXED or the WORKS keywords and that none of the completion values (the values to which you equate these keywords) may vary from the OS/3 standards given in the preceding paragraphs.

11.4.20. Recapitulation of DTFIS Keyword Parameters

Table 11—3 lists all of the keyword parameters that may be used as operands of the DTFIS declarative macroinstruction. An example of coding the DTFIS declarative macro follows the table.

Table 11—3. Keyword Parameters of the DTFIS Declarative Macro Instruction (Part 1 of 2)

Keyword	Specification	File Function				Remarks
		L	A	S	T	
ACCESS*	EXC	S	S	S	S	This DTF: read/update/add use Other jobs: no access
	EXCR			S	S	This DTF: read/update/add use Other jobs: read use
	SRD			S	S	This DTF: read use Other jobs: read/update/add use
	SRDO			S	S	This DTF: read use Other jobs: read use
BLKSIZE *	n	R	R	R	R	
ERROR	symbolic label	O	O	O	O	Address of subroutine to handle errors and exceptions
INDAREA	symbolic label	R	O	O	O	Address of main storage area to contain index
INDEXED	NO	O	O	O	O	Specifies that file is not to be indexed
INDSIZE **	n (in bytes)	R	O	O	O	Size of index area in main storage; minimum size is 256 bytes.
IOAREA1	symbolic label	R	R	R	R	Address of I/O area in main storage

Table 11—3. Keyword Parameters of the DTFIS Declarative Macro Instruction (Part 2 of 2)

Keyword	Specification	File Function				Remarks
		L	A	S	T	
IOAREA2	symbolic label	O	O	O	O	Address of a second I/O area in main storage to speed processing
IOREG	(r)=general register			O	O	Contains address of the I/O area in main storage.
IOROUT	ADD		R			Insert new records to file.
	ADDRTR		R	S	S	Insert new records and retrieve records randomly or sequentially.
	LOAD	R				Create new file or extend existing file.
	RETRVE			S	S	Retrieve and/or update randomly or sequentially.
KEYARG	symbolic label			O	R	Address of field containing key of desired record.
KEYLEN **	n	O	O	O	O	Key length in bytes for ISAM file. When specified for nonindexed files, a sequence check of keys is made.
KEYLOC **	n	O	O	O	O	Location, in bytes, of the key within a record. If omitted, KEYLOC=0 for fixed-length records; KEYLOC=2 for variable-length records.
LOCK	NO	O	O	O	O	Requests file lock not be set on a lockable file at OPEN
PCYLOFL	nn (00 to 80)	O				Percentage of cylinder (blocks) available for overflow
RECFORM *	FIXBLK	S	S	S	S	Specifies fixed-blocked records
	VARBLK	S	S	S	S	Specifies variable-blocked records
RECSIZE *	n	O	O	O	O	Size, in bytes, of fixed records to be processed
SAVAREA	symbolic label	O	O	O	O	Address of general register save area
TYPEFLE	RANDOM				O	Specifies random file processing function
	RANSEQ			O	O	Specifies random/sequential file processing function
	SEQNTL			O		Specifies sequential file processing function
UPDATE	NO			O	O	Eliminates update capability
VERIFY	YES	O	O	O	O	Specifies a parity check of data records is to be made after being written to output disc
WORK1	symbolic label	R	R		O	Address of work area for records being loaded, reloaded, extended, or inserted
WORKS	NO			O		No record work area available for sequential retrieval

LEGEND:

L File creation or extension
A Record insertion
S Sequential retrieval
T Random retrieval
O Optional
R Required
S Select one
Value assumed if keyword is not specified.

*Parameter may be changed by DD job control statement.
**Parameter may be changed by DD job control statement, indexed mode only.

Example:

```
LABEL     OPERATION     OPERAND                    COMMENTS
1         10            16                         72
*   DEFINE AN INDEXED SEQUENTIAL FILE NAMED GRAND
*
GRAND     DTFIS IOROUT=ADDRTR,        INSERTION AND RETRIEVAL
              TYPEFILE=RANSEQ,        RETRIEVAL IS RANDOM AND SEQUENTIAL  X
              IOAREA1=NEWRECD,        I/O AREA IN MAIN STORAGE            X
              WORK1=LABOR,            WORK AREA FOR INSERTION             X
              IOAREA2=OLDREC,         EXTRA I/O AREA TO SPEED PROCESSING  X
              RECFORM=VARBLK,         RECORDS ARE VARIABLE AND BLOCKED    X
              KEYLEN=25,              KEY IS 25 BYTES LONG                X
              KEYLOC=15,              15 BYTES OF RECORD PRECEDE KEY      X
              KEYARG=SEARCKEY,        KEY IS SEARCKEY FOR RANDOM RETRIEVAL X
              ERROR=ERREX,            ON ERROR GO TO ERREX ROUTINE        X
              VERIFY=YES              PARITY CHECK OF WRITE ORDERS IS TO
*                                     BE PERFORMED
```

11.5. IMPERATIVE MACROS FOR ISAM FILES

To process your ISAM files, you will issue imperative macros in your BAL program to communicate with OS/3 data management. These imperative macros expand as inline executable code to set up linkages, pass required parameters, and initiate transfer of control to the various ISAM transients and logic modules.

As explained in further detail in 11.7 and Appendix B, data management sets a flag in a 4-byte addressable field in your DTF file table (labelled *filenameC*), after the execution of each imperative macro you issue, to inform you of the normal completion of the processing you have specified or of error or exceptional conditions. If you do not provide an error/exception handler routine, it is your responsibility to interrogate *filenameC* after each inline return and to take appropriate action in your program.

Certain of these imperative macros also provide other useful information to you, in different fields of the DTF file table *(filenameH, filenameP,* and so on). These actions are pointed out in the individual macro descriptions and also recapitulated in 11.7.

The imperative macro descriptions are grouped according to the file processing functions involved:

- Basic macroinstructions: OPEN and CLOSE
- File loading and extending macros: SETFL; WRITE, NEWKEY; and ENDFL
- Random processing macros: READ, KEY; READ, ID; WRITE, KEY; UPDT; and WAITF
- Record insertion macros: WRITE, NEWKEY; ADD; and WAITF
- Sequential processing macros: SETL, GET, PUT, and ESETL

11.5.1. Basic Macroinstructions

You must use the OPEN and CLOSE macroinstructions to initiate and terminate action on an ISAM file. They call transient routines into main storage to perform the necessary preparation and close-out. The OPEN macroinstruction must be issued before any other macro function can be performed.

OPEN

11.5.1.1. Initializing an ISAM File (OPEN)

Format:

LABEL	ΔOPERATIONΔ	OPERAND
[name]	OPEN	{filename-1[,...,filename-n] / (1) / 1}

Positional Parameter 1:

filename
Is the label of the corresponding DTFIS declarative macroinstruction in the program. The maximum number of file names is 16.

(1) or **1**
Indicates that you have preloaded register 1 with the address of the DTFIS file table.

Examples:

LABEL (1)	ΔOPERATIONΔ (10)	OPERAND (16) Δ
	OPEN	EMPLMST
	OPEN	(1)

CLOSE

11.5.1.2. Terminating an ISAM File (CLOSE)

Function:

You must use the CLOSE macroinstruction to terminate processing of your file. The CLOSE macroinstruction calls on a transient routine that performs required termination operations, such as updating the format 2 label. Once your file is closed, no other macroinstructions can be executed for the file until it is reopened by the OPEN macroinstruction.

Format:

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

Positional Parameter 1:

filename
Is the label of the corresponding DTFIS declarative macroinstruction in your program. The maximum number of file names is 16.

(1) or **1**
Indicates that you have preloaded register 1 with the address of the DTFIS file table.

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

Example:

LABEL	ΔOPERATIONΔ	OPERAND Δ
	.	
	.	
	CLOSE	EMPLMST

1 10 16

Programming Considerations:

When you have issued a CLOSE macro, you may access *filenameS*, if desired, to ascertain the number of bytes required to hold the top index in main storage (11.6.2).

11.5.2. Loading and Extending an ISAM File

Whether a new ISAM file is to be loaded (created) or an existing one is to be extended by adding records at the end, the file processing functions are the same. Both functions are indicated in the DTFIS declarative macroinstruction when you specify IOROUT=LOAD. Records for the file are supplied in a work area to be blocked by data management.

The imperative macroinstructions you require are the same in either case. The two processing functions are differentiated by the disk format 2 label. Once a file has been loaded successfully and a CLOSE macroinstruction has been executed for it, subsequent processing of a file for which you have specified IOROUT=LOAD extends, rather than creates, the file.

The three imperative macroinstructions are:

- SETFL, which initiates the processing sequence:
- WRITE, NEWKEY, which loads a record to the file; and
- ENDFL, which terminates the processing sequence.

You do not follow the WRITE, NEWKEY macro with the WAITF macro for a load operation, although the WAITF macro is required after all *other* uses of the WRITE macro and all uses of the READ macro in OS/3 ISAM.

SETFL

11.5.2.1. Initiating the Load Sequence (SETFL)

Function:

The SETFL *(set file load)* macroinstruction calls on a transient routine which sets up controls in the DTFIS file table and in the indexes on the disk to prepare file for loading (or extending).

Format:

LABEL	ΔOPERATIONΔ	OPERAND
[name]	SETFL	{ filename (1) 1 }

Positional Parameter 1:

filename
Is the label of the corresponding DTFIS file table in your program.

(1) or **1**
Indicates that you have preloaded register 1 with the address of the DTFIS file table.

Example:

LABEL (1)	ΔOPERATIONΔ (10)	OPERAND (16) Δ
	.	
	.	
	SETFL	EMPLMST
	.	
	.	

WRITE, NEWKEY

11.5.2.2. Writing Initial Records to the File (WRITE, NEWKEY)

Function:

The WRITE, NEWKEY macroinstruction (that is, WRITE is the operation code, and NEWKEY is positional parameter 2) writes a logical record to a file being loaded or extended. Specifically, it transfers a record from the working storage area to the I/O area. Before issuing the WRITE, NEWKEY macroinstruction, you must have stored the logical record in the work area.

Format:

LABEL	Δ OPERATION Δ	OPERAND
[name]	WRITE	{ filename | (1) | 1 } ,NEWKEY

Positional Parameter 1:

filename
Is the label of the corresponding DTFIS declarative macroinstruction in the program.

(1) or **1**
Indicates that register 1 has been preloaded with the address of the DTFIS file table.

Positional Parameter 2:

NEWKEY
Indicates that a new record is to be loaded into an ISAM file.

Examples:

```
     LABEL      ΔOPERATIONΔ        OPERAND                    Δ
     1          10         16
                |          |
1.               WRITE      EMPLMST,NEWKEY
2.               WRITE      (1),NEWKEY
```

Programming Considerations:

The WRITE, NEWKEY macroinstruction causes the following actions:

- The key in the work area is checked against the key of the last record transferred into the I/O area, and if it is not greater, either a duplicate key or a sequence check error has occurred. Data management sets the appropriate flag in the *filenameC* field of the DTFIS file table and returns control to your error-handling routine, if you have one, or to you inline. The record in error is not transferred to the I/O area, and you may resume normal processing with another valid logical record.

- The record and its key are transferred from the work area to the I/O area.

- When the I/O area cannot accommodate the entire record, a block is written into the prime data area on the disk, and the given record is used to start the next block.

- When a block is written, a key-pointer entry is formed for the block index, provided that you have specified an indexed file. For an ASAM file (INDEXED=NO), this action is omitted.

- Following execution of the WRITE, NEWKEY macroinstruction, the disk address of the logical record is available to you in a DTFIS addressable field labeled *filenameH*. You may save these addresses during load and present them later for direct accessing. A 3-byte count of the total number of logical records in the prime data area (contained in *filenameP* of the DTFIS file table) is also available to you.

  You may not overwrite the dummy record at file start, which is not available to you for storing data. If you issue the WRITE, NEWKEY macro with a search key of all binary 0's in your KEYARG field (this being the key reserved for the dummy), error processing results. Data management sets the *invalid ID* error flag in *filenameC*, issues error message DM24 (INVALID REQUEST (ID) — OUTSIDE FILE LIMITS), and branches to your ERROR routine. Refer to Appendix B.

Examples:

	LABEL (1)	OPERATION (10)	OPERAND (16)	Δ
1.		WRITE	EMPLMST,NEWKEY	
2.		WRITE	(1),NEWKEY	

1. Assuming register 1 has been preloaded with the EMPLMST address, and the WRITE, NEWKEY macroinstruction has been executed successfully, the field in the DTDIS file table labeled EMPLMSTH holds the logical record address.

2. If an error or warning condition has occurred, the field in the DTFIS file table labeled EMPLMSTC contains an indication of this condition.

ENDFL

11.5.2.3. Terminating the Load Sequence (ENDFL)

Function:

The ENDFL (*end file load*) macroinstruction calls on a transient routine that terminates your file loading or extending functions for the file and writes the final block on the disk. Also, file parameters are tested and any required index processing is performed.

Format:

LABEL	△OPERATION△	OPERAND
[name]	ENDFL	{ filename (1) 1 }

Positional Parameter 1:

filename
Is the label of the corresponding DTFIS file table in your program.

(1) or **1**
Indicates that you have preloaded register 1 with the address of the DTFIS file table.

Example:

LABEL 1	△OPERATION△ 10	OPERAND 16 △
	ENDFL	EMPLMST

11.5.3. Inserting New Records in an ISAM File

Once an ISAM file has been created, you can add new records to the file, providing that you allocated overflow space on disk during load. Each new record is placed in the overflow area and chained into logical sequence. You specify this file processing function by specifying IOROUT=ADD (or IOROUT=ADDRTR, if retrieval is also desired) in the DTFIS declarative macroinstruction and use either the ADD or the WRITE,NEWKEY imperative macro to add each new record.

You supply new records in a work area, as you did during file creation. However, keys of added records need not be in ascending sequence. For ASAM files, the area specified by the KEYARG keyword parameter must contain the address of the record from which the current item is to be chained. The DTFIS keyword parameters must be equated to the same specifications as when the original file was created.

You issue the imperative macroinstructions WRITE, NEWKEY (or ADD) and WAITF to add records to an ISAM file. The form of the WRITE, NEWKEY macroinstruction for adding to a file is the same as you use for loading or extending a file, although the functions performed are different.

In ISAM, there is no restriction preventing the adding of records with keys lower than the key of the first record loaded, *except* the key of all binary zeros. ISAM has begun the file with a dummy record having this key; so error processing will result if you attempt to add a record whose key is binary 0. This is described under the WRITE, NEWKEY macro description in 11.5.2.2.

In the course of adding records to a file, overflow areas may become filled. After each WAITF macroinstruction, the conditions are reflected in the program-addressable fields in the DTFIS file table. These fields are as follows:

- *FilenameA*

  A 2-byte field indicating the number of prime data cylinders having full cylinder overflow areas. This field is set to zero if you have not specified cylinder overflow with the PCYLOFL keyword.

- *FilenameO*

  A 2-byte field indicating the total number of overflow records.

- *FilenameP*

  A 3-byte field indicates the total number of prime data records in the file.

WRITE, NEWKEY

11.5.3.1. Adding a New Record to Overflow in an Existing File (WRITE, NEWKEY)

Function:

When you specify IOROUT=ADD or IOROUT=ADDRTR, the WRITE, NEWKEY macroinstruction logically inserts a new record in an existing file. You cannot use this macroinstruction unless you have allocated cylinder overflow area during load with the PCYLOFL keyword (11.4.12). Before issuing the WRITE, NEWKEY macroinstruction, you must have stored the logical record in the working storage area in the logical record format.

In response to this macroinstruction, data management does the following:

1. Searches through the index to locate the record's prime data block. This block or the overflow chain is then searched to determine the position into which the new record should be inserted.

2. Places the new record in overflow.

3. Installs chaining in the blocks as necessary to maintain logical sequence.

For ASAM files, the index search is replaced by a direct access to the proper block; you provide data management with the 5-byte file-relative address of the record from which the new record is to be chained by loading it into the KEYARG field before issuing the WRITE, NEWKEY macro (11.4.9).

To ensure that all the actions initiated by the WRITE, NEWKEY macroinstruction have been completed, you must execute a WAITF macroinstruction. When control is returned to you from the latter, the work area is available for further insert records. The record just inserted is no longer in WORK1.

Format:

LABEL	Δ OPERATION Δ	OPERAND
	WRITE	{ filename | (1) | 1 } ,NEWKEY

Positional Parameter 1:

filename
Is the label of the corresponding DTFIS declarative macro in your program.

(1) or **1**
Indicates that you have preloaded register 1 with the address of the DTFIS file table.

Positional Parameter 2:

NEWKEY
Indicates that a new record is to be written into an ISAM file.

Examples:

LABEL (1)	OPERATION (10)	OPERAND (16)
	WRITE	EMPLMST,NEWKEY
	WRITE	(1),NEWKEY

Programming Considerations:

If an error or warning condition has occurred, the field in the DTFIS file table labeled EMPLMSTC will contain an indication of the condition.

ADD

11.5.3.2. Adding a New Record to Overflow in an Existing File (ADD)

Function:

The ADD imperative macro, exactly equivalent to the WRITE, NEWKEY macro used when IOROUT=ADD or IOROUT=ADDRTR is specified, adds a new record to the overflow area of an existing ISAM or ASAM file and chains it into the appropriate logical sequence.

As with the WRITE, NEWKEY macro, you must have previously provided an overflow area with the PCYLOFL keyword (11.4.12) when you originally created the file, and you must specify the IOROUT keyword as just stated. You store the logical record in the work area before you issue the ADD macro, and you issue a WAITF macro after it, before issuing another function to the file.

If your file is an ASAM file, the ADD macro does not conduct a search on key but directly accesses the data block (prime or overflow) containing the record from which the new one is to be chained. You provide data management with the 5-byte file-relative address of this record by loading it into the KEYARG field before issuing the ADD macro (11.4.9).

Format:

LABEL	Δ OPERATION Δ	OPERAND
[name]	ADD	{ filename (1) 1 }

Positional Parameter 1:

filename
Is the label of the corresponding DTFIS declarative macro in your program.

(1) or **1**
Indicates that you have preloaded register 1 with the address of the DTFIS file table.

WAITF

11.5.3.3. Ensuring Completion of Record Transfer (WAITF)

Function:

The WAITF macroinstruction ensures that the transfer of a record between main storage and disk has been completed. It must be issued after you issue one of the following macros and before you attempt to process another record: ADD; WRITE,NEWKEY; READ,ID; READ,KEY; or WRITE,KEY. Any exceptional (error or status) conditions detected during the execution of the WAITF instruction are reflected in the DTFIS *filenameC* field when control is returned to you.

Format:

LABEL	Δ OPERATION Δ	OPERAND
[name]	WAITF	{filename / (1)}

Positional Parameter 1:

filename
Is the label of the corresponding DTFIS file table in your program.

(1) or **1**
Indicates that you have preloaded register 1 with the address of the DTFIS file table.

Example:

LABEL	ΔOPERATIONΔ	OPERAND	Δ
1	10	16	
	WAITF	EMPLMST	

11.5.4. Processing an ISAM File Randomly

You can retrieve individual logical records in random order for processing and updating. The record to be retrieved from the file is designated by its key or address and, in the case of an updating operation, is written back into the file.

You indicate the random retrieval (and updating) file processing function in the DTFIS macroinstruction by specifying IOROUT=RETRVE (or IOROUT=ADDRTR if new record insertions are also to be performed), and TYPEFLE=RANDOM (or TYPEFLE=RANSEQ if sequential processing functions are also to be performed).

The following imperative macroinstructions are used in the random processing of an ISAM file: READ, ID; READ, KEY; UPDT; WRITE, KEY; and WAITF.

READ, ID
READ, KEY

11.5.4.1. Retrieving a Record (READ, ID and READ, KEY)

Function:

The READ macroinstruction initiates the retrieval of a single logical record from an ISAM file. Before issuing the instruction, you must have stored the key or the address of the record to be retrieved in the main storage area equated to the KEYARG keyword parameter of the DTFIS declarative macroinstruction.

For indexed files, data management uses the argument for an indexed search and retrieval of a record with a matching key. ASAM files treat the argument as a relative address and retrieve the block and record. If a work area has been specified in the DTFIS macroinstruction, the logical record is transferred to the work area designated. If the IOREG keyword parameter is used, the address of the first character of the logical record is placed in the general register specified by IOREG.

To ensure that the retrieval operation has been completed, you must execute a WAITF macroinstruction before attempting to access the logical record retrieved.

Format:

LABEL	Δ OPERATION Δ	OPERAND
[name]	READ	{filename / (1) / 1} , {ID / KEY}

Positional Parameter 1:

filename
Is the label of the corresponding DTFIS file table in your program.

(1) or **1**
Indicates that you have preloaded register 1 with the addres of the DTFIS file table.

Positional Parameter 2:

ID
Indicates that random retrieval by location is performed.

KEY
Indicates that random retrieval by key is performed.

Examples:

LABEL (1)	OPERATION (10)	OPERAND (16)
	READ	EMPLMST,KEY
	WAITF	EMPLMST

Programming Considerations:

When control is returned to you after execution of the WAITF macroinstruction, the logical record associated with the argument in the area specified by KEYARG is available in either the work area or the I/O area, depending upon the DTFIS macroinstruction specifications. If the record is available in the I/O area, the register specified by the IOREG keyword parameter contains the address of the first character of the logical record. The disk address of the physical record is available at the address EMPLMSTG in the DTFIS file table. Indications of any exceptional (status or error) conditions are available at EMPLMSTC.

The dummy record containing an all-zero key and inserted by data management at file start is not available for you to retrieve. If you issue the READ, ID macro with the address of the dummy specified in the KEYARG field (11.4.9), data management sets the *invalid ID* error flag in *filenameC* and issues error message DM24 (INVALID REQUEST (ID)—OUTSIDE FILE LIMITS). If you issue the READ, KEY macro with a key of all binary 0's specified in the KEYARG field, data management sets the *record not found* flag in *filenameC* and issues error message DM31 (RECORD NOT FOUND FOR RANDOM FUNCTION). In either case, control transfers to your ERROR routine if you have specified one; otherwise, errors return to you inline. Refer to Appendix B.

WRITE, KEY

11.5.4.2. Updating a Record (WRITE, KEY)

Function:

The WRITE,KEY macroinstruction initiates rewriting (updating) of the last record retrieved with a READ macroinstruction. You must not alter the key field or the record length field (for variable records) in any way.

Format:

LABEL	Δ OPERATION Δ	OPERAND
[name]	WRITE	{ filename / (1) / 1 } ,KEY

Positional Parameter 1:

filename
Is the label of the corresponding DTFIS file table in your program.

(1) or **1**
Indicates that you have preloaded register 1 with address of the DTFIS file table.

Positional Parameter 2:

KEY
Indicates that the last record retrieved by a READ,KEY or READ,ID macroinstruction is to be rewritten in the file.

Example:

1 LABEL	Δ OPERATION Δ 10	16 OPERAND Δ
	WRITE	EMPLMST,KEY
	.	
	.	
	.	
	WAITF	EMPLMST

Programming Considerations:

In response to a WRITE, KEY macroinstruction, the updated logical record from the work area is moved to the correct location in the I/O area, and the block is rewritten. If the record is updated in the I/O area through the use of the IOREG keyword parameter, no move is required since the record is already in its correct location.

You must use a WAITF macroinstruction following the WRITE macroinstruction to ensure completion of the rewrite function. When control returns to you after the execution of your WAITF macroinstruction, the logical record last retrieved by a READ, KEY or READ, ID macroinstruction, as well as the block that contains it, will have been rewritten onto your disk file. Any indications of exceptional conditions (error or status) detected during the write and wait operations are available to you at address EMPLMSTC in the DTFIS file table.

If you have tagged the record for deletion, according to your own conventions, you may increment the 2-byte tagged-for-deletion field at address EMPLMSTT in the DTFIS file table either before or after the rewrite operation. This count is available for the life of the file to help you decide when file reorganization is beneficial.

UPDT

11.5.4.3. Updating Last Record Retrieved (UPDT)

Function:

You issue the UPDT imperative macro (which is exactly equivalent to the WRITE,KEY macro for updating randomly processed files) to rewrite to its original disk location in an ISAM or ASAM file the updated logical record last retrieved by a READ,ID or READ,KEY macroinstruction. You do not use the UPDT macro unless you have updated the record; in updating, you must neither alter the key nor change the length of the record. Like the WRITE,KEY macro, the UPDT macro must be followed by a WAITF macro before you issue another function to the file.

Format:

LABEL	Δ OPERATION Δ	OPERAND
[name]	**UPDT**	{ **filename** / **(1)** / **1** }

Positional Parameter 1:

filename
Is the label of the corresponding DTFIS declarative macro in your program.

(1) or **1**
Indicates that you have preloaded register 1 with the address of the DTFIS file table.

11.5.5. Processing an ISAM File Sequentially

You may also retrieve and update logical records sequentially. The first record to be retrieved may be designated by the beginning of the file, by a relative record disk address, by a specified key, or by any key greater than or equal to a specified value. The SETL macroinstruction specifies which kind of starting point is desired. Individual records are then retrieved in sequence by the GET macroinstruction. Where an updating operation is to be performed, the individual records are rewritten into the file by means of the PUT macroinstruction.

You indicate the sequential retrieval (and updating) file processing function in the DTFIS declarative macro by equating the IOROUT keyword parameter to RETRVE (or to ADDRTR if new record insertions are also to be performed), and the TYPEFLE keyword parameter to SEQNTL (or to RANSEQ if random processing functions are also to be performed).

To terminate a retrieval sequence, you issue an ESETL macroinstruction. This ensures that logical records committed to output by the PUT macroinstruction are written onto the disk. After the ESETL macroinstruction has been executed, another retrieval sequence may be initiated by executing a SETL macroinstruction. Also, if you have specified TYPEFLE=RANSEQ in the DTFIS declarative macro, the READ,KEY; READ,ID; and WRITE,KEY macroinstructions may be issued, once you have terminated the sequential mode.

SETL

11.5.5.1. Initializing a Retrieval Sequence (SETL)

Function:

The SETL macroinstruction initializes a retrieval sequence. It specifies the file from which the records are to be retrieved and the point at which the retrieval is to start. For the starting point, the SETL macroinstruction can select the beginning of the file or other file locations. When control is returned to you from the SETL macroinstruction, the retrieval sequence may start.

Format:

LABEL	Δ OPERATION Δ	OPERAND
[name]	SETL	{filename / (1) / 1} , {BOF / GKEY / ID / KEY}

Positional Parameter 1:

filename
Is the label of the corresponding DTFIS file table in your program.

(1) or **1**
Indicates that you have preloaded register 1 with the address of the DTFIS file table.

Positional Parameter 2:

BOF
Indicates that the retrieval sequence is to begin with the first logical record of the file.

GKEY
Indicates that the retrieval sequence is to start with the first logical record whose key is greater than or equal to the value in the area equated to the KEYARG keyword parameter of the applicable DTFIS macroinstruction. Used only for indexed files.

ID
Indicates that the retrieval sequence begins at the location given in the KEYARG keyword parameter in the applicable DTFIS macroinstruction.

KEY

Indicates that the area equated to the KEYARG keyword parameter in the applicable DTFIS macroinstruction holds the key of the first logical record to be retrieved. Used only for indexed files.

Examples:

LABEL (1)	OPERATION (10)	OPERAND (16)
	SETL	EMPLMST,GKEY
	SETL	EMPLMST,KEY

Programming Considerations:

When your file is an ASAM file, the KEY and GKEY positional parameters of the SETL macroinstruction should not be used.

The dummy record inserted by ISAM at file start, whose key is all binary 0's, is not available for you to retrieve. If you issue the SETL,BOF macro, or issue the SETL,GKEY imperative macro with an all-zero key specified in the KEYARG field, ISAM prepares to give you the first record *after* the dummy, and no error processing results.

On the other hand, if you issue the SETL,KEY macro with a search key of all binary 0's specified, data management sets the *record not found* error flag in *filenameC* and issues error message DM31 (RECORD NOT FOUND FOR RANDOM FUNCTION). If you issue the SETL,GKEY macro and the search key specified is greater than the highest key in the file, DM32 (RECORD NOT FOUND FOR SEQUENTIAL FUNCTION) is issued. If you issue the SETL,ID macro with the address of the dummy record specified as a search argument, data management sets the *invalid ID* error flag and issues error message DM24 (INVALID REQUEST (ID)—OUTSIDE FILE LIMITS). Control is transferred in either case to your ERROR routine if you have specified one; otherwise, to your program inline. Refer to Appendix B.

GET

11.5.5.2. Retrieving Next Logical Record (GET)

Function:

The GET macroinstruction retrieves the next logical record in sequence. It must be part of a valid retrieval sequence initiated by a SETL macroinstruction. If the block containing the next logical record is not already in main storage, the GET instruction reads it into the I/O area designated by the DTFIS macroinstruction. The DTF parameters you specify indicate preference for one of the following two modes of handling the logical record:

- By omitting the WORKS and IOREG keyword parameters, you choose to specify the work area that is to receive the logical record with every GET macroinstruction.
- When you specify the WORKS and IOREG keyword parameters, you choose to process the record in the I/O buffer area. The value specified in the IOREG keyword parameter is the number of the general register that points to the record.

If the GET instruction requires the reading of a new block, and if a PUT macroinstruction was executed for any logical record in the previous block, then the previous block, as updated, is written back onto the disk before the new block is read.

Format:

LABEL	Δ OPERATION Δ	OPERAND
[name]	GET	{filename / (1) / 1} [, {workname / (0) / 0}]

Positional Parameter 1:

filename
Is the label of the corresponding DTFIS file table in your program.

(1) or **1**
Indicates that you have preloaded register 1 with the address of the DTFIS file table.

Positional Parameter 2:

workname
Is the label of the work area into which the record is to be transferred.

(0) or **0**
Indicates that register 0 has been preloaded with the address of the work area into which the record is to be transferred.

Positional parameter 2 is not required if the WORKS keyword of the DTFIS declarative macro was equated to NO.

Examples:

	LABEL (1)	OPERATION (10)	OPERAND (16)
1.		GET	EMPLMST,EMRCD
2.		GET	EMPLMST

1. The next logical record of EMPLSMST is transferred into the work area labeled EMRCD. Any abnormal conditions are indicated by the bit settings in addressable field at location EMPLMSTC. The disk address from which the logical record was transferred is available to you in the addressable field at location EMPLMSTG.

2. The register equated to the IOREG keyword parameter (for example, register 4) holds the address of the first character of the logical record. Addressable fields EMPLMSTC and EMPLMSTG have the same significance as in example 1.

PUT

11.5.5.3. Updating a Record (PUT)

Function:

The PUT macroinstruction indicates that the last record retrieved by a GET macroinstruction has been updated and is to be rewritten on the disk. It must be part of a valid retrieval sequence initiated by a SETL macroinstruction and must follow a GET macroinstruction. Updating takes place in a sequential retrieval operation only through the execution of the PUT macroinstruction. If a record retrieved by a GET instruction is not updated, there is not need to execute a PUT instruction.

The PUT macroinstruction sets an indicator in the DTFIS file control table to ensure that a write is done before the present block is abandoned. The next block is not accessed until all logical records in the present block have been processed.

Like the GET macroinstruction, the PUT macroinstruction has two forms: the form that uses a work area and the form that uses an I/O area. If the DTFIS macroinstruction has specified use of a work area, then the PUT macroinstruction must specify the address of that work area from which an updated record will be transferred to the I/O area. This may be the same area specified in the preceding GET macroinstruction, or it may be a different area. Under no circumstances may the original record length be changed during update operations, nor may you alter the keyfield of the updated record.

If you have equated the WORKS keyword parameters of your DTFIS to NO, then you must also select a general register to be the current record pointer by specifying the IOREG keyword, and you will supply the address of the logical record in this register when you execute the previous GET macroinstruction. In this event, data management assumes that you udpated the record in the I/O area.

Format:

LABEL	Δ OPERATION Δ	OPERAND
[name]	PUT	{filename / (1) / 1} [, {workname / (0) / 0}]

Positional Parameter 1:

filename
Is the label of the corresponding DTFIS file table in your program.

(1) or **1**
Indicates that you have preloaded register 1 with the address of the DTFIS macroinstruction.

Positional Parameter 2:

workname
Is the label of the work area from which the record is to be transferred.

(0) or **0**
Indicates that you have preloaded register 0 with the address of the work area from which the record is to be transferred.

Positional parameter 2 is not required if the WORKS keyword of the DTFIS declarative macro was equated to NO.

Examples:

	LABEL (1)	OPERATION (10)	OPERAND (16)
1.		PUT	EMPLMST,(0)
2.		PUT	EMPLMST

1. The logical record last retrieved from EMPLMST is replaced by a record in the work area whose address is specified in register 0. An indicator in the DTFIS file control table is set to initiate rewriting of the block before the next record is read into the I/O area.

2. The logical record, whose address was supplied in the register specified by the IOREG keyword parameter when the preceding GET macroinstruction was executed, is assumed to have been updated.

ESETL

11.5.5.4. Terminating a Retrieval Sequence (ESETL)

Function:

The ESETL macroinstruction terminates a retrieval sequence initiated by a SETL macroinstruction. If there are any updated logical records that have not yet been rewritten, they are rewritten at this time. After the ESETL instruction has been executed, another retrieval sequence may be initiated by means of a SETL instruction.

Format:

LABEL	Δ OPERATION Δ	OPERAND
[name]	ESETL	{ filename (1) 1 }

Positional Parameter 1:

filename
Is the label of the corresponding DTFIS file table.

(1) or **1**
Indicates that you have preloaded the address of the DTFIS file table into register 1.

Examples:

LABEL (1)	ΔOPERATIONΔ (10)	OPERAND (16) Δ
	ESETL	EMPLMST
	ESETL	(1)

11.6. ERROR AND EXCEPTION HANDLING

11.6.1. FilenameC

Whether OS/3 data management detects certain errors or exceptions to file-processing performance, or it ascertains that the processing you requested was carried out exactly as specified, it makes appropriate entries in a 4-byte, full-word-aligned addressable field of the DTFIS file table, designated *filenameC.* You address this field by concatenating the character "C" to your 7-character logical file name and may use the BAL *test-under-mask* (TM) instruction to ascertain its contents.

Each of the 32 bits in the 4-byte field *filenameC* has its own significance as a status or error flag. It is your responsibility to provide the coding for testing the bits of *filenameC* and for taking action appropriate to the condition reported. If you have provided an error/exception handling routine, data management returns control to this routine in all events other than successful, unexceptional performance of the requested function. (In the absence of such a routine, control invariably returns to you inline.) You can avoid the need to check *filenameC* at the inline return points by including the necessary coding in your error routine. Table B—3 in Appendix B gives the meaning of the bits in *filenameC* that are set by OS/3 ISAM for you.

11.6.2. Other Addressable Fields of the DTFIS File Table

The OS/3 ISAM imperative macros also provide other information to you, in different fields of the DTFIS file table. Most of these actions have been discussed in the previous paragraphs describing these imperatives. Table 11—4 summarizes the information provided in these fields and indicates the length of each field and whether it is half-word-aligned or full-word-aligned. The individual fields are addressed by concatenating the character in the left-hand column of the table to your 7-character file name.

Table 11—4. Summary of Filename-Addressable Fields in DTFIS File Table (Part 1 of 2)

Field Addressable by Suffix to Filename	Content	Alignment	Length, in Bytes
A	Number of cylinders with full overflow area	H	2
C	Error and Exception Conditions — see Appendix B	F	4
G	Relative disc address from which the last record was retrieved	U	5
H	Relative disc address to which the last record was written	U	5
O	Total number of overflow records	H	2
P	Total number of prime data records	U	3

Table 11—4. Summary of Filename-Addressable Fields in DTFIS File Table (Part 2 of 2)

Field Addressable by Suffix to Filename	Content	Alignment	Length, in Bytes
R	Number of overflow records retrieved that were not first in the chain	H	2
S	Number of bytes required to hold top index in main storage	H	2
T	Number of records your program has tagged for deletion	H	2

LEGEND:

H Half-word aligned
F Full-word aligned
U Unaligned

11.7. PROGRAMMING EXAMPLE

11.7.1. Sample ISAM File Load Program

The following coding forms contain a sample program for the initial load of an indexed ISAM file, including the OS/3 job control statements you need to assemble and execute it. The logical name of the file in the example is "AFILE." A note may be in order about the boundary alignment of the storage areas and buffers shown in the example (REGS, WKSP, INDBUF, and DATABUF). The register save area, as you recall from 11.4.14, must be full-word-aligned aligned, and the others need to be aligned on half-word boundaries.

That they are so aligned in the example, without the usual full- or half word constants needing to be defined to skip bytes ahead of them and force alignment, is the result of special circumstances:

- The first of these areas (REGS) immediately follows the DTF.
- The OS/3 DTF generator automatically ensures that the DTF file table is full-word-aligned.
- The length of the DTF file table generated when you have specified IOROUT=LOAD is 332 bytes (11.3).

This is enough to ensure that REGS is full-word aligned; its 72-byte length and the even lengths of the other areas, in turn, cause them to be properly aligned. In the real world, of course, you will seldom find circumstances so neat.

Example:

```
LABEL     OPERATION     OPERAND                                          COMMENTS
1         10            16                                               72
                                                                         This is typical of the coding to be placed
                                                                         in the control stream at point A in the
                                                                         following coding form.
CREATE    START         0
          PRINT         GEN,DATA
BEGIN     BALR          2,0
          USING         *,2
          B             GO
AFILE     DTFIS         IOROUT=LOAD,BLKSIZE=512,RECSIZE=100,KEYLEN=12,      X
                        KEYLOC=15,PCYLOFL=20,INDSIZE=256,INDAREA=INDBUF,    X
                        IOAREA1=DATABUF,WORK1=WKSP,SAVAREA=REGS,ERROR=ERRX
REGS      DS            CL72                                             FULLWORD ALIGNED
WKSP      DS            CL100                                            HALFWORD ALIGNED
INDBUF    DS            CL256                                            HALFWORD ALIGNED
DATABUF   DS            CL512                                            HALFWORD ALIGNED
ERRX                                                                     ERR ROUTINE HERE
GO        OPEN          AFILE
          SETFL         AFILE
LOOP                                                                     ONE REC TO WORKSPACE HERE
          WRITE         AFILE,NEWKEY
          BC            ,LOOP                                            BR IF MORE RECS TO LOAD
          ENDFL         AFILE
          CLOSE         AFILE
          EOJ
          END           BEGIN
// JOB     EXAMPLE,,,8000,8000
// DVC     20            // LFD PRNTR
// WORK1
// WORK2
// EXEC    ASM
/$
                                POINT A,  PROGRAM CODE
/*
// WORK1
// EXEC    LNKEDT
/$
          LOADM         CREATE
          INCLUDE       CREATE,$Y$RUN
/*
// DVC     50            // VOL 123456
// EXT     IS,C,10,CYL,50,100
// LBL     'SAMPLE ISAM LOAD FILE'
// LFD     AFILE,2
// EXEC    CREATE,$YRUN
/&
// FIN
```

12. IRAM Formats and File Conventions

12.1. GENERAL

The indexed random access method (IRAM), a fifth access method in OS/3 for handling disk files, is available for programs written in RPG II language. RPG II programs compiled in the IBM System/3 mode automatically access all disk data files via IRAM; however, RPG II programs compiled in OS/3 native mode may advantageously specify the use of IRAM instead of other OS/3 disk access methods.

The main advantage in using IRAM in RPG II is the inherent saving of main storage. Only one of two IRAM processing modules is required: the smaller module (about 1350 bytes) provides random and sequential functions for processing nonindexed IRAM files. The larger module (about 2050 bytes) provides the same functions and also keyed functions for random and sequential processing of IRAM files created with indexes.

The functionality provided by IRAM is equivalent to that provided by OS/3 ISAM and ASAM, and by the OS/3 nonindexed access disk methods, SAM and DAM (relative record addressing); however, these modules are considerably larger than those of IRAM. It is also equivalent to that provided for sequential, direct, and indexed files processed with RPG II programs under IBM System/3. IRAM files may reside on any of the disk subsystems used with OS/3. An IRAM file may occupy from one to eight disk packs, which must be of the same type.

The IRAM processor can access only disk files it has created or files created by the MIRAM processor that have IRAM characteristics. It cannot access disk files that have been created by the OS/3 ISAM, ASAM, DAM, or SAM access methods, nor can IRAM files be processed by these access methods. IRAM files can be processed using the OS/3 independent sort/merge program, however, and by the data utilities program. RPG II users converting to OS/3 from IBM System/3, moreover, may transcribe existing data files to IRAM format by using procedures detailed in the System/3 to OS/3 transition user guide/programmer reference, UP-8379 (current version).

12.1.1. IRAM Concepts

A number of features and concepts distinguish IRAM from other disk access methods:

- Data records in IRAM files are of uniform, fixed size and may span physical blocks and sectors, tracks, and even cylinders as required. They may extend from one volume onto another (unless the file is created for processing only a single volume at a time).

- Data records are written on disk compactly, as a continuous string of bytes, without any space in the string being used for system control or overhead. The data string is enlarged only by appending records.

- The string of data records is always usable in sequential and direct access modes, with direct access being made by a file-relative record number. Moreover, the data can be specified to be indexed by key; this causes IRAM to build a suitable index structure that resides in a second partition, separately from the data.

- An indexed IRAM file can be referenced by the additional modes of random-by-key and sequential-by-key.

- Indexed IRAM files, multivolume or single-volume, may be created by means of an orderly load (records submitted in ascending order of keys) or a disorderly load (record keys in no particular order), and they may be extended by appending records in either manner. If orderly load or extend is specified, automatic sequence checking is performed and results in immediate rejection of any record with an out-of-sequence key.

- Duplicate keys are not permitted; a record with a key duplicating one already in the file is rejected immediately. IRAM does not sort the index at the completion of a disorderly load, but maintains the index current on a record-by-record basis.

- When a new record has been added to an indexed or nonindexed IRAM file, it is immediately available for retrieval.

- Multivolume IRAM files may be created for processing with either one volume online at a time, or with all volumes online. They must be processed in the same manner as created.

- All programs accessing an IRAM file need not use the same data buffer size for I/O as was used to create the file. However, all those accessing an indexed IRAM file must use the same index buffer size (unless they are not issuing keyed functions).

- IRAM's restrictions are the following:

  — All records are fixed-length.

  — Duplicate keys are rejected in indexed files.

  — The maximum key length is 80 bytes (RPG II allows only 29 bytes, however). No byte of a record key may contain the hexadecimal value 'FF'.

  — The minimum size for the index buffer is 256 bytes.

  — No IRAM function is provided for deleting records — logical deletion and physical removal are responsibilities of the user's programs.

The following subsections describe the IRAM record and file formats and conventions and the processing of sequential, direct, and indexed IRAM files.

12.2. IRAM FILE CONVENTIONS AND FORMATS

A nonindexed IRAM file contains only one partition, the data partition, which IRAM defines to the system access technique (SAT) as containing 256-byte physical blocks, unkeyed. An indexed file, on the other hand, contains yet a second partition, the index partition, specified to SAT as containing 256-byte keyed blocks. The data partition of an indexed IRAM file is laid out exactly like that in a nonindexed file; it precedes the index partition, which begins on a separate cylinder from it.

12.2.1. The Data Partition

The data partition, cylinder aligned, consists of a single, compact string of data records, spanning the sector or physical block boundaries as necessitated by their uniform, fixed lengths. Records may be keyed or unkeyed; they contain no bytes reserved for control overload, nor are there any such nondata bytes in the string. Figure 12—1 shows the appearance of IRAM data records and lists the rules for their structure.

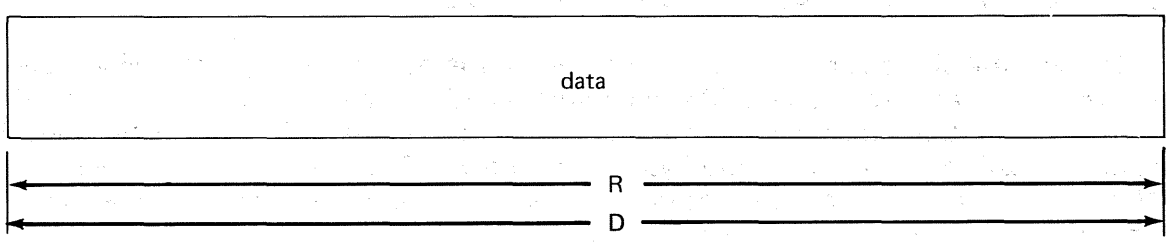
When these data records are stored in an IRAM file, the records are loaded consecutively, arranged in the same order as you originally presented them to the IRAM processor. Although the records are stored in 256-byte sectors on your fixed-sector disk packs, and in 256-byte physical blocks on variable-sector disks, the data records may span these physical boundaries. Record-length and sector-length need not coincide, as shown in Figure 12—2. Your data records are also independent of track boundaries, cylinder boundaries, and even volume boundaries (except when a multivolume file is created for single-volume processing). The data partition of your IRAM file is a long continuous string of data records. When new records are added, they are *appended* to the string, that is, added at the end as a continuation of the original string of records.

12.2.2. Entries in the Index Partition

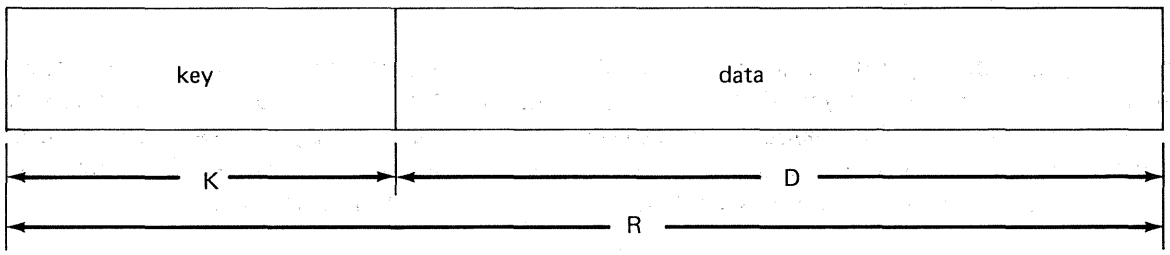
As it loads your keyed records into the data partition of an indexed file, IRAM extracts the key from each record and constructs a 3-byte pointer from the file-relative record number of the position to which the record is written. From these it forms an index entry for each record, and stores the entry in the index partition. The index entry for each record is then equal to the specified key length plus three bytes; it is stored in what is called the *fine level* of the IRAM index.

The fine level of index is not formatted for hardware search, unlike the other levels of index described in subsequent paragraphs. It is treated as a chain of multisector blocks (each sector being 256 bytes long, as previously stated). Initially, each fine-level index block is only partly filled (to just beyond the three-quarter level) with index entries so as to permit later insertion of new entries, as required. All entries in the fine-level index are maintained in ascending key order. Figure 12—3 represents a fine-level index block comprising three 256-byte sectors.

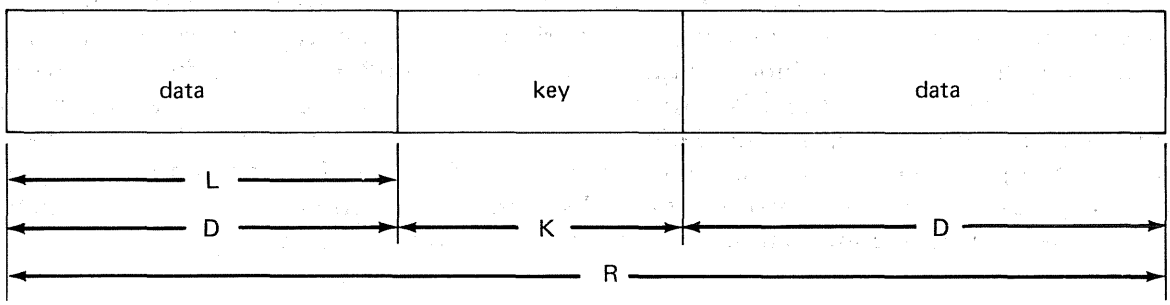
Without Key



Key at Head of Record



Key Internal to Record



LEGEND:

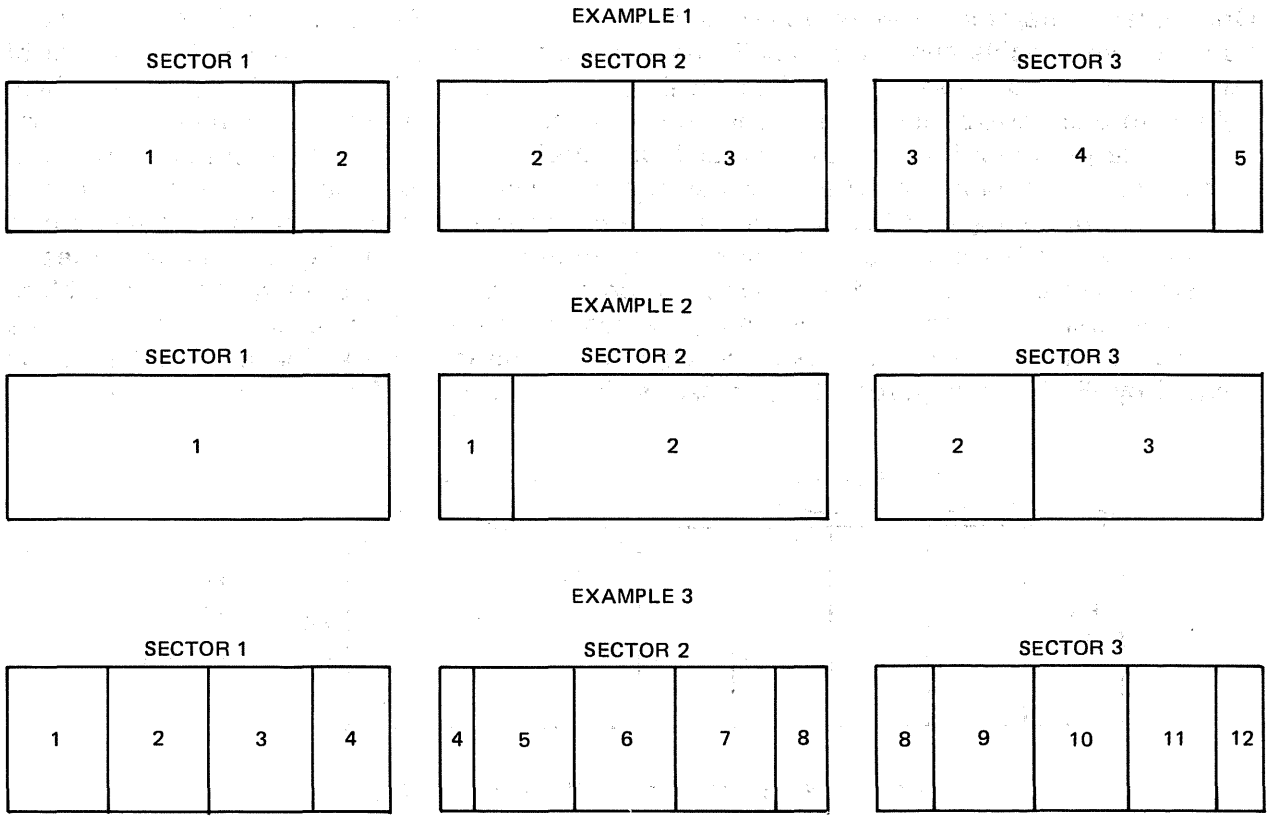
K = Record key. All keys in a keyed file must have the same length; each record in a keyed file must have one unique key; and the starting location of the key must be the same in each record. You specify the length of the key, which may range from a minimum length of 3 bytes to a maximum of 80. (The maximum key length for RPG II records is 29 bytes.) No byte of any key may contain the hexadecimal value 'FF'.

L = Key location. The starting location of the key must be the same in each record. You may specify the number of bytes of data preceding the key. If you default, IRAM assumes the key starts in the first byte of the record.

D = Data portion of your logical record

R = Length of logical record (key plus data). You specify this length, measured in bytes. All records in an IRAM file must have the same length.

Figure 12—1. IRAM Data Records with and without Keys



NOTES:

All sectors equal 256 bytes.
Records in example 1 equal approximately 190 bytes each.
Records in example 2 equal approximately 300 bytes each.
Records in example 3 equal approximately 70 bytes each.

Figure 12—2. IRAM Data Records Spanning Disk Sectors on a Fixed Sector Disk

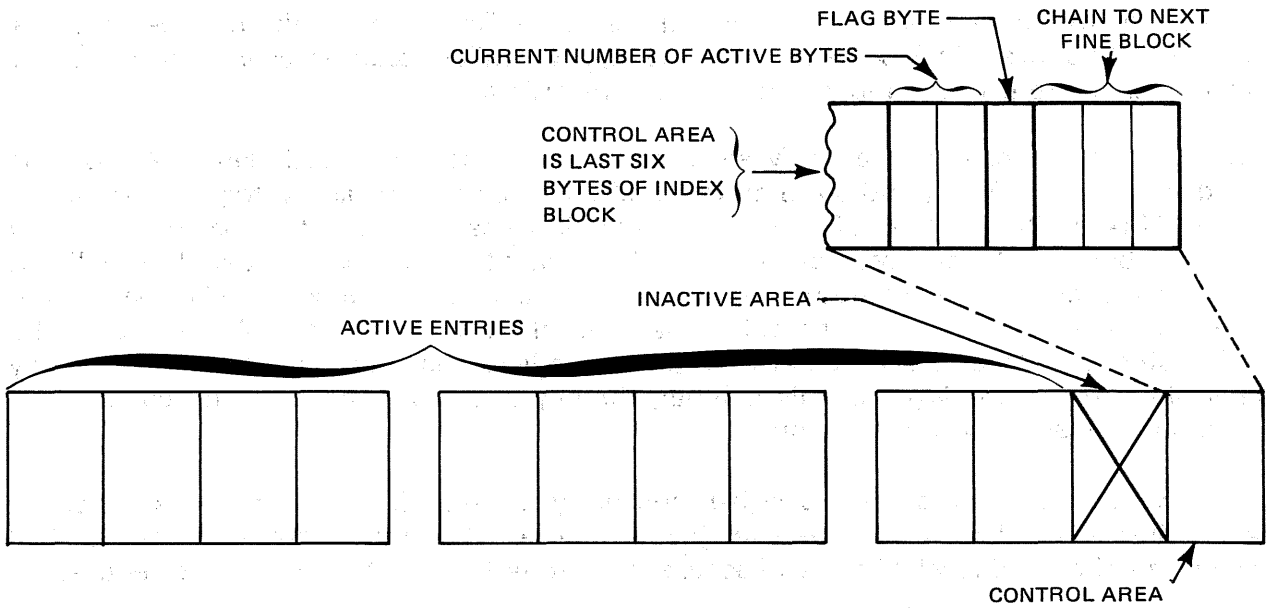


Figure 12—3. Typical Fine-level Index Block of Three Sectors

One other hierarchic level of index is always created: the coarse-level index. This is hardware searchable and comprises 256-byte blocks, each containing a variable number of entries similar to those at the fine level. One difference, however, is that the 3-byte pointer in each coarse-level entry does not represent the file-relative number of a record in the data partition; it points to another index block at a lower level — either a fine-level index block, or a block in what is called the mid-level index. Another difference is that, instead of containing a 6-byte control area, each coarse-level block uses its final byte to indicate the number of bytes in the block that represent active entries. The index entries in a coarse-level block are filed in descending order of key values, the high key of the block being the first encountered by the hardware search. Mid-level index blocks have the same construction as those in the coarse level; refer to Figure 12—4. The mid-level index is created by IRAM as required; the process is described in the following subsection.

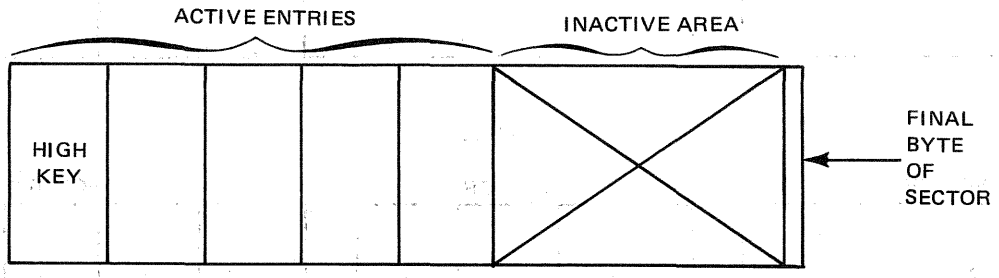


Figure 12—4. Typical Coarse- or Mid-Level Index Sector

12.2.3. Structure of IRAM Index

When the IRAM processor builds an index for your file, it creates at least two levels of index: a fine level and a coarse level. If your file is very large, one or more mid-levels of index are created as they are needed.

The fine level of index consists of an entry for every record in the data partition of your file. The fine-level entries are filed in ascending key order until an index block (256 bytes) is filled. At this time, one coarse-level entry is made that points to the high key entry in that filled fine-level block. As each fine-level block is filled, another coarse-level entry is made. This pattern is continued until all your records are on file.

The coarse-level index is arbitrarily allocated by IRAM; its size is disk-dependent. On the fixed-sector 8415, 8416, and 8418 disks, IRAM allocates two tracks; on the variable-sector 8411, 8414, 8424, 8425, 8430, and 8433 disks, it allocates four. If the coarse-level index is filled before all your records are on file, a mid-level index is created. The IRAM processor allocates a new track, designates it as a mid-level index, and copies three-quarters of the entries from the filled coarse-level track onto this mid-level track. The IRAM processor creates a new entry in the coarse-level index that points to the high key of a block in the mid-level. In this manner, three-quarters of a track on the coarse-level index are replaced by a single entry.

As new fine-level entries are recorded, one entry is made in the coarse-level index for each filled index block in the fine level, just as before. When the coarse-level track is filled again, another mid-level track is allocated, three-quarters of the coarse-level entries on

that track are moved into the mid-level index track, and one new coarse-level entry is created to point to the high key of the second track of the mid-level index. This process can be repeated until each entry in the coarse-level index points to the high key of a track in the mid-level index. Figure 12—5 illustrates the structure of an IRAM index.

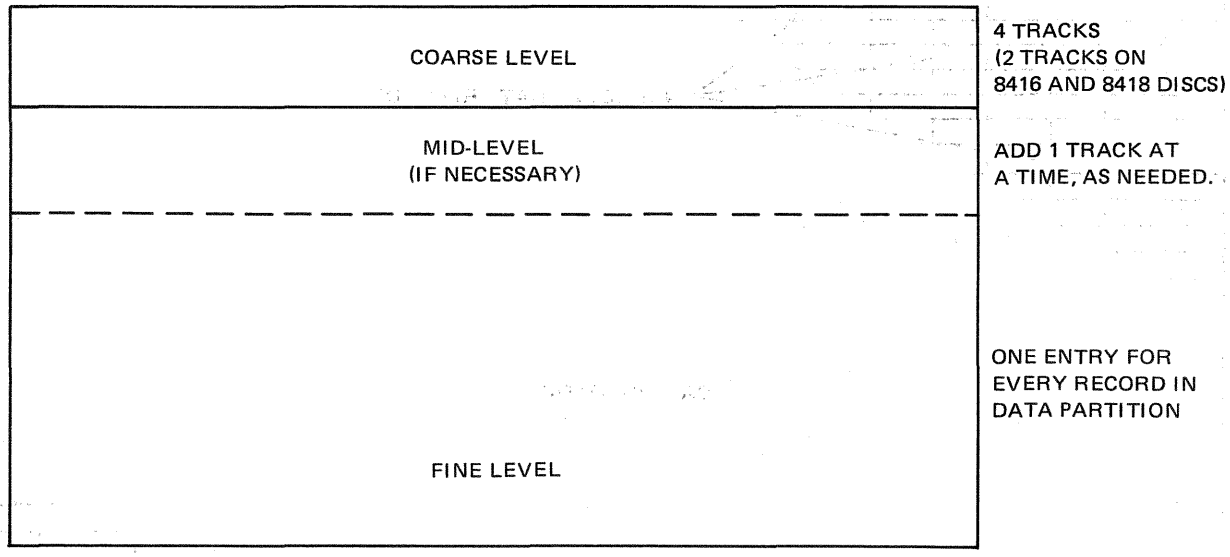


Figure 12—5. IRAM Index Partition

At this point, the addition of new records would cause another mid-level index to be created between the filled coarse-level index and the old mid-level index. A search for a specific data record by key in a 4-level index would proceed as follows (refer to Figure 12—6):

- the search begins in the coarse-level index;
- a hit is made that points to the new mid-level index;
- the new mid-level is searched;
- a hit is made that points to the old mid-level index;
- the old mid-level is searched;
- a hit is made that points to the fine-level index;
- the fine-level is searched;
- a hit is made which points to the data record in question; and
- the data record is retrieved.

It is not likely that your file would be large enough to require more than three index levels, but IRAM can create an index for any size file up to the physical limitation of eight disk devices.

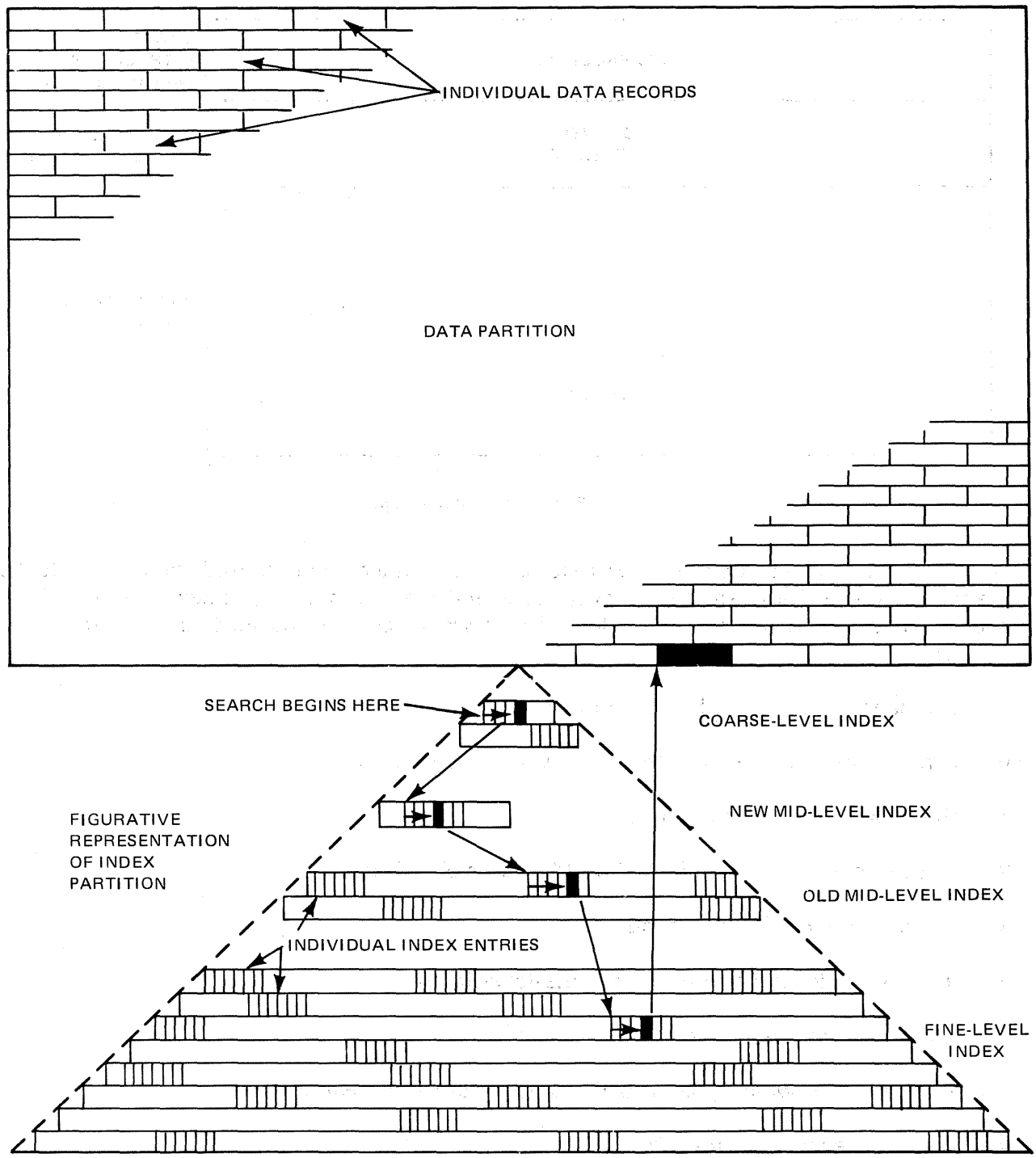


Figure 12—6. Typical Search of 4-Level IRAM Index

12.2.4. Estimating Disk Space Required for an Indexed IRAM File

To estimate the number of cylinders for your primary allocation of disk space to an indexed IRAM file, follow the straightforward procedure outlined herein. The result is a good first-level approximation for your use in specifying the EXT statement in the job control device assignment set that is used in allocating disk space for an indexed IRAM file to be created by your program. The same calculations may also be used for specifying the number of cylinders to be allocated for an indexed IRAM file to be generated from another file by the OS/3 data utility program (as in the task of transcribing your data files in the System/3-to-OS/3 transition process).

The number of cylinders required for an indexed IRAM file includes those occupied by the data partition and the index partition; the latter comprises the fine-level index and the top levels of index (the coarse-level and the mid-level, if any). Your initial space allocation for an indexed IRAM file must always be at least two cylinders because the data partition and the index partition are separate, and each is cylinder-aligned. To obtain the approximate size of the space the file will occupy, proceed as follows.

First, calculate D, the number of 256-byte sectors required for your data records:

$$D = \frac{\text{record-length} \cdot \text{number-of-records}}{256} \qquad (1)$$

Next, calculate B, the number of index blocks required by your fine-level index:

$$B = \frac{\text{number-of-records} \cdot (\text{keylength} + 3)}{(256 \cdot m) - 6} \cdot \left(\frac{4}{3}\right) \qquad (2)$$

where:

the factor 4/3 is used because the average fine-level index will be 3/4 full. m is the number of 256-byte sectors in the index buffer. (See 13.2.1.)

Next, calculate F, the number of 256-byte sectors required by your fine-level index:

$$F = m \cdot B \qquad (3)$$

NOTE:

Over 95 percent of the file allocation is used by the sum of the data requirements (D) and the fine-level requirements (F). Unless you need a more accurate estimate, skip the calculations to obtain the number of 256-byte sectors required for the coarse level (T) and the mid-levels (S) of the index, and do your final calculation as described in step 6 with T and S set to zero.

Next, do the following calculations to obtain the number of 256-byte sectors required for the top levels of the index — the coarse-level, T, and the mid-level, S.

The coarse-level calculation is automatic: IRAM always sets aside two tracks of coarse-level index for an indexed IRAM file that resides on an 8416 or 8418 fixed-sector disk; it sets aside four tracks for a file on a variable-sector disk (8411, 8414, 8430, or 8433). The number of sectors these tracks contain are found in column T of Table 12—1. If they can contain enough index entries to point to all your fine-level index blocks, no mid-level index is needed.

On the other hand, if the number of fine-level index blocks exceeds what the maximum coarse-level index can control, IRAM automatically creates one or more tracks of mid-level index, one at a time, as it finds the need to control the remainder. You may proceed as follows to calculate the number of sectors these mid-level index tracks will contain:

- Take (from Table 12—1) the value T, which represents the number of sectors allocated to the coarse-level index for a file on the type of disk this index is to reside on, calculate E, the number of index entries that T can contain, and compare this number to B:

$$E = \left[\frac{255}{(\text{keylength}+3)}\right] \cdot T \qquad (4)$$

  where:

  [ ] implies the integer value only: no remainder.

  If E is equal to or greater than B, no mid-level index is required. The value S (the number of sectors required for the mid-level) in step 6 is set to zero.

- On the other hand, if E is less than B, a mid-level index will be constructed; proceed to calculate S, the number of sectors it will contain:

$$S = \frac{B - E}{\left[\frac{255}{\text{keylength} + 3}\right]} \cdot \left(\frac{4}{3}\right) \qquad (5)$$

  where:

  The factor of 4/3 is used because the average mid-level index will be 3/4 full.

The final calculation of the number of cylinders to allocate to the whole file is represented by the following formula:

$$C = \frac{(F + T + S)}{U \cdot N} + \frac{D}{A \cdot N} \qquad (6)$$

where:

C

Is the number of disk cylinders to allocate to the indexed IRAM file.

A

Is the disk dependent number of 256-byte sectors per track (data partition) from Table 12—1.

D
Is the number of 256-byte sectors required for the data partition.

F
Is the number of 256-byte sectors required by the fine-level index.

T
Is the disk-dependent number of sectors allocated automatically for the coarse-level index, from Table 12-1.

S
Is either zero (when no mid-level index is required), or the number of mid-level index sectors required.

U
Is the disk-dependent number of 256-byte sectors per track, from Table 12-1.

N
Is the disk-dependent number of tracks per cylinder, from Table 12-1.

Example:

Given the following parameters, calculate the number of cylinders to allocate for an indexed IRAM file residing on an 8416 disk:

Number of records: 77,500

Record length: 512 bytes

Key length: 28 bytes

Index buffer length: 512 bytes

- $$D = \frac{\text{record-length} \cdot \text{number of records}}{256} \qquad (1)$$

  $$= \frac{512 \cdot 77{,}500}{256}$$

  = 155,000 sectors for data partition.

- $$B = \frac{\text{number-of-records} \cdot (\text{keylength}+3)}{(256 \cdot m)-6} \cdot \frac{4}{3} \qquad (2)$$

  $$= \frac{77{,}500(28+3)}{(256 \cdot 2)-6} \cdot \frac{4}{3}$$

  = 6331 blocks for fine-level index.

- $F = m \cdot B$ (3)

  $= 2 \cdot 6331$

  $=$ 12,662 sectors for fine-level index.

- The coarse-level index for an IRAM file on an 8416 disk contains 80 sectors; by inspection this number can be seen to be too small to avoid the need of a mid-level index.

- $E = \left[\frac{255}{(\text{keylength}+3)}\right] \cdot T$ (4)

  $= \left[\frac{255}{(28+3)}\right] \cdot 80$

  $=$ [8.22] · 80 = 8 · 80 = 640 index entries can be contained by T.

- $S = \frac{B - E}{\left[\frac{255}{(\text{keylength}+3)}\right]} \cdot \frac{4}{3}$ (5)

  $= \frac{6331-640}{8} \cdot \frac{4}{3}$

  $=$ 949 sectors for mid-level index.

- $C = \frac{(F + T + S)}{U \cdot N} + \frac{D}{A \cdot N}$ (6)

  $= \frac{(12{,}662 + 80 + 949)}{40 \cdot 7} + \frac{155{,}000}{40 \cdot 7}$

  $= \frac{168{,}691}{280}$

  $=$ 603 cylinders to be allocated for file.

12.2.5. Estimating Disk Space Required for a Nonindexed IRAM File

To estimate the number of cylinders to be allocated for an IRAM file created without an index, proceed to calculate D, the number of 256-byte sectors required for your data records, using formula 1 in the preceding subsection, and divide by the product of A times N (taken from Table 12-1):

$$C = \frac{D}{A \cdot N} \quad (7)$$

Table 12—1. Disk-Dependent Factors for Calculating Size of Top-Level Index for an IRAM File

SPERRY UNIVAC Disk Subsystem	U (Number of 256-byte sectors per disk track for index partition)	A (Number of 256-byte sectors per disk track for data partition)	T (Number of sectors allocated to course-level index)	N (Number of tracks per disk cylinder)
8416	40	40	80	7
8418	40	40	80	7
8411	10	11	40	10
8414	17	20	68	20
8424	17	20	68	20
8425	17	20	68	20
8430	29	33	116	19
8433	29	33	116	19

13. Functions and Operations of IRAM

13.1. PROCESSING NONINDEXED IRAM FILES

Nonindexed IRAM files are created with specific processing requirements in mind. A *sequential* nonindexed file is one in which the physical, consecutive order of records on disk is of specific significance to your application, and you expect to process these records one after the other. A *direct* nonindexed file, on the other hand, is one arranged on disk so as to provide ready access to a specific record without processing any of the records preceding it. The consecutive, physical order of records in the direct file is usually of less importance than your ability to access each record at random, independently of any other record in the file.

A capability that sets IRAM off from other disk access methods, however, is that record retrieval, update, and other operations on nonindexed files may be performed consecutively or randomly, regardless of the primary purposes for which files were created. The direct file created randomly by relative record number has several special characteristics that need separate consideration; for this reason, the processing procedures for the randomly processed and sequentially processed consecutive file are taken up separately in the following paragraphs.

Nonindexed IRAM files spanning two or more volumes may be created with only one volume mounted at a time, or with all volumes mounted. They must always be processed in the same way as they were created: only one volume online, or all online.* However, it is important to realize that the nonindexed multivolume file intended for single-volume processing may not be created randomly by relative record number, nor may it be processed this way. Multivolume files for which relative record addressing is planned must have all volumes mounted — whether they are sequentially processed or randomly processed consecutive files.

All IRAM files may be processed with a randomly-ordered disk file that contains relative record addresses; this type of file is known to the RPG II programmer as a tag, or ADDROUT, file. You may create such a file from an IRAM file by means of the ADDROUT option of the independent OS/3 sort/merge program; this process is documented in the sort/merge user guide, UP-8342 (current version).

For details of the actual programming specifications you must use for creating and processing IRAM files in OS/3 using RPG II, refer to the RPG II programmer reference, UP-8044 (current version).

**The IBM System/3 programmer recognizes the file created to be processed with all volumes mounted as an online multivolume file; the file created for single-volume processing he recognizes as an offline multivolume file.*

13.1.1. Processing Sequential IRAM Files

A sequential file is one organized consecutively, its records being written on the disk in the physical order in which you provide them to IRAM. They are usually processed consecutively, one after the other, in the order in which they occur on the disk, and usually all the records in the file are processed. IRAM furnishes you functions for consecutive processing, but also provides functions for random processing, by relative record number. The following subsections discuss procedures for creating and extending a sequential IRAM file; for adding, retrieving, updating, and deleting records; and for reorganizing a sequential IRAM file.

13.1.1.1. Creating a Sequential IRAM File

You define your IRAM file as an output file to be created in a sequential (consecutive) file processing mode, and you specify the uniform size of your fixed-length records and the size of your data buffer (or buffers). You may use two contiguous I/O areas, each half-word aligned, for double buffering if you desire, but they must have the same length.

To calculate the minimum size that you may specify as data buffer length, use the following algorithm:

- If record size divides into 256 bytes without remainder, minimum buffer size is 256 bytes.
- If record size is a multiple of 256 bytes, the minimum buffer size equals record size.
- Otherwise, minimum buffer size depends on the sum of record size + 255 bytes. If the sum is a multiple of 256 bytes, then the minimum buffer size equals this sum. If not, you must round this sum upward to the next multiple of 256 bytes; this, then, is the minimum buffer size.

The same algorithm is used for minimum data buffer length calculations in creating direct and indexed IRAM files as well. To specify data buffers larger than the minimum may enhance your program's performance. Note that subsequent programs processing this file need *not* specify the same data buffer size that you use to create the file, but they must, of course, specify at least the minimum.

After the file is opened, your records are submitted to IRAM, one after the other, until you have no more records available. IRAM stores them in the data partition in the order of submission — this is the consecutive order in which you may process them later. The relative record number of the last record written is recorded by IRAM in the file control table and the volume table of contents. Records are stored as a single, compact string of bytes, without any space in the string unused or devoted to system overhead. IRAM performs no sequence checking or duplicate record rejection in loading a sequential IRAM file; these functions, if they are necessary, are up to your programs.

13.1.1.2. Extending a Sequential IRAM File

Once the string of data records has been created, it may be enlarged through IRAM only by appending new records at the end. IRAM does not provide functions for inserting new records within the string, nor for adding them at the head of the string. Records may not be appended during retrieval processing.

The usual method of appending new records to a sequential file is essentially the same as file creation. Again, the file is defined as an output file for sequential processing, and the same specifications made as before (except, as previously noted, that this program need not use data buffers of the same size as the file-creating program used). After the file is opened, your new records are submitted to IRAM and stored in consecutive order, as in file creation; all are appended to the data string. This procedure requires you to specify the EXTEND option in the LFD job control statement for the file.

It is also possible, in IRAM, to extend a sequential file in random mode — again by appending each new record to the end of the string. For this manner of enlarging the file, you redefine it as a chained output file for random (direct) processing, and you define a field in your program to be used for providing a relative record number for each new record IRAM is to append. You must also define a record work area in your program (equal in size to one record length) in which you make each record available to IRAM. When the file is opened, IRAM automatically initializes the relative record number field to the next record number available for file extension. You use this for the first record to be appended, incrementing the field by 1 for each succeeding record before you issue the output function to append it to the file.

13.1.1.3. Adding Records to a Sequential File

When you have to enlarge a sequential file by inserting new records between existing records, or by adding them at the head of the string, you must make use of processors other than IRAM, which has no sequential or random functions for performing these tasks. Your new records must be sorted into the consecutive sequence in which you expect to process them, and be provided as an input file (together with your IRAM file) to either the OS/3 independent sort/merge program or the OS/3 data utilities program. Either of these processors can be used to create a new sequential IRAM file containing your inserted or added records in their proper consecutive order. Refer to current versions of the OS/3 sort/merge user guide, UP-8342, or the OS/3 data utilities user guide/programmer reference, UP-8069, for the details of these procedures.

13.1.1.4. Retrieving and Updating Records in a Sequential IRAM File

You may retrieve or retrieve-and-update records in an IRAM file either in sequential (consecutive) order or at random (using relative record number). Recall that if yours is a multivolume file, however, it must have been created for processing with all volumes mounted if you are to process it via relative record number (see 13.1). For consecutive retrieval, define the IRAM file as a nonindexed input file for sequential (consecutive) file processing mode. For consecutive retrieval-and-update, define the IRAM file as a nonindexed update file for sequential processing. If your processing is to begin elsewhere

than at the first record of the file, you must predefine a field in your program where you must provide the relative record number of the starting point to IRAM by setting a lower limit for processing before you issue the input function (these actions are not required when your program is to read all the records). Your retrieval may begin at any record number that is not higher than the current file limit; once this record has been retrieved, however, the remainder of the records in the file are read automatically in unbroken, consecutive order, unless you select a second starting point.

You may provide two data buffers if you are only retrieving records without updating, but you must provide only one data buffer if you are updating. The lengths of these buffers must be the same, but need not equal the data buffer size used to create the file.

IRAM provides data records to you in the data buffer in the same consecutive order in which it received them at file creation or extension, and in the same order that they were written on the disk. Consecutive retrieval continues until the file is closed or the end of file is reached; if it is to be terminated by the end-of-file condition, you must have specified the address of a routine for handling this event. If the relative record number you specified as the starting point lies outside the file limit, retrieval does not take place; control is transferred to an error routine with status flags set to indicate a no-find condition.

Random retrieval by relative record number from a sequential file requires that you define the file as a chained input file for random processing by relative record number. Your program defines a field in which you will supply the desired relative record number to IRAM and moves this number into the field before you issue the input function. If you request a number that lies outside the current file limit, IRAM transfers control to an error routine with status flags set to indicate a no-find condition. Random retrieval terminates when the file is closed.

For random retrieval-with-update from a sequential file, you proceed as for random retrieval, but you may predefine a field in your program as a record work area (of a size equal to record length) from which you will make the updated record available to IRAM instead of using the data buffer. (Only one data buffer may be used in the update mode.) IRAM will make the retrieval record available to you in this record work area, instead of in the data buffer, if your program specifies if before you issue the input function. You retrieve by supplying IRAM with the desired relative record number in the predefined field of your program before you issue the input function. The output function to write the changed record back to the file does not require a relative record number and none should be supplied.

Updating records in any IRAM file may include marking records that have become inactive with a deletion flag — according to your own conventions. Such records will always be retrieved; therefore, if your update processing does include the writing of a deletion flag, then your retrieval programs must include checking for the presence of this flag to determine whether each record retrieved is to be bypassed or processed.

Records may not be appended during retrieval or retrieval-and-update operations.

13.1.1.5. Deleting Records from a Sequential IRAM File

IRAM does not provide a function for logically deleting records from a file nor for physically removing them from the unbroken string of data records. As records become inactive or otherwise eligible for removal from the file, your update programs should mark them in some way (of your own choosing) so that they can be recognized for bypassing — and for eventual physical removal when you reorganize the file. A common method for logically deleting a record is to write a conventional deletion flag in a specific byte of the record. Because IRAM has no means of checking this flag for you, logically deleted records will always be retrieved; your programs must check for the presence of the delete code.

13.1.1.6. Reorganizing a Sequential IRAM File

A sequential file may eventually require reorganization. One reason may be to conserve disk space by physically removing records that have been logically deleted from the file; another may be to reorder the file according to a more convenient physical sequence than was used to create or extend it.

Two methods are available to you for reorganizing a sequential file. Either the independent sort/merge program or the data utilities program will accept your IRAM file as input and resequence the data records, physically deleting records you have tagged. For details on these processors, refer to the current versions of the sort/merge user guide, UP-8342, and the data utilities user guide/programmer reference, UP-8069.

13.1.2. Processing Direct IRAM Files

A direct IRAM file is one organized to allow any record in the file to be retrieved directly when the location of the record, in relation to the beginning of the file, is specified. The IRAM processor does not search an index or process other records that precede the one to be retrieved; all records in the file are assigned to specific file-relative positions, independent of the order in which they are presented to IRAM for writing to the file.

IRAM provides not only functions for direct or random processing but also functions for consecutive processing of records in a direct file. The following subsections diskuss procedures for creating and extending a direct IRAM file; for adding, retrieving, updating, and deleting records; and for reorganizing the file.

13.1.2.1. Creating a Direct IRAM File

When your plans for processing a nonindexed IRAM file call for each record to be assigned to a specific disk location, you create a *direct* file by presenting your records, one by one, to the IRAM processor in a work area, and by supplying the file-relative position to which each record is to be written on disk. This position is not a disk address, but a file-relative record number that you supply to IRAM in a predefined field of your program before your issue each output function to write a new record to disk. The RPG II programmer knows this type of file as a chained output file.

The relative record number that you assign to each record as it is written out may have been determined directly from some control field in the new record itself, or it may have been derived from the record by a conversion process that you have programmed separately. Or, it may have been obtained from some other source entirely: a control field in a record in some other file. In any event, it is entirely possible that a number of the relative record positions available in the extent you have allocated to this direct IRAM file will not be occupied by a data record in your initial load. It is also possible, on the other hand, that your method of obtaining a relative record number may result in your assigning the same number to two or more records.

When you are creating a direct IRAM file randomly by relative record number, IRAM does not check for duplicate assignment of relative record numbers, nor does it detect or prevent two records being loaded into the same position in the file. Later addition of records or extension of the file may, likewise, result in overwriting record positions already containing valid records that you do not intend to be "updated" in this backhand way. Such problems must be dealt with in your programming and in your preparation of the file before you load.

IRAM preformats the extents of direct files residing on the variable-sector 8411, 8414, 8424, 8425, 8430, and 8433 disk subsystems, initializing or "dummying" all relative record positions to binary 0; it does not do so, however, for files on the fixed-sector 8415, 8416, and 8418 disks. Record positions on the latter disks will contain spurious data (from your point of view) resulting from previous uses of the disk, or residual patterns from prior disk prep or testing routines. Before loading a direct IRAM file onto an 8415, 8416, or 8418 fixed-sector disk, you should therefore take care of dummying with a file preparation program of your own: one that writes into all record positions the null pattern (blanks, zeros, or whatever you determine) that your subsequent load program will recognize as virgin territory. In this way, the algorithm in your load program for detecting and preventing the collision of synonyms will be more likely to work as you intend.

In calculating the minimum size of your data buffers, follow the same procedure as for a sequential file, described in 13.1.1.1.

13.1.2.2. Extending a Direct IRAM File

You enlarge or extend a direct IRAM file, which has been created randomly by relative record number, in the same way as you created it. That is, provide each new record to IRAM, singly in a work area or I/O area, and supply the relative record number for the record before you issue the output function to write it, placing this number in a predefined field in your program.

When a direct IRAM file is opened for extension in random mode, IRAM automatically initializes the predefined relative record number field in your program by moving into it the next available record number. This number is the next relative record position where IRAM expects to write the first data record to be appended to the string of records in the file. It has recorded this record position in the volume table of contents (VTOC) of the file and in the file control table as an end-of-data address.

You must use this IRAM-supplied relative record number with the first new record you add to the file, and each successive relative record number you provide (by incrementing the content of this field) must be 1 higher than the current highest numbered record. Therefore, if you are enlarging your file by adding data records by relative record numbers that you are calculating or obtaining by algorithm, your file extension program should provide a dummy record for each consecutive relative record position to which a record containing actual data is *not* to be written. (An alternative, of course, is to extend your direct file with all dummied records, to be updated later at random by actual data records whose relative record numbers you provide by the same process used in file creation.) In either case, the dummying is your own convention, recognized by your programs — not by IRAM.

Records may not be appended during retrieval operations.

13.1.2.3. Adding Records to a Direct IRAM File

In the sense of appending new records to the end of the string of data records in a direct IRAM file, adding records extends the file; this process is described in the preceding paragraph. In the quite different sense of inserting new data records into the existing data string, however, adding new records by relative record number is tantamount to updating without prior retrieval. However, an important difference between this way of adding records and the update procedures described in the following paragraph is that your direct file must *not* be defined as an update file.

To insert new records by relative record number without first retrieving the record at the specified file-relative position, you must deine your file as an output file. To add a new data record to an output file in this method, your program must predefine both a record work area (size equal to record length) and the field in which you supply to IRAM the relative record number of the position within the file to which the new record is to be written. You provide the relative record number in this field before you issue the output function.

Whether you use this method of inserting new data records into a direct IRAM file depends upon your methods for generating or obtaining relative record numbers and for dealing with synonyms. Some methods rely on your examining a record position before you add a new record; to add bindly in these methods would eventually destroy the integrity of your file.

13.1.2.4. Retrieving and Updating Records in a Direct IRAM File

There are three methods available for record retrieval or retrieval-with-update from a direct IRAM file. If you specify sequential processing mode, you may retrieve data records in the order in which they occur on disk. You may retrieve records randomly by supplying relative record numbers in your program, and you may retrieve records randomly by processing your direct IRAM file with a tag file that contains a set of relative record numbers produced by the ADDROUT option of the sort/merge program.

For consecutive retrieval, define the direct IRAM file as an input file for sequential processing mode; for retrieval-with-update, define it as an update file. (Only one data buffer may be specified for update mode.) When you issue the input function, consecutive retrieval begins automatically with the first record position in the data partition.

If you do not want to begin processing with the first consecutive record in the file, you may issue the appropriate function to set a different lower limit for retrieval. You must do this after the file is open, having predefined a field in which you provide the relative record number of the starting position desired for retrieval. Your first input function retrieves the record at this position, and the succeeding input functions retrieve data records in consecutive order. You may reset the lower limit at any time while the file is open.

To retrieve randomly by relative record number, define the direct IRAM file as an input file for random processing mode; for retrieval-with-update, define it as an update file. Your program must predefine the field in which you supply IRAM with the desired relative record number, which you do before issuing each input function. The output function for writing a retrieved, updated record back to its original disk location does not require a relative record number, and none should be supplied.

If your updating programs include provisions for tagging records for deletion, or overwriting them with a null pattern (blanks, zeros, etc.) so that they may be recognized as available for reuse, your retrieval programs should include a check for these conventions in order to avoid unnecessary processing of invalid records. IRAM itself has no means of avoiding the retrieval of logically deleted records.

If your file contains synonyms, your retrieval and updating programs must, obviously, contain the necessary coding to locate the desired record.

13.1.2.5. Deleting Records from a Direct IRAM File

Because IRAM does not provide a function for logically deleting a record from a file, your update programs should provide for whatever means are necessary, according to your own conventions. Although you may flag each record for later deletion when your updating program determines that it is eligible for removal from the file, consider also the alternative of overwriting a record that you no longer want to process with the null pattern that (according to your own programming conventions) is recognized as a relative record position available for reassignment to a new record. Depending upon your method of handling synonyms in your direct file, there may be synonym linking or chaining information in a record that should not be deleted when its data content is inactivated by your logical deletion procedure.

13.1.2.6. Reorganizing a Direct IRAM File

Unlike a sequential IRAM file, a direct file cannot be usefully reorganized with the sort/merge or data utility programs. It is not feasible, for example, to physically remove records flagged for deletion in copying a direct file with the data utility because of the inevitable change in the file-relative positions of the valid records carried over to the new file. They are carried over in the same *consecutive* order that they held in the old file; but, because the data utility has no means of generating new relative record numbers and substituting these in your synonym linkage fields, these fields are unchanged and are no longer valid.

Even though your file does not contain synonyms, if the valid records are to be written to relative record positions contained in or derived from a control field in each record, the data utility or sort/merge program nonetheless lacks the facility to write them to the desired locations in the new file. A method for compressing a direct file without synonyms would be to copy it consecutively, via the data utility, and to use the resulting sequential IRAM file as input to a direct IRAM file-creation program you have written in RPG II. This effectively recreates a direct IRAM file in a smaller disk space.

For details on the use of the data utility, refer to the data utilities user guide/programmer reference, UP-8069 (current version).

13.2. PROCESSING INDEXED IRAM FILES

Indexed IRAM files contain two separate partitions: a data partition with fixed-length records ordered consecutively in the order of submission and stored (exactly as in nonindexed IRAM files) in a single compact string of bytes; and, an index partition, containing blocked index entries at two or more hierarchic levels. Each data record in the data partition contains a key, a character string specified by the user, that uniquely identifies the record. All keys in the IRAM file are of the same length; a key may start at the head of the record it identifies, or it may be elsewhere within the record, but the location of all keys must be the same for all records in one file. For details on the structure of keyed records and the IRAM index, refer to 12.2.

If the index partition is activated during processing, data records may be referenced randomly or sequentially by the values of their keys. When the index partition is inactive, data records may be accessed consecutively (in the physical order in which they occur on disk) or randomly, according to their relative positions in the data partition. However, records may not be added unless the index is active. Activating the index is a specific detail of file definition, performed before the file is opened.

A multivolume indexed IRAM file may be created for processing with all volumes online, or with only one volume online at a time, and it must always be processed in the mode for which it was created. When a multivolume file is created for single-volume processing, random processing must always use the keyed retrieval functions, for which the index partition must be active; random retrieval by record number is not possible.

Both multivolume and single-volume indexed IRAM files may be created in an orderly or disorderly load. In an ordered load, records are submitted to IRAM in ascending order of key values; an out-of-sequence record is rejected, and an error is reported. In an unordered load, no checking of key sequence is performed by IRAM. In both types of load, however, IRAM checks for duplicate keys; a record whose key duplicates a key already in the file is rejected, and an error is reported.

A new data record with a key duplicating one already in the file may not be added to the file at any time. No sequence checking of keys may be specified during file extension operations. All IRAM files may be processed randomly with a tag file containing relative record numbers of the records selected for processing. Such a disk file may be prepared from the IRAM file by using the ADDROUT option of the independent sort/merge program; for details, see the sort/merge user guide, UP-8342 (current version). In addition, an indexed IRAM file may be processed sequentially with a file containing record key limits. Alternatively, the lower limit may be set within the RPG II program.

The following subsections discuss, in general terms, procedures for creating and extending an indexed IRAM file; for adding, retrieving, updating, and deleting records; and for reorganizing an indexed IRAM file. For detailed programming specifications, refer to the RPG II programmer reference, UP-8044 (current version).

13.2.1. Creating an Indexed IRAM File

You define your IRAM file as an indexed output file and present each data record to IRAM in a predefined record work area (of a size equal to one record length) before you issue the output function to write it to the data partition. All data records are of a uniform length, and each contains a unique record key. You must specify the record length, the key length, and the fixed location of the key in all records when you define the file.

Other specifications you must make in defining the file include:

- the size and address of your primary data buffer;
- the size and address of your index buffer; and
- the address of a field in your program that is to contain a search key.

For calculating the minimum data buffer size you may specify, follow the same procedure used for creating a sequential IRAM file — described in 13.1.1.1. The primary data buffer is half-word aligned and contiguous to the index buffer in main storage. You may optionally specify a secondary data buffer (except for update operations); this must be contiguous to the primary buffer and of exactly the same size. The secondary data buffer follows the primary buffer in main storage.

The index buffer in main storage is also half-word aligned and has a minimum length of 256 bytes; if larger, its size must be a multiple of 256 bytes. The RPG II compiler specifies this minimum length for you and provides you with means for increasing the size of the index buffer (by one or more (up to nine) 256-byte increments) to enhance the performance of your programs: not only the load program itself, but all programs subsequently accessing the indexed IRAM file when its index is active. You should do so if you can afford the main storage, bearing in mind that all subsequent programs that use keyed functions must specify the same index buffer size you used to create the file. (They need not use the same *data* buffer size, however; see 13.1.1.1.)

A good rule of thumb in determining your index buffer size is to multiply the sum of your specified key length plus three bytes by 20, rounding the result upward to the next multiple of 256 bytes. This figure is used in calculating disk space required for an indexed IRAM file (see 12.2.4).

The length of the field you must predefine in your file creation program to hold a search key is your specified key length plus three bytes. This field is required in any program that uses keyed functions. IRAM uses this field itself, and sometimes overlays it. Your programs should avoid this field except for placing a search key in it, just prior to requesting a random read.

In defining the file for creation, you may specify that IRAM is to check that the sequence of the keys in your submitted data records is in ascending order. If you do so, you must submit them in this order for the load (but not necessarily in subsequent file extensions); an attempt to load a data record containing an out-of-sequence or duplicate key is rejected immediately, and an error is reported.

If you do not request key sequence checking, you may submit your data records to IRAM in any key order; only duplicate keys will be rejected. Depending on the bias of your key distribution in this disordered load, you should expect the process to be less than half as fast as an orderly load with sequence checking.

13.2.2. Extending an Indexed IRAM File

Once the file is created, it may be extended in the same manner we have just described for creating the file. IRAM appends new records to the end of the data string. If you have specified sequence checking in creating the file, you are not constrained to extending the file with an orderly sequence of added record keys, but if you do specify sequence checking for extending the file, the key in each record submitted must be successively higher than any in the file or volume. Duplicate keys are rejected. You may also add new records while you are retrieving existing records from the file; see 13.2.4.

Once you have successfully added a new record to an indexed IRAM file, it is immediately available for retrieval. This is true because IRAM updates the index structure on a record-by-record basis during load, extend, and add operations. It does not sort the index following a disorderly load, but maintains the fine-level index in unbroken ascending key order at all times. Refer to 12.2 for details.

13.2.3. Retrieving and Updating in an IRAM File with Index Active

When a multivolume indexed file has been created for single-volume processing, random retrieval from each volume must always be performed with the index marked active, using the key of the desired record as a search argument. It is not possible to retrieve records from the mounted volume at random by relative record number.

Random retrieval from a single-volume indexed file is not limited in this way. You may retrieve records at random by key when the index is active, and at random by relative record number when it is not. The same is true for multivolume files created for multivolume processing. When the index is marked inactive, IRAM files may be processed only in retrieval and update modes, as described in 13.2.5. When the index is marked active, all retrieval from an IRAM file is done by key.

To retrieve randomly by key, define the file as an input file for random mode processing and specify that the index is active. Your program predefines a field in which you provide IRAM with the key of the desired record as a search argument before you issue the input function. The length of this field is three bytes greater than the key length specified for the file. No search key may contain a byte with the hexadecimal value 'FF'.

If you have specified update mode, random retrieval may be followed by an output function to update the record retrieved. IRAM prevents you from issuing the update function if the desired record has not been found. In updating a keyed record, you should take care not to alter its key in any way. IRAM does not check for alteration of the key nor does it prevent your update from being issued if the key of the updated record duplicates one already in the file. IRAM updates without reference to the index; therefore, if an updated record is written back to the file with a key that has been changed, subsequent retrieval by key is either impossible or unreliable.

Sequential retrieval by key requires that the file be defined as an input file for sequential processing and that the index be marked active. Before you issue the input function to retrieve the first record, you must establish the value of the record key at which the retrieval sequence is to begin. This is done by issuing a function to set the lower limit for retrieval and providing a key value in a predefined field of your program. IRAM will then search for a record containing a key equal to or greater than this value.

If the value you supply is zero, the record retrieved by your first input function is the record containing the lowest key in the file. (This is the first record in the data partition only if the file was created with an orderly load.) If the value you supply is greater than the highest key contained in the fine index, no retrieval sequence can begin. In this case, IRAM reports a "no-find".

Once a sequential-by-key retrieval sequence has been successfully started, it continues until:

- you reach end of file or end of volume;
- you specify a new lower limit to start a new sequential retrieval sequence;
- you reset file processing mode from sequential to random; or
- the file is closed (by your program or by IRAM, as in case of I/O error).

A sequential-by-key retrieval sequence is not terminated when you add a new record during retrieval operations, but is resumed with your next input function (see 13.2.4).

If your updating operations include provisions for flagging records (by your own conventions) that are to be deleted from the file, your retrieval programs should include coding to check for the presence of this delete flag, and to bypass or process each record accordingly. IRAM does not recognize your deletion code and will not avoid retrieval of a flagged record.

13.2.4. Adding Records during Retrieval — Index Active

Provided that you have marked the index active and have specified that you intend to add during retrieval, you may provide a new record to IRAM in a predefined work area in your program and request that it be appended to the file. IRAM refuses the action if the key of the new record duplicates a key already in the file, but the value of the key may be lower than or greater than any key in the file, or fall within the range of the existing keys. You may not specify that IRAM is to check key sequence when you add during retrieval.

If you append a new record in this manner during a sequential retrieve-by-key operation, the retrieval sequence is not terminated, but resume with your next issue of an input function. A record may not be added to an indexed IRAM file unless its index is active.

13.2.5. Retrieval and Update when Index Is Inactive

An indexed IRAM file may be processed only in retrieval or retrieval-and-update modes when the index is marked inactive. Update is not possible without prior retrieval. Programs accessing such files may not issue any keyed function, nor may they add new records. Attempts to extend the data partition are disallowed and result in an error report with status flags set to indicate an invalid macro issue. IRAM does not use an index buffer for the nonkeyed retrieval or update functions allowed for an indexed IRAM file processed with an inactive index, and therefore, your program does not require or define an index buffer.

When its index is inactive, random retrieval and retrieval-with-update of an indexed IRAM file may be performed by relative record number — but only if the file is a 1-volume file or was created for multivolume processing and all volumes are mounted. Define the file as an input file for random processing mode, with update if desired. The file may not be defined as an output file. Your program predefines the field where you provide the relative record number of the desired record before you issue the input function to retrieve it. An input request with a file-relative record number higher than the highest number recorded for the file results in transfer of control to your error routine, with status flags set to indicate an end-of-file condition. To terminate random retrieval, you close the file. For update, you may use a work area (length equal to one record length), or one data buffer to present the updated record to IRAM. Two buffers may not be specified for update processing.

The other method for retrieval or retrieval-with-update, from an indexed IRAM file with its index inactive, is consecutive processing. For this, define the file as an input file for sequential mode processing (with update if desired) and mark the index inactive.

If yours is a 1-volume file, or was created for multivolume processing and all volumes are to be mounted, you may set the lower file limit for consecutive retrieval. Your program predefines a 4-byte field in which you supply IRAM with the file-relative record number where consecutive processing is to begin; you move this number into the field before you issue the function to set the lower processing limit. Your first input function then retrieves the record at this file-relative address, and your successive input functions retrieve the remaining records in their consecutive, physical order on disk. If you do not set a lower limit, consecutive retrieval starts with the first record in the file. Retrieval terminates when IRAM detects end-of-file; it detects end-of-volume conditions automatically and issues mount messages to the operator for subsequent volumes.

If your file is a multivolume indexed file created for single-volume processing, you do not have the option of setting a *file*-relative lower limit for consecutive retrieval. If you are to provide a record number to IRAM intended to be a lower limit for retrieval, you must realize that IRAM treats this as a *volume*-relative record number — the first record in the volume data partition being record number 1. (This is the first record retrieved if you do not set a lower limit.) Consecutive retrieval terminates when end-of-volume is detected by IRAM.

13.2.6. Deleting Records from an Indexed IRAM File

Because IRAM does not provide a function for deleting records from your files, you must tend (yourself) to whatever is necessary for this purpose in your programming. A common method of logically deleting a record from a file that is being updated is to mark it with a deletion code: for example, a specific character in a specific data byte. Records so marked or flagged for deletion may later be physically removed from the file when it is reorganized — offline from IRAM processing. In the meantime, your retrieval programs should be coded so as to check for the presence of your conventional deletion flag in each record retrieved so that a logically deleted record may be recognized and bypassed. IRAM has no provisions for recognizing your deletion flag and avoiding the retrieval of records containing it.

In establishing your own convention for logical deletion, restrict your flagging to one or more data bytes, recalling the unpredictable results of changing the key of a record in an indexed IRAM file during update (12.2.3). The index of the IRAM file is not available to you for marking index entries that refer to records you intend to be logically deleted, and you should not attempt this.

13.2.7. Reorganizing an Indexed IRAM File

You may have occasion to reorganize an indexed IRAM file: for example, to compress it by physically removing data records tagged for deletion, or to resequence the data records for more efficient or convenient processing. If you have created or extended your file with the disorderly load option, but then find increasing need to scan it in key sequence, you would improve your sequential retrieval program's performance as a result of dumping the file and reloading it in an ascending key order.

To reorganize an IRAM file, you will generally need to use other processors: either the independent sort/merge program, or the data utility program. Either of these, for example, will accept your IRAM file as input and delete or omit records as specified in the process of sorting or copying and recreating the file. These procedures are documented in the current version of the sort/merge user guide, UP-8342 and the data utilities user guide/programmer reference, UP-8069.

When your multivolume indexed IRAM file has been created for single-volume processing, you may find a need to regroup data records onto different volumes for more convenient processing. You cannot perform this task with the data utility program, however, because the utility is not parameterized to allow you to select the volume onto which a given record is to be copied. For this sort of reorganization, therefore, you would need to prepare a specific RPG II program.

13.3. DEFINING AN OS/3 IRAM FILE (DTFIR)

The DTFIR declarative macro is used to define an IRAM file to data management. It establishes a 240-byte nonindexed file table or a 307+KEYSIZE-byte indexed file table.

Normally, you do not need to know what the format of the DTFIR macro is because the file definition statements you use in your program to define the file are effectively translated into a DTFIR macro.

If, however, you want to temporarily change your file definition at run time by using a DD job control statement, you must know what the format is. To help you in these cases, the DTFIR Macro format and a summary of the keyword parameters (Table 13—1) that indicates which parameters can be changed by the DD job control statement are provided. Examples of typical DTFIR macros follow Table 13—1 and detailed descriptions of the individual DTFIR keyword parameters are provided in 13.4.

Format:

LABEL	Δ OPERATION Δ	OPERAND
filename	DTFIR	[ACCESS= {EXC, EXCR, SRD, SRDO}]
		[ADD=YES]
		,BFSZ=n
		[,EOFA=symbol]
		[,ERRO=symbol]
		[,INDA=symbol]
		[,INDS=n]
		[,INDX=YES]
		,IOA1=symbol
		[,IOA2=symbol]
		[,IORG=(r)]
		[,KARG=symbol]
		[,KLEN=n]
		[,KLOC=n]
		[LOCK=NO]

LABEL	Δ OPERATION Δ	OPERAND
filename	DTFIR (cont)	[,MODE= {SEQ / RAND}]
		[,OPTN=YES]
		,RCSZ=n
		[,SKAD=symbol]
		[,SQCK=YES]
		[,TYPE= {INPUT / OUTPUT}]
		[,UPDT=YES]
		[,VRFY=YES]
		[,VMNT=ONE]
		[,WORK=YES]

Table 13–1. Summary of DTFIR Keyword Parameters (Part 1 of 2)

Keyword	Specification	Keyed Operations		Nonkeyed Operations		Restrictions	Remarks
		INPUT	OUTPUT	INPUT	OUTPUT		
ACCESS*	EXC	O	O	O	O		This DTF: read/update/add use Other jobs: no access
	EXCR	X	X	O	O		This DTF: read/update/add use Other jobs: read use
	SRD	X	X	O	O		This DTF: read use Other jobs: read/update/add use
	SRDO	O	O	O	O		This DTF: read use Other jobs: read use
ADD	YES	O	X	X	X	Used only with keyed operations	Indicates new records are to be added to a file
BFSZ*	n	R	R	R	R	Always required	Supplies data buffer size
EOFA	symbol	R	X	R	X	Required if MODE=SEQ	Address of end-of-file routine
ERRO	symbol	O	O	O	O		Address of error-handling routine
INDA	symbol	R	R	X	X	Used only with keyed operations	Address of main storage area to contain index
INDS**	n	R	R	X	X	Used only with keyed operations	Indicates size of index area
INDX	YES	R	R	X	X	Used only with keyed operations	Indicates keyed operations
IOA1	symbol	R	R	R	R	Always required	Address of primary buffer
IOA2	symbol	O	O	O	O	Not permitted when UPDT=YES	Address of secondary buffer

Table 13–1. Summary of DTFIR Keyword Parameters (Part 2 of 2)

Keyword	Specification	Keyed Operations		Nonkeyed Operations		Restrictions	Remarks
		INPUT	OUTPUT	INPUT	OUTPUT		
IORG	(r)=general register	O	X	O	O	Not permitted when WORK=YES	Indicates I/O buffer index register
KARG	symbol	R	R	X	X	Used only with keyed operations	Address of field containing key of desired record
KLEN**	n	R	R	X	X	Used only with keyed operations	Indicates key length
KLOC**	n	R	R	X	X	Used only with keyed operations	Indicates the byte number location of the key within a record
LOCK	NO	O	O	O	O		Indicates file lock
MODE	SEQ	S	S	S	S		Sequential file processing (default)
	RAND	S	S	S	S		Random file processing
OPTN	YES	O	O	O	O		Optional file for sequentially processed files.
RCSZ*	n	R	R	R	R	Always required	Indicates record size
SKAD	symbol	X	X	R	R	Required if MODE=RAND	Address of seek address field
SQCK	YES	X	O	X	X	Used only with keyed operations	Indicates that sequence of keys for ordered load should be verified
TYPE	INPUT	R	X	R	X		Indicates input file type (default)
	OUTPUT	X	R	X	R		Indicates output file type
UPDT	YES	O	X	O	X	Input only	Indicates update capability
VRFY	YES	O	O	O	O	Used for TYPE=INPUT and permitted only when ADD=YES or UPDT?YES	Read/check of output records to be performed
VMNT	ONE	O	O	O	O	Not permitted when MODE=RAND unless INDX=YES	Defines file to be processed with only one volume online at any time
WORK	YES	O	R	O	O	Also required for keyed operations when TYPE=INPUT and ADD=YES. Not permitted in conjunction with IORG	Indicates that the record processing is in a work area

LEGEND:

O = Optional
R = Required
S = Select one
X = Not used

*Parameter may be changed on DD job control statement.
**Parameter may be changed on DD job control statement for index mode only.

Example (IRAM Output File):

```
LABEL    ΔOPERATIONΔ    OPERAND                    Δ    COMMENTS
1        10             16                                                  72
*        DEFINE THE FORMAT FOR CREATING AN INDEXED OUTPUT FILE LABELED
*        FILEOUT
FILEOUT  DTFIR  BFSZ=512,                                                    X
                RCSZ=512,                                                    X
                ERRO=ERI,                                                    X
                INDA=PLACEI,                                                 X
                IOAI=SPOTI,                                                  X
                INDS=256,                                                    X
                INDX=YES,                                                    X
                KARG=AREAI,                                                  X
                KLEN=30,                                                     X
                KLOC=5,                                                      X
                MODE=RAND,                                                   X
                TYPE=OUTPUT
```

Example (IRAM Input File):

```
LABEL    ΔOPERATIONΔ    OPERAND                    Δ    COMMENTS
1        10             16                                                  72
*        DEFINE THE FORMAT FOR PROCESSING AN INDEXED INPUT FILE
*        LABELED FILEIN
FILEIN   DTFIR  ADD=YES,                                                     X
                BFSZ=512,                                                    X
                EOFA=ENDFIL,                                                 X
                ERRO=ERI,                                                    X
                INDA=PLACEI,                                                 X
                IOAI=SPOTI,                                                  X
                INDS=256,                                                    X
                INDX=YES,                                                    X
                KARG=AREAI,                                                  X
                KLEN=30,                                                     X
                KLOC=5,                                                      X
                MODE=RAND,                                                   X
                RCSZ=512
```

13.4. DTFIR KEYWORD PARAMETERS

13.4.1. Specifying File Accessing Options (ACCESS)

See 11.4.1 for a detailed explanation of each ACCESS keyword parameter.

Note that indexed files should not be shared in an environment that permits only one writer to a file but any number of readers. If a file is shared, readers may get unpredictable results; that is, DM24, DM39 error messages or no-find errors may result when attempting to read records that were previously accessible. Consequently, the ACCESS=EXCR or ACCESS=SRD specification should not be made for an indexed file in either the DTFIR declarative macroinstruction or the DD job control statement.

Records added by the writer (ACCESS=EXCR) to a nonindexed file, in a shared environment that permits one writer and any number of readers, are not available to the reader (ACCESS=SRD). Once the writer closes the job, any added records will be available to users who subsequently open the file.

13.4.2. Specifying the Addition of Records to IRAM Input File (ADD)

The ADD parameter indicates that records may be added to an input file during record retrieval. These additions may be made only to indexed files processed by key.

Keyword Parameter ADD:

ADD=YES
If specified, the DTFIR declarative macro must also contain the INDX=YES keyword parameter.

13.4.3. Specifying the Buffer Size for IRAM File (BFSZ)

The BFSZ parameter indicates the size of the data buffer in the IRAM file.

Keyword Parameter BFSZ:

BFSZ=n
Is always required. n represents the number of bytes in the data buffer. The size must be at least 256 bytes as well as a multiple of 256. To calculate minimum buffer size, see 13.1.1.1.

13.4.4. Specifying the End-of-File Handling Routine (EOFA)

When data management senses the end of data while processing an IRAM input file sequentially by key or consecutively, it looks for the symbolic address of the user's end-of-data routine and transfers control there.

Keyword Parameter EOFA:

EOFA=symbol
Specifies the symbolic address (name) of your required routine that handles the end-of-data condition for IRAM input files. This parameter is required if MODE=SEQ is included in the DTFIR declarative macro; however, it is optional for randomly processed input files.

13.4.5. Specifying Error Routines (ERRO)

When data management detects any error or exception in processing, it looks for the symbolic address of your error-handling routine.

Keyword Parameter ERRO:

ERRO=symbol
Specifies the symbolic address (name) of the user error-handling routine. When data management transfers control to the error routine, *filenameC* contains information on the reasons for the error. (See Tables B—1 and B—3.) If ERRO parameter is omitted, control returns to your program inline.

13.4.6. Naming Main Storage Location for Index Block Processing (INDA)

During keyed operations, IRAM processes index blocks in a main storage index area.

Keyword Parameter INDA:

INDA=symbol
Specifies the symbolic address of this index block processing area in main storage. The specified area must be half-word aligned and its length must be specified in the INDS parameter. The location of the index area in main storage must immediately precede the primary I/O buffer area (IOA1). This parameter is required for all keyed or indexed IRAM file processing.

13.4.7. Specifying the Index Area Length in Main Storage (INDS)

When data management processes indexed (keyed) IRAM files, it uses an index area in main storage. The length of this area must be defined.

Keyword Parameter INDS:

INDS=n
Indicates the number of bytes used in main storage for the index area named in the INDA parameter. The index area length must be at least 256 bytes and, in addition, a multiple of 256 bytes. Both INDS and INDA parameters are required specifications for IRAM indexed files.

13.4.8. Indicating Processing by Key (INDX)

Data management processes nonindexed and indexed IRAM files.

Keyword Parameter INDX:

INDEX=YES
Indicates that IRAM file processing is to be performed by key. This parameter is required for input and output indexed IRAM files. In addition, the parameters INDA, INDS, KARG, KLEN, and KLOC must be specified if INDX=YES is used.

13.4.9. Identifying the I/O Area (IOA1)

When data management processes nonindexed or indexed IRAM files, it always uses at least one required I/O area.

Keyword Parameter IOA1:

IOA1=symbol
Specifies the symbolic address of the I/O processing area. IOA1 must be half-word aligned and greater than or equal to 256 bytes, a multiple of 256, and consistent with the BFSZ specification. IOA1 must immediately follow the index buffer (INDA), if specified, and must immediately precede the secondary I/O buffer (IOA2), if specified.

13.4.10 Identifying an Additional I/O Area (IOA2)

An additional I/O area may be indicated optionally for double buffering.

Keyword Parameter IOA2:

IOA2=symbol
Specifies the symbolic address of secondary I/O area. Similar to IOA1, the IOA2 parameter must be half-word aligned, the same size as the required IOA1 parameter, and immediately follow the primary I/O buffer. IOA2 may not be specified if the UPDT=YES parameter is specified.

13.4.11. Pointing to Current I/O Area (IORG)

When you are not referencing records in work areas, you must indicate an I/O buffer index register number.

Keyword Parameter IORG:

IORG=(r)
Indicates the register number used to point to the current I/O area. Registers 2 through 12 are available. Either the IORG or WORK parameter may be specified, but not both. If both parameters are specified, the WORK parameter specification is used.

13.4.12. Naming a Place for Key Retrieval (KARG)

When using indexed (keyed) operations, the user must name and define a location in his program where keys are placed for retrieval of IRAM records.

Keyword Parameter KARG:

KARG=symbol
Provides the symbolic address of the field in the user's program where keys are placed. The length of KARG must be equal to the specification in the KLEN parameter plus 3. The KARG parameter is required for all keyed operations.

13.4.13. Specifying Key Lengths for IRAM Files (KLEN)

In processing indexed IRAM files, data management must have the length of keys in an IRAM file.

Keyword Parameter KLEN:

KLEN=n
Specifies the number of bytes in a key for an IRAM indexed file. All keys must be the same length; the minimum length is 3 bytes and the maximum, 80 bytes. This parameter is required for all keyed operations.

13.4.14. Specifying Number of Bytes Preceding Keys (KLOC)

Often keys do not appear at the beginning of a record; when they do not, the number of bytes offset must be indicated.

Keyword Parameter KLOC:

KLOC=n
Indicates the number of bytes preceding the key of an IRAM record. The key location must be the same within all records of the file. If the key begins in the first byte of the record, KLOC=0 should be specified. This parameter is required for all keyed operations. If the KLOC parameter is omitted, IRAM assumes a value of zero; i.e., the keys begin in the first byte of each record.

13.4.15. Suppressing a File Lock (LOCK)

For a detailed explanation of the LOCK keyword parameter, see 11.4.11.

13.4.16. Specifying Retrieval and Load Modes for Indexed and Nonindexed IRAM Files (MODE)

Data management can process IRAM files sequentially or randomly according to the MODE keyword parameter specification.

Keyword Parameter MODE:

MODE=SEQ
Specifies sequential retrieval operations for an indexed IRAM file and sequential retrieval or sequential load operations for nonindexed IRAM files. Sequential mode is assumed if no MODE parameter is specified.

MODE=RAN
Specifies random (direct) retrieval operations for an indexed file and random retrieval or random load operations for a nonindexed file.

13.4.17. Specifying Optional Files (OPTN)

Sometimes you will not need to use a file on every program execution. In this case, the file is considered optional. Only sequentially processed input or output IRAM files can be optional files.

Keyword Parameter OTPN:

OPTN=YES
Specifies that the sequential input or output file defined by the DTFIR macro is an optional file. When this parameter is specified for an input file not allocated to a device by the DVC job control statement, data management transfers control to your EOFA routine on the first issue of an input operation. When the OPTN parameter is specified for an output file, data management transfers control to your program inline and with no error.

13.4.18. Specifying Record Length (RCSZ)

This parameter is always required and specifies the length of each record in bytes.

Keyword Parameter RCSZ:

RCSZ=n

13.4.19. Locating Relative Disk Address for Processing IRAM File by Relative Record Numbers (SKAD)

When data management randomly processes files defined by the DTFIR macro, you must specify the SKAD parameter.

Keyword Parameter SKAD:

SKAD=symbol
Specifies the symbolic address of an area in your program into which you load the relative disk address for use in processing nonindexed files by relative record number. The form of a record address is a 4-byte value, and the first record is relative record 1.

13.4.20. Verifying Ascending Record Key Order during File Creation (SQCK)

Data management can verify that record keys are in ascending sequence during file creation. This check is possible only on keyed operations, i.e., indexed files.

Keyword Parameter SQCK:

SQCK=YES
Specifies that data management should verify ascending key sequence on file creation for indexed files.

13.4.21. Specifying the File Type (TYPE)

The DTFIR declarative macro describes input and output files. The TYPE parameter designates input or output file types.

Keyword Parameter TYPE:

TYPE=INPUT
Specifies a read-only DTFIR file.

If omitted, data management assumes the TYPE=INPUT specification. You may not issue an output function to this file unless:

- you specify either the UPDT=YES or ADD=YES parameter; or
- you close the file, reset the file processing direction, and reopen the file.

TYPE=OUTPUT
Specifies a write-only file.

You may not issue an input function to this file unless you close, reset, and reopen the file.

13.4.22. Updating Records (UPDT)

When you wish to update a nonindexed or indexed input file and you have also specified TYPE=INPUT, you specify the UPDT parameter. If this parameter is omitted, you cannot update a file.

Keyword Parameter UPDT:

UPDT=YES

13.4.23. Verifying Output Records (VRFY)

Data management can check parity of output records after they have been written to disk. If it detects bad parity, data management sets the parity check flag (byte 2, bit 2) in *filenameC* and transfers control to your error routine or to your program inline if you have no error routine. Specifying this parameter results in an increase in execution times for update and file addition operations.

Keyword Parameter VRFY:

VRFY=YES

13.4.24. Specifying File Processing with One Volume Online at a Time (VMNT)

IRAM files created with one volume online at a time must be processed in the same manner.

Keyword Parameter VMNT:

VMNT=ONE
Specifies that only one volume be processed online at any time. This parameter cannot be used if the MODE=RAND parameter is specified unless the INDX=YES parameter is also specified.

13.4.25. Specifying Input or Output Record Processing in a Work Area (WORK)

To specify that input or output records are to be processed in a work area rather than an I/O buffer area, you specify the WORK parameter.

Keyword Parameter WORK:

WORK=YES
May not be specified in the same DTFIR with the IOREG parameter. When you issue the input, output, or file addition operations you specify the address of the work area. The WORK parameter is required for indexed files when TYPE=OUTPUT or when TYPE=INPUT and ADD=YES.

13.4.26. Nonstandard Forms of the Keyword Parameters

OS/3 data management accepts certain variant spellings for the keyword parameters described in this section. These variations are:

DTFIR Spelling	OS/3 Standard Form
BFSZ	BLKSIZE/BKSZ
EOFA	EOFADDR
ERRO	ERROR
INDA	INDAREA
INDS	INDSIZE
INDX	INDEXED
IOA1	IOAREA1
IOA2	IOAREA2
IORG	IOREG
KARG	KEYARG
KLEN	KEYLEN
KLOC	KEYLOC
OPTN	OPTION
RCSZ	RECSIZE
SKAD	SEEKADDR
TYPE	TYPEFLE/TYPF
UPDT	UPDATE
VRFY	VERIFY
WORKA	WORKA

13.5. IRAM KEYWORD PARAMETERS — DD JOB CONTROL STATEMENT SUPPORT ONLY

The following keyword parameters can be specified *only* by using a DD job control statement.

13.5.1. Variable Sector Support for IRAM Files (VSEC)

In IRAM files, both the data and index partitions in the DTF specify a fixed sector size of 256 bytes. This is required for the sectorized devices (8415, 8416, and 8418 disk subsystems), which are formatted for a 256-byte sector. The selector channel devices (8411, 8414, 8424, 8425, 8430, and 8433 disk subsystems) have no such constraint. However, they are preformatted at OPEN time to accept the 256-byte sector size. This sectorization requires hardware overhead and thus decreases the effective capacity of the disk.

Variable sector support eliminates the problem. It allows you to create IRAM files with data partition sector sizes larger than 256 bytes on the selector channel devices. Since the hardware overhead remains constant, the use of the larger sector size increases the effective capacity of the disk.

To use variable sector support, you must specify it in a DD job control statement that you include in your job control stream when the file is created. The format of this DD job control statement is:

```
// DD VSEC= {n   }
         {YES}
```

where:

VSEC=n
Specifies the sector size (number of bytes) to be used in creating the file.

VSEC=YES
Specifies that the sector size is to be computed at OPEN time, based upon record size and buffer size. The computed sector size will be the largest multiple of record size that does not exceed the buffer size.

If you use a DD statement to specify a sector size for a file, the statement must be used in all subsequent job control streams that access the file, unless you specify the ACCEPT parameter in the LFD statement for the file. If you do, the DD statement that specifies variable sector support may be omitted.

The message DM17 INVALID BLOCK SIZE SPECIFICATION is displayed if you use the VSEC parameter incorrectly in a DD job control statement. This occurs if:

- a sector size other than 256 bytes is specified for a sectorized device;
- the VSEC parameter specifies a sector size that is different from the sector size used to create the file; or

- changing the sector size results in the buffer size being less than the minimum buffer size required. To compute the minimum buffer size, the following rules apply:

  — If the record size divides evenly into the sector size, the minimum buffer size is equal to the sector size.

  — If the record size is a multiple of the sector size, the minimum buffer size is equal to the record size.

  — If neither of the preceding rules apply, add the sector size to the record size, subtract 1, and then round up to the next multiple of the sector size to find the minimum buffer size.

13.5.2. File Recovery Support for IRAM Files (RECV)

When you perform operations such as adding or updating records to a file, the physical file structure constantly changes during execution of your program. If your program runs to completion and the file is successfully closed, the file limits information contained in the file labels is updated. If a system failure occurs, the file is not closed and the file limits information is not updated. The effect of a system failure on a nonindexed file is that the file reverts back to its original state before it was opened. For example, added records are lost. In the case of an indexed file, system failure may cause the file to be compromised; that is, it must be recreated.

File recovery support eliminates these problems. It allows you to create IRAM files that can be reopened after system failure. It does this by updating the file limits information and writing it on the disk each time an operation that affects the information is performed. If you have specified file recovery and there is a system failure, you can reopen your file because the file limits information was saved.

To use file recovery support, you must specify a DD job control statement that you include in your job control stream when the file is created. The format for this DD job control statement is:

```
// DD RECV=YES
```

This statement is only valid at file creation time. It should not be included in the job control stream for subsequent jobs that process the file after creation.

If a system failure occurs during file creation, you will have to start over because the file creation program must run to completion and the file must be successfully closed before you can use the file recovery facility. If file creation is successful, the file recovery facility is activated each time you open the file.

If you are processing an indexed file (created with file recovery) where the physical file structure is changing and the structure modification process is interrupted by an operator cancel, HPR, or a hardware I/O error, the file may be compromised. If the file is compromised and you attempt to reopen it, the error message DM66 FILE COMPROMISED is displayed to inform you that the file should not be used. If you want to open the file solely for reading its data contents in order to recreate the file, the file recovery support facility allows you to override the error condition. This is accomplished by including the appropriate DD job control statement in the job control stream when you reopen the file. The format for this DD job control statement is:

```
// DD RECV=FCE
```

This statement causes the file compromised error to be ignored. The message DM66 FILE COMPROMISED does not appear when the statement is present. The statement should not be used unless you receive the error message, and then *only* if you want to reopen the file to read its data contents in order to recreate the file.

13.5.3. Automatic Computation of Initial Allocation Percentages for IRAM Files (AUTO)

Normally, when an indexed IRAM file is created, 50% of the initial allocation is assigned to the data partition, and 1% is assigned to the index partition. The remainder is left as unassigned space to be used for logical extensions to the file. As the file is loaded, the data partition and the index partition are filled at different rates. This results in the two partitions extending on an alternating basis which, for large files, tends to rapidly use up the extent table entries. This causes the extent table to be exhausted and the error message DM45 — EXTENT TABLE EXHAUSTED to be displayed.

Automatic computation of initial allocation percentage helps minimize the problem. It causes the entire initial allocation to be assigned to the index and data partitions in a calculated ratio based upon the record and key sizes.

To use the automatic computation of initial allocation percentage, you must specify it in a DD job control statement that you include in your job control stream when the file is created. The format of this DD job control statement is:

```
// DD SIZE=AUTO
```

This statement is only valid at file creation time and has no effect at other times. When it is encountered in the file creation job control stream for an IRAM file, this statement causes n% of the initial allocation to be assigned to the data partition and (100—n)% to the index partition. The value n is calculated at file open time and is dependent upon the record and key sizes.

There are two major factors that determine the accuracy of the calculated ratio:

1. The manner in which the file is loaded: Space in the index partition is used more efficiently if the records are loaded in ascending key sequence rather than in unordered key sequence.

2. The size of the file: Due to possible roundoff in the computation, the result is more accurate for relatively large files than for smaller files. For example, a given set of record and key lengths may yield a ratio of 3 to 1. For a 100 cylinder file, the allocation would be 75 cylinders for the data partition, 25 cylinders for the index partition. For a 10 cylinder file, the allocation would be 8 cylinders for the data partition, 2 cylinders for the index partition.

13A. MIRAM Formats and File Conventions

13A.1. GENERAL

The multiple indexed random access method (MIRAM), a sixth disk access method in OS/3, is used for handling sequential, relative, and indexed files in programs that are written in the OS/3 1974 American National Standard COBOL language, and for sequential and relative (direct) files in programs that are written in FORTRAN IV language. MIRAM provides the same functions as those provided by OS/3 ISAM, ASAM, IRAM, SAM, and DAM disk access methods. A MIRAM file may reside on any of the disk subsystems used with OS/3, and it may occupy from one to eight disk packs, which must be of the same type.

The MIRAM processor can access only MIRAM characteristic files and IRAM characteristic files that it has created or IRAM files created by the IRAM processor. It cannot access disk files that have been created by the ISAM, ASAM, DAM, or SAM access methods, nor can MIRAM files be processed by these access methods. MIRAM files can be processed by using the sort/merge program, however, and by the data utilities program.

A MIRAM characteristic file is one that meets any of the following conditions:

- More than one key per record is permitted.
- The file contains variable-length records.
- Records may be logically deleted from the file (RCB is present).
- Duplicate record keys are permitted, or key changes are allowed on update.
- The length of a key in a record is one or two bytes.

An IRAM characteristic file differs from a MIRAM characteristic file in that it meets none of the conditions required for the latter; that is:

- Only one key per record is permitted.
- The file contains fixed-length records.
- Records cannot be logically deleted from the file (RCB not present).
- Duplicate record keys or key changes during update are not permitted.
- The minimum length of the record key must be three bytes.

Note that IRAM characteristic files can be accessed by all programs that access IRAM files.

The discussions that follow deal with MIRAM characteristic files. For information on IRAM characteristic files, refer to Section 12.

13A.1.1. MIRAM Concepts

MIRAM has a number of features and concepts that distinguish it from other disk access methods.

- The data record slots in MIRAM files, for either fixed- or variable-length records, are of uniform size and may span physical blocks, sectors, tracks, and cylinders as required. They may even extend from one volume to another (unless the file was created for processing only a single volume at a time).
- Data records are written on disk compactly as a continuous string of bytes.
- The string of data records can always be accessed sequentially (consecutively) or by relative record number. In addition, the data can be specified to be indexed by up to five keys; this causes MIRAM to build a suitable index structure for each key type in a partition separate from the data.
- An indexed MIRAM file can be accessed by the additional random-by-key or sequential-by-key modes using a given key of reference, which can be changed.
- Indexed MIRAM files, either multivolume or single-volume, may be created by means of an orderly load (records submitted in ascending key order) or a disorderly load (records submitted in no particular key order) and they may be extended by appending records in either manner. MIRAM does not sort the index at the completion of a disorderly load, but maintains the index current on a record-by-record basis.
- MIRAM files are always extended unless the INIT parameter is specified on the LFD job control statement of the device assignment set. The EXTEND parameter is not supported for MIRAM files.

- Duplicate keys are permitted.
- When a new record has been added to an indexed or nonindexed file, it is immediately available for retrieval.
- Multivolume MIRAM files may be created for processing with either one volume online at a time, or with all volumes online. They must be processed in the same manner as they were created.
- All programs that access a MIRAM file need not use the same data buffer size for input/output as was used to create the file. Those that access an indexed MIRAM file, however, must use the same index buffer size.
- MIRAM allows you to logically delete records in your files; that is, it allows you to mark records so that in subsequent processing they will be ignored.
- MIRAM's restrictions are:
  - The maximum key length is 80 bytes.
  - No byte of a record key may contain the hexadecimal value 'FF'.
  - The minimum size for the index buffer is 256 bytes.

13A.2. MIRAM FILE ORGANIZATION

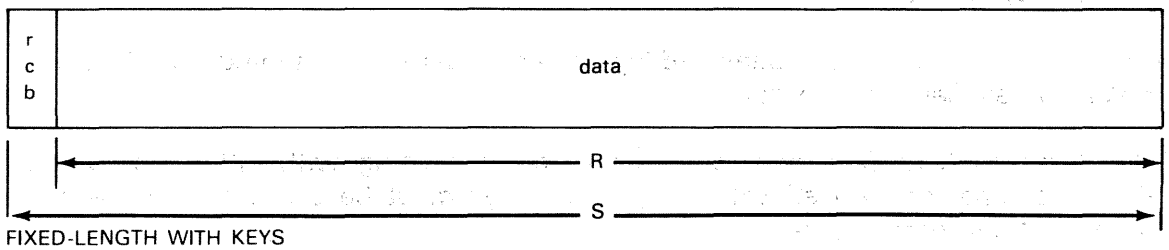
All MIRAM characteristic files contain two partitions: the data partition, which MIRAM defines to the system access technique (SAT) as containing 256-byte unkeyed physical blocks, and the index partition, which is defined as containing 256-byte keyed blocks. If the file is a nonindexed file, the index partition is not used; that is, no entries will be placed in it and no space will be allocated to it. If the file is an indexed file, entries will be placed in the index partition and space will be allocated to it.

For indexed files, the data partition precedes the index partitions, which begins on a separate cylinder.

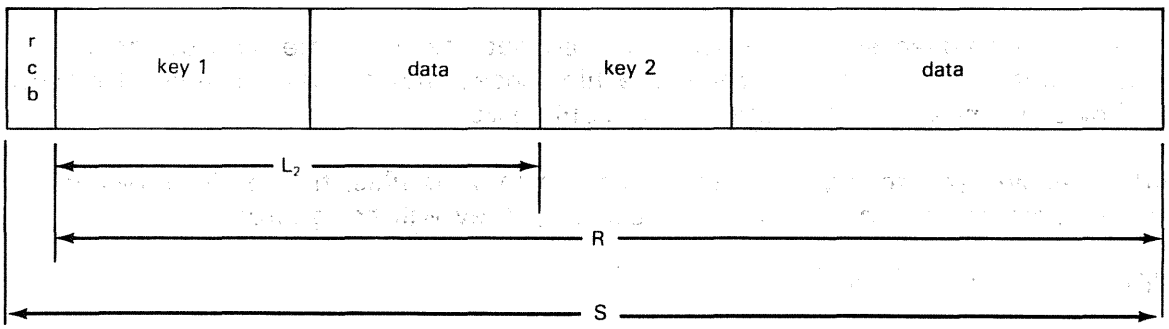
13A.2.1. The Data Partition

The data partition is arranged in the same way for both nonindexed and indexed files. It is cylinder-aligned and consists of a single compact string of data records that may be keyed or unkeyed. The formats of the MIRAM data records are shown in Figure 13A-1.

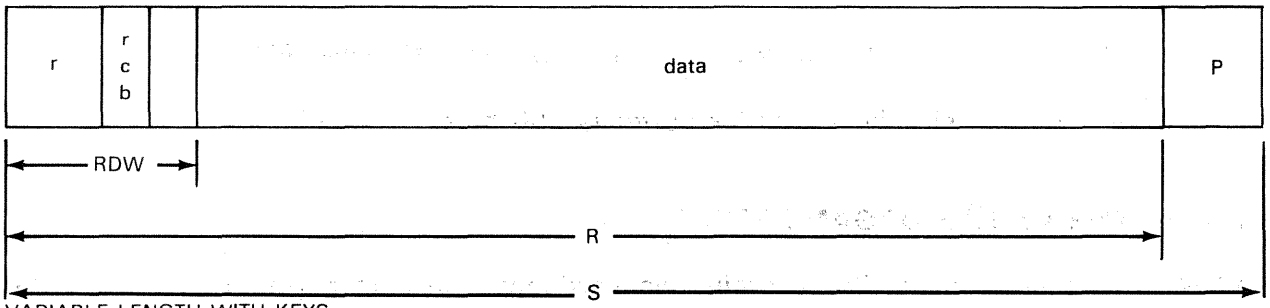
FIXED-LENGTH WITHOUT KEYS



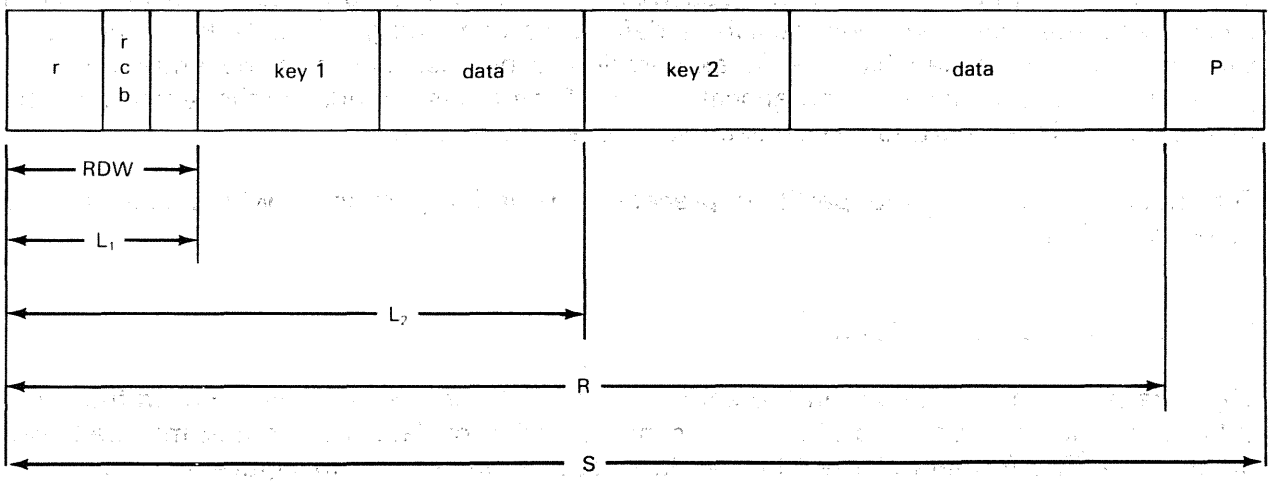
FIXED-LENGTH WITH KEYS



VARIABLE-LENGTH WITHOUT KEYS



VARIABLE-LENGTH WITH KEYS



LEGEND:

rcb = Record control byte (optional). Used to indicate that a record has been logically deleted from the file. For MIRAM fixed-length records, this byte is placed at the beginning of each record. For variable-length records, the third byte of the record descriptor word (RDW) is used as the rcb.

R = Length of the logical record (RDW plus keys plus data). You specify this length as the number of bytes. For variable-length records, this value, expressed in binary, must be placed in the first two bytes of the RDW.

Figure 13A—1. MIRAM Characteristic Data Record Formats (Part 1 of 2)

RDW = 4-byte record descriptor word for variable-length records. The first two bytes contain the logical record length (r) expressed in binary; the third byte may be used as the rcb; the fourth byte is not used.

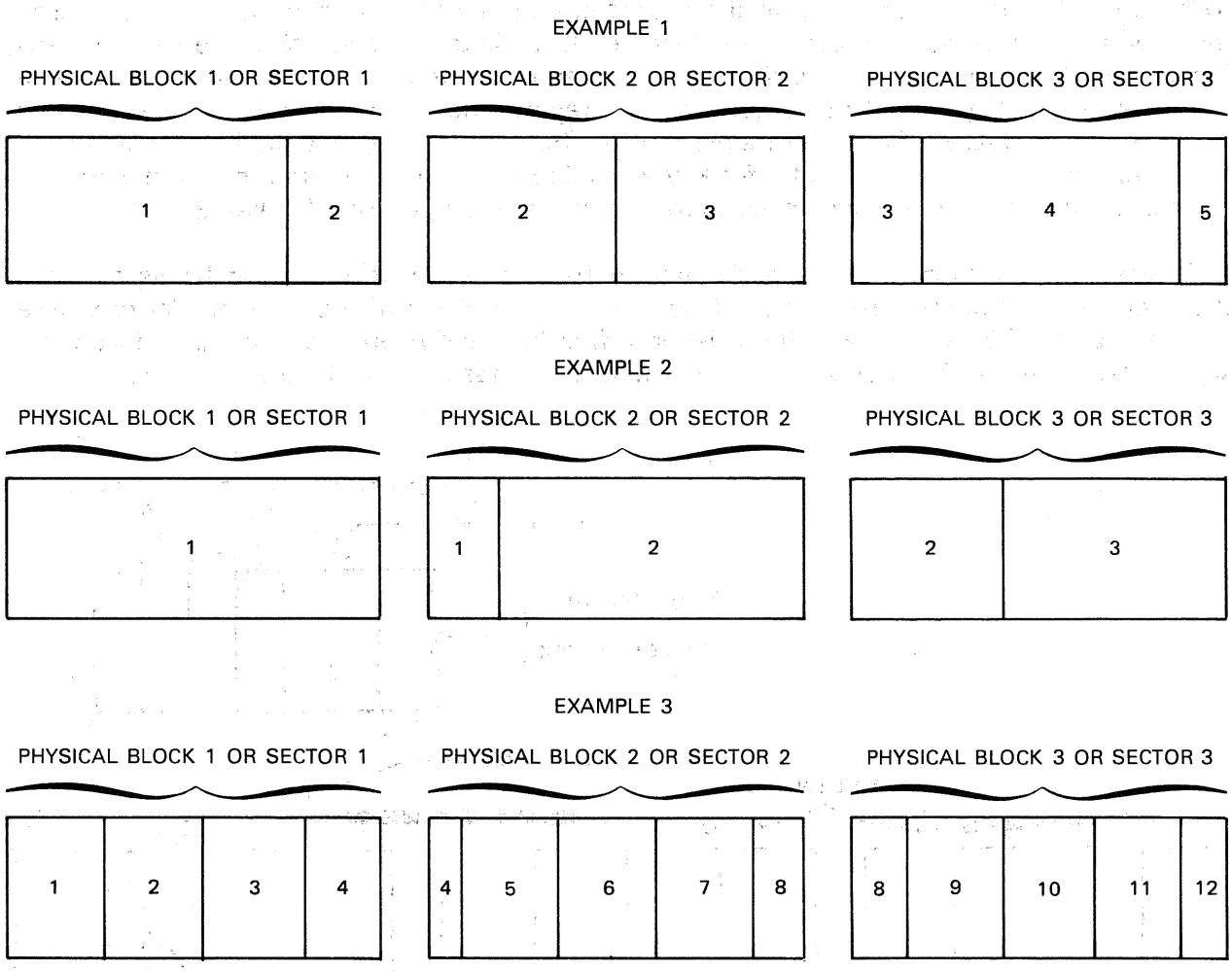
L_n = The starting location of record key n (n = 1 through 5) of a MIRAM file data record when the key does not start in the first byte of the record. L_n represents the number of bytes (RDW plus data) that precede key n. The starting location of key n must be the same in each record. Key n must have the same length in each record (a minimum of 1 byte and a maximum of 80), and no byte may contain the hexadecimal value 'FF'.

S = Slot size. All records are written into fixed-size slots. Slot size equals the record size + 1 for fixed-length records with a record control byte; otherwise, slot size equals the record size. Slot size for variable-length records equals maximum record size + 4-byte record descriptor word.

P = Padding.

Figure 13—1. MIRAM Characteristic Data Record Formats (Part 2 of 2)

When data records are stored in a MIRAM file, the records are placed in uniform size record slots and are arranged in the same order you originally presented them to the MIRAM processor. These data records are stored in 256-byte physical blocks or sectors on your disk packs. Since the record slot size does not have to conform to the physical block or sector size, the records may span these physical boundaries as shown in Figure 13A-2.



NOTES:

1. All physical blocks or sectors are 256 bytes.
2. 1, 2, 3 ... n represent record slots.
3. Record slots in Example 1 are approximately 190 bytes each.
4. Record slots in Example 2 are approximately 300 bytes each.
5. Record slots in Example 3 are approximately 70 bytes each.

Figure 13A—2. MIRAM Data Record Slots Spanning Physical Block or Sector Boundaries

Your data records may also span track boundaries, cylinder boundaries, and volume boundaries (except when a multivolume file is created for processing with only one volume online at any one time). When new records are added to a file, they are appended to the existing data record string; that is, they are added at the end as a continuation of the original string.

13A.2.2. Entries in the Index Partition

If you have keyed records, entries are placed in the index partition as these records are loaded into the data partition. MIRAM extracts all the keys from each record (a maximum of five keys is permitted) and constructs a 3-byte pointer for each of the keys from the file relative record number of the position the record was written to. From these it forms an index entry for each of the keys in the record and stores them in the index partition. The index entry for each key consists of the key plus three bytes (it is equal to the specified key length plus three bytes) and it is stored in an area of the index partition, which is called a fine-level index. If you had three keys in each record, the index entry for each key would be stored in a separate fine-level index; that is, the entry for key 1 would be stored in the fine-level index for key 1, the entry for key 2 would be stored in the fine-level index for key 2, and the entry for key 3 would be stored in the fine-level index for key 3.

A fine-level index is not formatted for hardware search, unlike the other levels of index that will be described subsequently. It is treated as a chain of multisector blocks where each sector is 256 bytes long. All entries in a fine-level index are maintained in ascending key order. Figure 13A—3 shows a typical fine-level index block of three sectors.

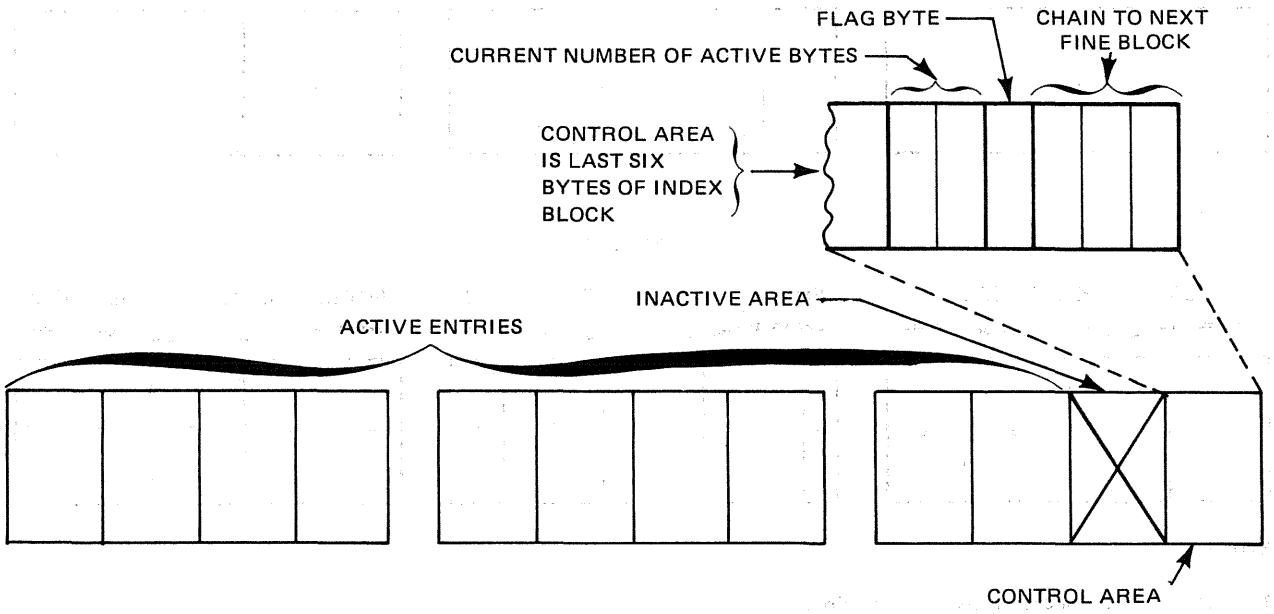


Figure 13A—3. Fine-Level Index Block

When a fine-level index is created, another hierarchical level of index is always created – the coarse-level index. This is hardware searchable and is composed of 256-byte blocks that contain entries similar to those in the fine-level index. They differ, however, in that the 3-byte pointer in each coarse-level entry does not represent the file-relative number of a record in the data partition. Instead, it points to another index block at a lower level – either a fine-level block or a block in what is called a mid-level index. Another difference is that instead of having a 6-byte control area, each coarse-level block uses its final byte to indicate the number of active entries. The entries in a coarse-level block are filed in descending key order. The high key of the block is the first one encountered by the hardware search and its length is equal to the longest key in the group plus four bytes. Both the coarse-level and mid-level blocks have the same format (Figure 13A–4).

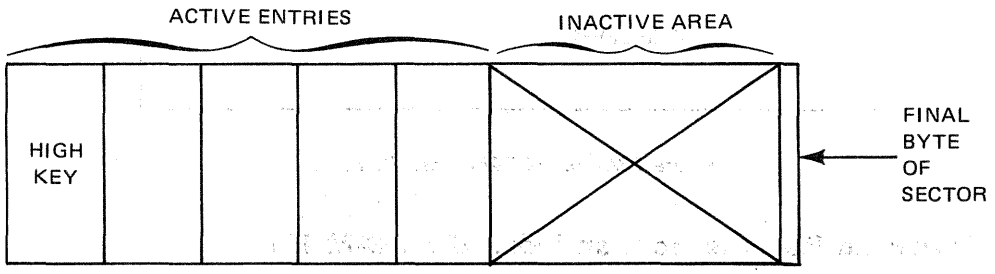


Figure 13A—4. Coarse- or Mid-Level Index Block

13A.2.3. MIRAM Index Structure

As you know, you can specify up to five keys for a file. For each key that you specify, the MIRAM processor will build a separate index structure. In those cases where you have more than one key, these separate index structures will allow you to use any of the key types as the key of reference to access your data records when you subsequently use the file in a program.

When the MIRAM processor builds an index structure for your file, it creates a minimum of two levels of index: a fine-level index and a coarse-level index. If your file is very large, one or more mid-level indexes are created as needed. The fine-level index consists of one entry for every record in the data partition of your file. The fine-level entries are filed in ascending key order until an index block (256 bytes) is filled. At this time, one coarse-level entry is made that points to the high key entry of that filled fine-level block. As each fine-level block is filled, another coarse-level entry is made. This process continues until all your records are on file.

The coarse-level index is automatically allocated by MIRAM. Its size is always one track regardless of the type of disks being used. If the coarse-level index is filled before all your records are on file, a mid-level index is created. The MIRAM processor allocates two tracks, designates them as a mid-level index, and copies the entries from the coarse-level track onto these tracks. It then places two entries in the coarse-level index. Each entry points to a high key in one of the tracks in the mid-level index. In this manner the entries on the coarse-level track are replaced by two entries. As new fine-level entries are recorded, one entry is made in the coarse-level index for each filled index block in fine-level index and when the coarse-level index is filled a new mid-level index is created just as before. This process continues until all records are on the file. Figure 13A–5 shows the structure of a MIRAM index.

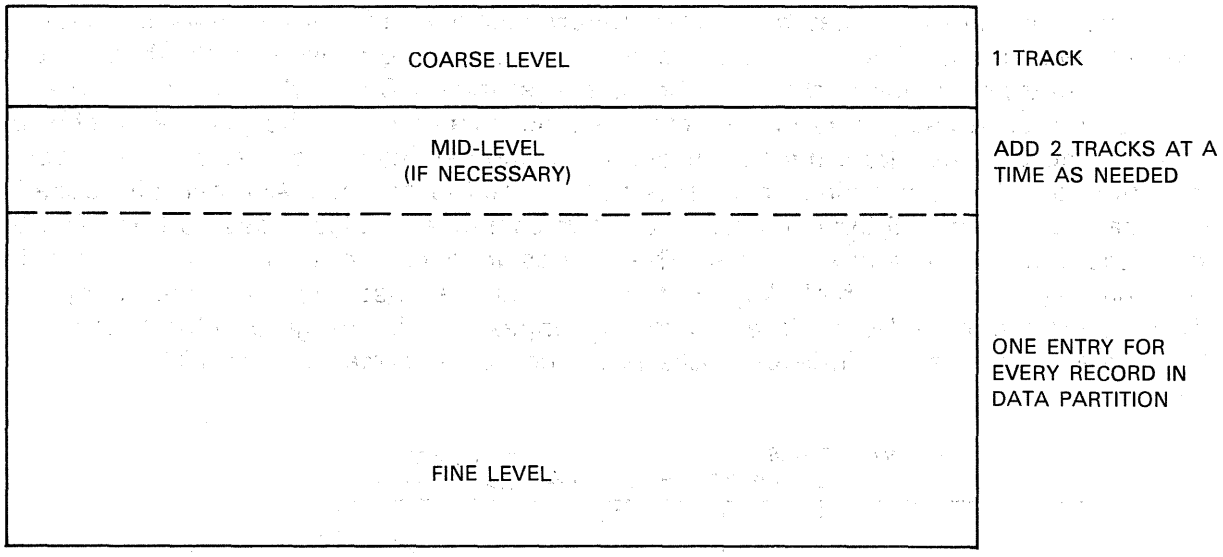


Figure 13A—5. MIRAM Index Partition

13A.2.4. Retrieving Records from an Indexed MIRAM File

To show how records are retrieved from an indexed MIRAM file, assume that a file with a 4-level index has been created. A search for a specific data record by key in this case would proceed as follows:

- the search begins in the coarse-level index;
- a hit is made that points to the first mid-level index;
- the new mid-level is searched;
- a hit is made that points to the second mid-level index;
- the second mid-level is searched;
- a hit is made that points to the fine-level index;
- the fine-level is searched;
- a hit is made that points to the data record in question; and
- the data record is retrieved.

13A.2.5. Estimating Disk Space Required for an Indexed MIRAM File

The following procedure will allow you to estimate the number of cylinders for your primary allocation of disk space to an indexed MIRAM file. The result is a good approximation that you can use in specifying the EXT statement in the job control device assignment set that allocates disk space for an indexed MIRAM file to be created by your program. This procedure can also be used to determine the number of cylinders to be allocated for an indexed MIRAM file that is to be generated from another file by the OS/3 data utility program.

The number of cylinders required for an indexed MIRAM file includes those occupied by the data partition and the index structures for each key type in the file. To estimate the number of cylinders the file will require, proceed as follows:

First, calculate D, the number of sectors required for your data records (the data partition).

Step 1:

$$D = \frac{\text{record-length} \cdot \text{number-of-records}}{S}$$

where:

S is the sector size. The default size is 256 for all types of disk subsystems. For 8411, 8414, 8424, 8425, 8430, and 8433 disk subsystems, this value can be greater than 256.

Next, calculate B_i, the number of index blocks required by your fine-level index for key_i.

Step 2:

$$B_i = \frac{\text{number-of-records} \cdot (\text{keylength}_i + 3)}{(256 \cdot m) - 6} \cdot \left(\frac{4}{3}\right)$$

where:

the factor of 4/3 is used because the average fine-level index will be 3/4 full.

m is the number of 256-byte sectors in the index buffer. (See 13B.3.1.)

Then calculate F_i, the number of 256-byte sectors required by your fine-level index for key_i.

Step 3:

$$F_i = m \cdot B_i$$

Repeat steps 1 through 3 as many times as necessary and then calculate F, the number of 256-byte sectors required by your fine-level indexes for all keys in the file.

Step 3a:

$$F = \sum_{i=1}^{n} F_i$$

where:

n is the number of keys in the file.

Then perform the final calculation. This calculation, which is the sum of the data requirements and the fine-level index requirements, represents over 95 percent of the space required for an indexed file. Once this is determined, it is a simple matter to figure out what your space requirements are for a given file.

Step 4:

$$C = \frac{F}{U \cdot N} + \frac{D}{A \cdot N}$$

40 7 40 7

where:

C
Is the number of cylinders to allocate to the indexed MIRAM file.

A
Is the disk dependent number of 256-byte sectors per track for data partition (Table 13A—1).

D
Is the number of 256-byte sectors required for the data partition.

F
Is the number of 256-byte sectors required by all the fine-level indexes in the file.

U
Is the disk-dependent number of 256-byte sectors per track for index partition (Table 13A—1).

N
Is the disk dependent number of tracks per cylinder (Table 13A—1).

Example:

Assume that you want to calculate the number of cylinders to allocate for an indexed MIRAM file and the following conditions apply:

Number of records	77,500
Record length	512 bytes
$Keylength_1$	28 bytes
$Keylength_2$	30 bytes
Sector size (data partition)	256 bytes
Index buffer length	512 bytes
Type of disk	8433

- $$D = \frac{\text{record length} \cdot \text{number-of-records}}{256}$$

  $$= \frac{512 \cdot 77{,}500}{256}$$

  = 155,000 sectors for data partition.

- $$B_1 = \frac{\text{number-of-records} \cdot (\text{keylength}_1 + 3)}{(256 \cdot m) - 6} \cdot \left(\frac{4}{3}\right)$$

  $$= \frac{77{,}500 \cdot (28 + 3)}{(256 \cdot 2) - 6} \cdot \left(\frac{4}{3}\right)$$

  = 6331 index blocks required for the fine-level index for key_1.

- $$F_1 = m \cdot B_1$$

  $$= 2 \cdot 6331$$

  = 12,662 sectors for the fine level index for key_1.

- $$B_2 = \frac{\text{number-of-records} \cdot (\text{keylength}_2 + 3)}{(256 \cdot m) - 6} \cdot \left(\frac{4}{3}\right)$$

  $$= \frac{77{,}500 \cdot (30 + 3)}{(256 \cdot 2) - 6} \cdot \left(\frac{4}{3}\right)$$

  = 6739 index blocks required for the fine-level index for key_2.

- $F_2 = m \cdot B_2$

  $= 2 \cdot 6739$

  $=$ 13,478 sectors for the fine-level index for key_2.

- $F = \sum_{i=1}^{n} F_i$

  $= F_1 + F_2$

  $= 12{,}662 + 13{,}478$

  $=$ 26,140 sectors for all the fine-level indexes for all keys in the file.

- The maximum coarse-level index on an 8433 disk can contain 132 sectors. As you can see, this number is too small to contain all of the index blocks for either of the keys.

- $C = \frac{F}{U \cdot N} + \frac{D}{A \cdot N}$

  $= \frac{(26{,}140 + 58 + 2090)}{29 \cdot 19} + \frac{155{,}000}{33 \cdot 19}$

  $=$ 299 cylinders to be allocated to the file.

NOTE:

After you have calculated your disk space requirements and you proceed to create your file, you must provide enough volumes to hold the file. This must be considered because the amount of space available is not the same for all types of disk. Refer to Table A—4 to determine how many volumes you will need based upon your calculations and the type of disk you intend to use.

13A.2.6. Estimating Disk Space Required for a Nonindexed MIRAM File

The following procedure will allow you to estimate the number of cylinders required for your primary allocation of disk space to a nonindexed MIRAM file. First, you must calculate D, the number of 256-byte sectors required for your data records (13A.2.5). Then, divide by the product of A times N (from Table 13A—1):

$$C = \frac{D}{A \cdot N}$$

13500

554

Table 13A—1. Disk-Dependent Factors for Determining Disk Space Requirements

SPERRY UNIVAC Disk Subsystem	U (Number of 256-byte sectors per disk track for index partition)	A (Number of 256-byte sectors per disk track for data partition	N (Number of tracks per disk cylinder)
8415	40	40	2* 3**
8416	40	40	7
8418	40	40	7
8411	10	11	10
8414	17	20	20
8424	17	20	20
8425	17	20	20
8430	29	33	19
8433	29	33	19

*Removable portion
**Fixed portion

8470

13B. Functions and Operations of MIRAM

13B.1. GENERAL

Before you create a MIRAM file (a MIRAM characteristic file unless noted otherwise) you must carefully consider how you will use the file in subsequent programs because this governs how it should be created. If you expect to process unkeyed records consecutively or you need to access specific records quickly without processing the preceding ones, the file should be created as a nonindexed file. If you intend to process keyed records either consecutively or randomly, the file should be created as an indexed file.

After you have created a MIRAM on file, you can perform retrieval, update, and other operations either consecutively or randomly, or you can switch back and forth between consecutive and random operations. Both nonindexed and indexed MIRAM files that span two or more volumes can be created with only one volume online (mounted) at a time, or with all volumes online. A multivolume file must always be processed in the same way it was created; only one volume is online at a time, or all volumes are online.

The discussions that follow are at a general level. For details of the actual programming statements you use for creating and processing MIRAM files in OS/3, refer to the OS/3 1974 American National Standard COBOL programmer reference, UP-8613 (current version), or the OS/3 FORTRAN IV supplementary reference, UP-8474 (current version).

13B.2. PROCESSING NONINDEXED MIRAM FILES

A nonindexed file is one that is organized consecutively. Its records are written on the disk in the physical order they are presented to the MIRAM processor. The records are processed consecutively in the order they appear on the disk. A nonindexed file can also be one that is organized relatively; each record in the file is written on the disk in a specific position relative to the beginning of the file (independent of the order in which they are presented to the MIRAM processor). This allows any record in the file to be retrieved directly without processing any preceding records when the location of the record is specified.

The following subsections describe the general procedures for creating and processing nonindexed files.

13B.2.1. Creating a Sequential MIRAM File

If you want to create a file for sequential processing, you must define the file as a sequential output file in your program. You must also specify the uniform size of your record slots and the size and address of your data buffers (I/O areas). If you have variable-length records, the slot size must be large enough to hold the maximum size record plus the required 4-byte record descriptor word (RDW). You may use two data buffers, each half-word aligned, but they must be contiguous and the same size. The minimum data buffer size that you can specify for your program is determined as follows:

- If the slot size divides into 256 without remainder, the minimum buffer size is 256 bytes.
- If the slot size is a multiple of 256, the minimum buffer size is equal to the slot size.
- If the slot size does not divide into 256 without remainder and is not a multiple of 256, add 255 to the slot size and round this sum upward to the next multiple of 256.

(Note that for fixed-length records, the slot size + 1 must be used in your buffer calculation. This is required because space must be provided in the buffer for the extra byte, the RCB, that is appended to the front of each fixed-length record by the MIRAM processor and is not counted when you specify your slot size.)

This calculation also applies when you create relative or indexed files. If you specify data buffers larger than the minimum, you may improve your program's performance. Note that when a file is processed in subsequent programs, you do not have to specify the same data buffer size that you used to create the file, but it must be at least the minimum.

After you open the file, you submit your records, one after the other, until you have no more records. The records are stored in the data partition in the order you submitted them. When the file is subsequently processed, it is processed in the same order. The MIRAM processor records the relative record number of the last record written in the file control table and the volume table of contents.

13B.2.2. Extending a Sequential MIRAM File

Once a sequential file has been created, it can be extended (enlarged) only by appending new records at the end of the existing data string. Records cannot be inserted between existing records nor can they be appended during retrieval or update operations.

Extending a sequential file is essentially the same process as creating the file because the same specifications are made. The difference is that you must define the file as a sequential file that is to be extended rather than a sequential output file. After you open the file you submit the new records, one after another, as in file creation, and they are stored in consecutive order starting after the last record of the existing data string.

13B.2.3. Adding Records to a Sequential MIRAM File

If you want to enlarge a sequential file by inserting records between existing records or appending records at the head of the existing data string, you must use some other processor. A way to solve this problem is to sort the new records into the consecutive sequence you expect to process them, and then use them as an input file (together with you sequential MIRAM file) to the sort/merge program or the data utilities program to create a new sequential file with the new records in the proper order. Refer to the sort/merge user guide, UP-8342 (current version) or the data utilities user guide/programmer reference, UP-8069 (current version) for details.

13B.2.4. Retrieving and Updating Records in a Sequential MIRAM File

You can retrieve records or retrieve and update records in sequential (consecutive) order in a sequential file. For sequential retrieval, you must define the file as a sequential input file in your program. For sequential retrieval and update, you must define the file as a sequential file for update. You can provide two data buffers. If you do, the lengths of these buffers need not be the same as the data buffer that was used to create the file. However, if two buffers are used, each one must be the same size as the other.

The MIRAM processor will provide the records in the same order that they were written on disk when the file was created or extended. Consecutive retrieval will continue until you close the file or the end of file is reached. If the file is to be terminated by end of file, you must have specified the address of a routine for handling this situation in your program. Records may not be appended during retrieval operations or retrieval and update operations.

13B.2.5. Deleting Records from a Sequential MIRAM File

If a MIRAM file is defined as a sequential file in your program, you cannot logically delete records from your file (mark specific records so they will be bypassed when the file is processed in subsequent programs). To delete records, the file must have been created as a relative file.

13B.2.6. Reorganizing a Sequential MIRAM File

At some point you may want to reorganize a sequential file. Perhaps your experience in using the file has shown that a different physical sequence of records would be more convenient. There are two methods that you can use to reorganize your file — the sort/merge program or the data utilities program. Either of these will accept your file as input and resequence the data records. For details, refer to the sort/merge user guide, UP—8342 (current version) or the data utilities user guide/programmer reference, UP—8069 (current version).

13B.2.7. Creating a Relative MIRAM File

If you require a file in which each data record is to be assigned to a specific position on disk, you must create a relative file. You do this by defining the file as a relative output file. You must also specify the uniform size of your record slots and the size of your data buffer. The minimum data buffer calculation is the same as that described for a sequential file in 13B.2.1.

After you open the file, you present each record, one by one, to the MIRAM processor in a work area you have defined along with the file relative position the record is to occupy. This position is not a disk address. It is a 4-byte file-relative number (relative to the beginning of the file; position 1 is relative record number 1, etc) that you supply in a field you have defined before you issue the output function to write the record to disk.

The method you use to determine the relative record number that you assign to each record is up to you. It can be a potential source of trouble if you are not careful; the method you use may generate a relative record number of a record that has already been placed on the file. If an output function is issued in this situation, the attempt to place the new record in an occupied position will be rejected and an error condition will result.

13B.2.8. Extending a Relative MIRAM File

Extending a relative file is essentially the same as creating it. You provide each record to be placed in the file in a work area, and you supply the relative record address for the record in a predefined field before you issue the output function. If you direct a record to a point beyond the current file end (established at file creation: that is, the relative record address of the last valid record on the file), any gap will be filled with void records. If you direct a record to a position short of the file end, the operation will be rejected if it is occupied by a valid record.

13B.2.9. Retrieving and Updating Records in a Relative MIRAM File

You can retrieve records or retrieve and update records in a relative file sequentially (consecutively) or randomly by relative record number. You also have the ability to switch retrieval methods (between sequential and random retrieval), and to specify at what point you want to commence sequential retrieval.

For sequential retrieval, define the file as an input file for sequential processing. For retrieval with update, define the file as an update file. When you issue an input function, retrieval begins automatically with the first record position in the file.

If you do not want to begin your sequential processing with the first record, you can issue a function that establishes the starting point. This must be done after the file has been opened and before you issue an input function. To establish a starting point, you must place a relative record number in a predefined field in your program and then use the starting point function to specify where the starting point is to be; that is, the starting point is equal to, greater than, or not less than the relative record number you have supplied. Your first input function retrieves the record at the starting point. The subsequent input functions will retrieve the records in consecutive order. You can change the starting point at any time while the file is open.

For random retrieval by relative record number, define the file as an input file for random retrieval. For retrieval with update, define the file as an update file. For random retrieval, you must supply the relative record number of the desired record in a predefined field in your program before you issue an input function. If you update a retrieved record, the output function to rewrite this record does not require you to supply a relative record number.

13B.2.10. Deleting Records From a Relative MIRAM File

If you want to logically delete records from a relative file, you must define the file as an update file. After the file is opened, you can delete a record by first issuing an input function for that record, and then following this with a delete function. The delete function logically deletes the record by marking it as a void record. (The record is not physically removed from the file.) The marking process consists of setting the high order bit in the RCB.

When a deleted record is encountered in subsequent sequential processing, it is bypassed. If you are retrieving records randomly and you specify the relative record number of a deleted record, it is treated as a no find.

13B.2.11. Reorganizing a Relative MIRAM File

If you want to reorganize a relative file, you cannot do it in a straightforward manner. For example, if you have logically deleted records in a relative file and you use the data utility program to remove the invalid records and recopy the file, the relative position of the valid records will change. This occurs because the valid records are copied in the same consecutive order that they appeared on the old file. As a result, any subsequent processing of the new file may prove unsatisfactory because the valid records will not be where they originally were.

A possible solution would be to use the data utility program to remove the invalid records and copy the valid records to a sequential file. This sequential file could then be used as input to a relative file creation program that will place the valid records in the positions you want them to be.

13B.3. PROCESSING INDEXED MIRAM FILES

An indexed MIRAM file contains a data partition with the data records ordered consecutively in the order that you submitted them, and an index partition that consists of one or more index structures arranged in ascending key order. The number of index structures is governed by the number of keys specified for the file. Each data record can contain from one to five keys (uniform length character strings that uniquely identify the record). A key may start at the head of the record or may be embedded within it; however, the location of each key type must be the same for all records in the file. Duplicate key values are permitted.

When you create an indexed file for processing one volume online at a time or with all volumes online, you can submit your records to the MIRAM processor as an orderly load or a disorderly load. The former means that the data records are submitted in ascending key order and the latter that the records are submitted in any other order. In both cases, the records are placed in the data partition in the order submitted; however, the keys are always arranged in ascending key order in the index structures. The following subsections describe the general procedures for creating and processing indexed files.

13B.3.1. Creating an Indexed MIRAM File

To create an indexed MIRAM file you must define the file as an indexed output file in your program. You must also specify the following:

- uniform size of your record slots;
- the size and address of your data buffers;
- the length and location of all keys in your records;
- the size and address of your index buffer; and
- the address of a field in your program that is to contain a search key.

If you have variable-length records, the slot size must be large enough to hold the maximum size record plus the required 4-byte record descriptor word (RDW). You may use two data buffers (each half-word aligned), but they must be contiguous, the same size, and they must immediately follow the index buffer. The minimum data buffer calculation is the same as that described for a sequential file in 13B.2.1.

The index buffer is also half-word aligned and must be at least 256 bytes in length. It can be larger; however, if it is, it must be a multiple of 256 bytes up to a maximum of 32,512 bytes. The index buffer must immediately precede the primary data buffer. A good rule to apply when determining your index buffer size is to multiply the sum of the largest specified key plus 3 bytes by 20, rounding the result up to the next multiple of 256.

The length of the field that is to contain a search key must be equal to the length of the largest key in your file plus three bytes.

After you open the file, you present each record to the MIRAM processor in a work area you have defined, and then you issue an output function to write the record to disk.

13B.3.2. Extending an Indexed MIRAM File

Extending an existing indexed file is essentially the same process as creating the file. As in creating the file, you present each record to the MIRAM processor in a work area you have defined and then you issue an output function to write the record to disk. The records are appended to the end of the data string in the data partition. Your records can be submitted as an orderly load or a disorderly load. In either case, the records are placed in the data partition in the order submitted and the record keys are placed in the index structures in ascending key order. After a record has been successfully added to the file, it is immediately available for retrieval because the index structures are updated each time a record is added to the file.

13B.3.3. Retrieving and Updating Records in an Indexed MIRAM File

You can retrieve records or retrieve and update records in an indexed file sequentially (consecutively) or randomly by key. You also have the ability to switch retrieval methods (between sequential and random) and to specify at what point you want to commence sequential retrieval.

For sequential retrieval, define the file as an input file for sequential processing. For sequential retrieval with update, define the file as an update file.

After the file is opened, the sequential position is established at the lowest key value in the key_i index structure. If you want a different starting point, you must establish a new starting point (the value of the record key at which retrieval is to begin). To do this, you must place a key value in a predefined field in your program and then use the starting point function to specify where the starting point is to be; the starting point is equal to, greater than, or not less than the key value you have supplied. Your first input function retrieves the record at the starting point and the subsequent input functions will retrieve the records in consecutive order.

After a sequential-by-key retrieval sequence has been started, you can continue it until:

- you reach the end of file or the end of volume;
- you specify a new starting point;
- you change the file processing mode from sequential to random; or
- you close your file or it is closed as the result of an error.

For random retrieval by key, define the file as an input file for random retrieval. For retrieval with update, define the file as an update file. For random retrieval you must supply the key of the desired record in a predefined field in your program before you issue an input function. If you update a retrieved record, the output function to rewrite this record does not require you to supply a key value.

If you are performing a sequential retrieval sequence and you interrupt it to perform a random retrieval operation, the sequential retrieval sequence cannot be resumed at the point it was interrupted (unless random retrieval with hold is requested).

If you have specified that duplicate keys are permitted and you are performing a sequential retrieval sequence, records with duplicate keys will be retrieved in the order they were presented to the file during file creation. If you perform a random retrieval, the first record encountered in a duplicate key series that contains a key equal to the search key will be retrieved.

13B.3.4. Adding Records to an Indexed MIRAM File during Retrieval

You can add records to an indexed file during retrieval if you have defined the file as an update file. To add a record during retrieval you must provide the new record in a predefined work area in your program, and then issue an output function to place it in the file. The record will be placed in the file if the value of the record key is less than or greater than any key in the file, or it falls within the range of the existing keys. If you have specified that duplicate keys are not permitted, the record will be rejected if its record key duplicates the key of a record that is already in the file.

If you interrupt a sequential retrieval sequence to add a record, the sequence will be resumed at the point it was interrupted when you issue the next input function.

13B.3.5. Deleting Records from an Indexed MIRAM File

If you want to logically delete records, you must define the file as an update file. After the file is opened, you can delete a record by first issuing an input function to retrieve the record and then follow this by a delete function. The delete function logically deletes the record by marking it as a void record. (The record is not physically removed from the file.) This marking consists of setting the high order bit in the RCB.

When a deleted record is encountered in subsequent sequential processing, it is bypassed. If you are retrieving records randomly and you specify the key of a deleted record, it is treated as a no find.

13B.3.6. Reorganizing an Indexed MIRAM File

Your experience in processing an indexed file may indicate that the file should be reorganized. For example, your processing may have logically deleted a large number of records and you want to compress the file by physically removing these records. Another reason to reorganize is that the file was created or extended with disorderly load, and you want to improve your program's performance because the majority of your processing involves sequential retrieval.

In the former case, you can use the data utility program to delete the invalid records. In the latter case, you can use the sort/merge program to sort your data records in ascending key order. For details refer to the data utilities user guide/programmer reference, UP-8069 (current version) or the sort/merge user guide, UP-8342 (current version).

13B.4. DEFINING AN OS/3 MIRAM FILE (DTFMI)

The DTFMI declarative macro is used to define a MIRAM file. It establishes a 388-byte file table.

Normally, you do not need to know what the format of the DTFMI macro is because the file definition statements you use in your program are effectively translated into a DTFMI macro.

If, however, you want to temporarily change your file definition at run time by using a DD job control statement, you must know what the format is. To help you in these cases, the DTFMI macro format and a summary of the keyword parameters (Table 13B—1) that indicates which parameters can be changed by the DD job control statement are provided. Examples of typical DTFMI macros follow Table 13B—1, and detailed descriptions of the individual DTFMI keyword parameters are provided in 13B.5.

Format:

LABEL	Δ OPERATION Δ	OPERAND
filename	DTFMI	[ACCESS= {EXC / EXCR / SRD / SRDO}]
		,BFSZ=n
		[,EOFA=symbol]
		[,ERRO=symbol]
		[,INDA=symbol]
		[,INDS=n]
		,IOA1=symbol
		[,IOA2=symbol]
		[,IORG=(r)]
		[,KARG=symbol]
		[,KEYn=(s,[I],[{NDUP / DUP}] [{NCHG / CHG}])]
		[,LOCK=NO]
		[,MODE= {SEQ / RAN / RANH}]
		[,OPTN=YES]
		[,PROC= {KEY / UNK}]
		[,RCB=NO]
		[,RCFM= {FIX / VAR}]
		,RCSZ=n
		[,RETR= {INF / MOD / REP}]
		,SKAD=symbol
		[,VMNT=ONE]
		[,VRFY=YES]
		[,WORK=YES]

Table 13B—1. Summary of DTFMI Keyword Parameters (Part 1 of 2)

Keyword	Specification	Keyed Operations	Nonkeyed Operations	Restrictions	Remarks
ACCESS*	EXC	S	S		This DTF: read/update/add use Other jobs: no access
	EXCR	X	S		This DTF: read/update/add use Other jobs: read use
	SRD	X	S		This DTF: read use Other jobs: read/update/add use
	SRDO	S	S		This DTF: read use Other jobs: read use
BFSZ*	n	R	R	Always required	Supplies data buffer size
EOFA	symbol	O	O		Address of end-of-file routine
ERRO	symbol	O	O		Address of error handling routine
INDA	symbol	R	X	Used only with keyed operations	Address of index buffer
INDS**	n	R	X	Used only with keyed operations	Indicates size of index buffer
IOA1	symbol	R	R	Always required	Address of primary data buffer
IOA2	symbol	O	O	Only allowed with sequential output or unkeyed sequential input	Address of secondary data buffer
IORG	(r)=general register	O	O	Not permitted when WORK=YES	Indicates I/O buffer index register
KARG	symbol	R	X	Used only with keyed operations	Address of field that contains key of desired record
KEYn**	n	R	X	Used only with keyed operations. n>1, key length <3, duplicate keys, or changes to keys not permitted for IRAM characteristic files.	Indicates key length, key location, and whether duplicate keys or changes to keys are allowed
LOCK	NO	O	O		Indicates file lock
MODE	SEQ	S	S		Sequential file processing (default)
	RAN	S	S		Random file processing
	RANH	S	S		Random file processing (hold sequential position)
OPTN	YES	O	O		Optional file
PROC	KEY	S	X		Keyed (index and data) operation (default)
	UNK	X	R		Nonkeyed (data) operation

Table 13B—1. Summary of DTFMI Keyword Parameters (Part 2 of 2)

Keyword	Specification	Keyed Operations	Nonkeyed Operations	Restrictions	Remarks
RCB	NO	O	O	Required for IRAM characteristic files	No record control byte
RCFM	FIX	S	S		Fixed-length records (default)
	VAR	S	S	Not permitted for IRAM characteristic files	Variable-length records
RCSZ*	n	R	R	Always required	Indicates record size
RETR	INF	S	S	Update not allowed	Retrieval for information (default)
	MOD	S	S		Retrieval for modification
	REP	S	S		Retrieval for replacement
SKAD	symbol	R	R	Always required	Address of seek address field
VRFY	YES	O	O		Check parity of output records after they have been written
VMNT	ONE	O	O	Nonkeyed random operations and random output operations not allowed.	Defines file to be processed with only one volume online at a time
WORK	YES	O	O	Required for all output, keyed update, and delete functions	Indicates that record processing is in a work area

LEGEND:

O = Optional
R = Required
S = Select one
X = Not used
*Parameter may be changed on DD job control statement.
**Parameter may be changed on DD job control statement for indexed file only.

Example (MIRAM Output File):

```
*        DEFINE THE FORMAT FOR CREATING AN
*        INDEXED MIRAM FILE LABELED INDXOUT

INDXOUT  DTFMI  BFSZ=512,                    X
                ERRO=ERRS,                   X
                INDA=TRAN,                   X
                IOA1=PBUF,                   X
                KARG=SKEY,                   X
                KEY1=(28,,DUP,CHG),          X
                KEY2=(30,40,DUP,CHG),        X
                INDS=512,                    X
                MODE=SEQ,                    X
                RETR=MOD,                    X
                RCSZ=512,                    X
                SKAD=LOOK,                   X
                WORK=YES
```

Example (MIRAM Input File):

```
*        DEFINE THE FORMAT FOR PROCESSING AN
*        INDEXED MIRAM FILE LABELED ININDX

ININDX   DTFMI  RCSZ=512,                    X
                BFSZ=512,                    X
                EOFA=ENDPRO,                 X
                ERRO=GOODFS,                 X
                INDA=PROI,                   X
                INDS=512,                    X
                IOA1=BUF1,                   X
                MODE=RAN,                    X
                KARG=SEARCH,                 X
                SKAD=RELOC,                  X
                WORK=YES,                    X
                KEY1=(10,NDUP,NCHG),         X
                KEY2=(6,391,DUP,CHG)
```

13B.5. DTFMI KEYWORD PARAMETERS

The following paragraphs discuss how each of the DTFMI keyword parameters are used.

13B.5.1. Specifying File Accessing Options (ACCESS)

See 11.4.1 for a detailed explanation to the ACCESS keyword parameter.

Note that indexed files should not be shared in an environment that permits only one writer to a file but any number of readers. If a file is shared, the readers may get unpredictable results; that is, DM24, DM39 error messages or no-find errors may result when attempting to read records that were previously accessible. Consequently, the ACCESS=EXCR or ACCESS=SRD specification should not be made for an indexed file in either the DTFMI declarative macroinstruction or the DD job control statement.

Records added by the writer (ACCESS=EXCR) to a nonindexed file, in a shared environment that permits one writer and any number of readers, are not available to the reader (ACCESS=SRD). Once the writer closes the job, any added records will be available to users who subsequently open the file.

13B.5.2. Specifying the Buffer Size for a MIRAM File (BFSZ)

The BFSZ parameter specifies the size of the data buffer in the file.

Keyword Parameter BFSZ:

BFSZ=n
Is always required. n represents the number of bytes in the data buffer. The size must be at least 256 bytes as well as a multiple of 256. To calculate the minimum buffer size, see 13B.2.1.

13B.5.3. Specifying the End-of-File Handling Routine (EOFA)

When data management senses the end of data while you are processing a MIRAM file sequentially by key or consecutively, it looks for the symbolic address of your end-of-file handling routine and transfers control to that address.

Keyword Parameter EOFA:

EOFA=symbol
Specifies the symbolic address of your end-of-file handling routine.

13B.5.4. Specifying Error Handling Routines (ERRO)

When data management detects any error or exception in processing, it looks for the symbolic address of your error handling routine.

Keyword Parameter ERRO:

ERRO=symbol
Specifies the symbolic address of your error handling routine to which data management transfers control when an error is detected. When control is transferred to this routine, filenameC contains information on the reasons for the error (see Tables B—1 and B—3). If this parameter is omitted, control is returned to your program inline.

13B.5.5. Naming the Main Storage Area for Index Block Processing (INDA)

During keyed operations, index blocks are processed in a main storage index area.

Keyword Parameter INDA:

INDA=symbol
Specifies the symbolic address of the main storage area in which index blocks are processed. This area must be half-word aligned and must immediately precede the primary I/O (data) buffer area IOA1. This parameter is required for all keyed operations and it requires that all related keyword parameters must be specified: INDS, KARG, and KEYn. If INDA is specified and any of the related parameters are omitted, it is assumed that indexed operations were not intended to be used, and none will be permitted.

13B.5.6. Specifying the Index Area Length in Main Storage (INDS)

When indexed files are processed, the index blocks are processed in the index area specified by INDA. The length of this area must be specified.

Keyword Parameter INDS:

INDS=n
Specifies the number of bytes to be used in main storage for the index area named in the INDA parameter. The length must be at least 256 bytes or a multiple of 256 up to a maximum of 32,512 bytes. This parameter is required for all indexed files.

13B.5.7. Identifying the Primary Data Buffer (IOA1)

When any file is processed, at least one data buffer is required and this area must be identified.

Keyword Parameter IOA1:

IOA1=symbol
Specifies the symbolic address of the primary data buffer. This area must be half-word aligned, greater than or equal to 256, a multiple of 256, and it must be consistent with the BFSZ parameter. It must immediately follow the index area (INDA) if specified, and must immediately precede the secondary data buffer (IOA2) if specified.

13B.5.8. Identifying the Secondary Data Buffer (IOA2)

If you want to use double buffering to improve the performance of your program you can specify an additional buffer in certain cases.

Keyword Parameter IOA2:

IOA2=symbol
Specifies the symbolic address of a secondary data buffer. As with IOA1, this buffer must be half-word aligned, the same size as IOA1, and must immediately follow the primary buffer (IOA1). You can use a secondary data buffer only when performing keyed or nonkeyed sequential output or nonkeyed sequential input operations.

13B.5.9. Pointing to the Current Data Buffer (IORG)

If you are not referencing records in work areas, you must specify a data buffer index register number.

Keyword Parameter IORG:

IORG=(r)
Specifies the number of the general register to be used to point to the current data buffer when you do not reference records in the work area. Registers 2 through 12 may be used. Either IORG or WORK must be specified. If both are specified, the WORK parameter is used.

13B.5.10. Naming the Key Argument Field (KARG)

If you are using keyed operations, you must name the location in your program where you will place the search key for record retrieval.

Keyword Parameter KARG:

KARG=symbol
Specifies the symbolic address of the field in you program where keys are placed to effect the retrieval of records. The length of this area must be equal to the largest key in your program plus 3 bytes. This parameter is required for all keyed operations.

13B.5.11. Specifying the Keys for an Indexed File (Keyn)

When you process an indexed file, you must specify (for each key in your records) the length, location, if duplicate keys are permitted, and whether the key may be changed during update.

Keyword Parameter KEYn:

KEYn=(s,[l],[{NDUP / DUP},[{NCHG / CHG}]])

Specifies one of up to five keys for an indexed file; that is, $1 \leqslant n \leqslant 5$. There must be a KEYn parameter for each key in the file. s specifies the size of the key and may range from 1 to 80 bytes. l specifies the number of bytes preceding the key. If l is omitted, 0 is assumed for fixed records and 4 for variable records. DUP specifies that duplicate keys are allowed. NDUP specifies that they are not allowed and is the default case. CHG specifies that the key can change during update. NCHG specifies that it cannot change and is the default case.

KEYn parameters are required unless you intend to accept the KEYn parameters that were specified when the file was created. If so, no KEYn specifications should be present.

For IRAM characteristic files, $n > 1$, $s < 3$, DUP, and CHG are not permitted.

13B.5.12. Suppressing a File Lock (LOCK)

See 11.4.11 for a detailed explanation of the LOCK keyword parameter.

13B.5.13. Specifying Processing Mode for MIRAM Files (MODE)

MIRAM files can be processed sequentially or randomly as specified by the MODE keyword parameter.

Keyword Parameter MODE:

MODE=SEQ

Specifies sequential file processing. This is the default case.

MODE=RAN

Specifies random file processing.

MODE=RANH

Specifies random file processing (hold sequential position).

13.B.5.14. Specifying Optional Files (OPTN)

If your program does not need to use a particular file each time you execute the program, the file is considered an optional file. Only sequentially processed files can be optional files.

Keyword Parameter OPTN:

OPTN=YES
Specifies that the sequentially processed file is an optional file. When this parameter is specified for a file not allocated to a device by a DVC job control statement, control is transferred to your EOFA routine when you issue an input operation. Control is transferred to your program inline with no error when you issue an output operation.

13B.5.15. Specifying Type of Operations (PROC)

When you process a file you must specify the type of operations you are going to perform (keyed operations or nonkeyed operations).

Keyword Parameter PROC:

PROC=KEY
Specifies keyed operations. This is the default case.

PROC=UNK
Specifies nonkeyed operations.

13B.5.16. Specifying Record Control Byte (RCB)

When a file is created, a record control byte (RCB) is appended to the beginning of each record unless you specify that you do not want this.

Keyword Parameter RCB:

RCB=NO
This specification only applies to files that are being created. It is required for IRAM characteristic files and it specifies that each record is not to contain a record control byte. If this parameter is specified, delete functions will not be permitted when the file is subsequently processed. The default case is that each record will contain an RCB. When the file creation program is completed and the file is closed, the format label is marked to indicate whether or not the RCB is present. Once the file is created, this format label indication cannot be changed; that is, if you subsequently process the file and attempt to use the RCB parameter, the format label indication will override it.

13B.5.17. Specifying Record Format (RCFM)

This parameter specifies the record format, either fixed-length or variable-length.

Keyword Parameter RCFM:

RCFM=FIX
Specifies fixed-length records. This is the default case.

RCFM=VAR
Specifies variable-length records. The first four bytes of a variable-length record is the record descriptor word (RDW). The first two bytes of the RDW contain the effective record size that you supply on output and data management supplies on input. The record size includes the 4-byte RDW. Variable-length records are contained within a fixed-size slot that is equal to the RCSZ specification.

13B.5.18. Specifying Record Length (RCSZ)

This parameter is always required. It specifies the length of each record in bytes.

Keyword Parameter RCSZ:

RCSZ=n
Specifies the length of each record in bytes. If you have variable records, this parameter should specify the maximum size plus the 4-byte record descriptor word (RDW) required for variable-length records. (See Figure 13A—1.)

13B.5.19. Specifying the Record Retrieval Purpose (RETR)

If you are going to retrieve records for other than information purposes, you must use this parameter.

Keyword Parameter RETR:

RETR=INF
Specifies that records are to be retrieved for information purposes only. This is the default case.

RETR=MOD
Specifies that records are to be retrieved for modification.

RETR=REP
Specifies that records are to be retrieved for replacement.

13B.5.20. Specifying the Location of the Relative Disk Address for Processing a MIRAM File by Relative Record Numbers (SKAD)

This parameter is always required. It specifies where the relative disk address is placed in your program for use in processing by relative record number.

Keyword Parameter SKAD:

SKAD=symbol
Specifies the symbolic address in your program where you place the relative disk address for use in processing files by relative record number. The form of the relative disk address is a 4-byte value. The first record is relative record 1.

13B.5.21. Verifying Output Records (VRFY)

You can specify that data management is to check the parity of output records after they have been written to disk.

Keyword Parameter VRFY:

VRFY=YES
Specifies that data management is to check the parity of output records after they have been written to disk. If it detects bad parity, data management sets the parity check flag (byte 2, bit 2) in filenameC and transfers control to your error routine, or to your program inline if you have no error routine. If you specify this parameter, it will result in an increase in execution times for output operations.

13B.5.22. Specifying File Processing with One Volume Online at a Time (VMNT)

If you want to process a file with one volume online at a time, you must specify this parameter.

Keyword Parameter VMNT:

VMNT=ONE
Specifies that the file is to be processed with only one volume online at any time. A file that is created in this manner must be processed in this manner. Nonkeyed random operations are not permitted. If a file was not created for processing with one volume online at a time, it cannot be processed in this manner.

13B.5.23. Specifying Record Processing in a Work Area (WORK)

You can use this parameter to specify that your records are to be processed in a work area rather than in a data buffer (I/O) area.

Keyword Parameter WORK:

WORK=YES
Specifies that input and output records will be processed in a work area rather than a data buffer area. The IORG parameter should not be specified when the WORK parameter is specified. If both are specified, the WORK parameter is used. When you issue input, update, output, or delete operations, you specify the address of the work area. The WORK parameter is required for all output, keyed update, and delete operations.

13B.5.24. Nonstandard Forms of the Keyword Parameters

OS/3 data management accepts certain variant spellings for the keyword parameters described in this section. These variations are:

DTFMI Spelling	OS/3 Standard Form
BFSZ	BLKSIZE/BKSZ
EOFA	EOFADDR
ERRO	ERROR
INDA	INDAREA
INDS	INDSIZE
IOA1	IOAREA1
IOA2	IOAREA2
IORG	IOREG
KARG	KEYARG
OPTN	OPTION
RCFM	RECFORM
RCSZ	RECSIZE
SKAD	SEEKADR
VRFY	VERIFY
WORK	WORKA

13B.6. MIRAM KEYWORD PARAMETERS — DD JOB CONTROL STATEMENT SUPPORT ONLY

The following keyword parameters can be specified *only* by using a DD job control statement.

13B.6.1. Variable Sector Support for MIRAM Files (VSEC)

The detailed explanation of this facility for IRAM files (13.5.1) also applies to MIRAM files.

13B.6.2. File Recovery Support for MIRAM Files (RECV)

The detailed explanation of this facility for IRAM files (13.5.2) also applies to MIRAM files.

13B.6.3. Automatic Computation of Initial Allocation Percentages for MIRAM Files (AUTO)

The detailed explanation of this facility for IRAM files (13.5.3) also applies to MIRAM files except that normally 0% of the initial allocation is assigned to the index partition for MIRAM files.

14. Nonindexed Disk File Formats and Conventions

14.1. GENERAL

This section describes the disk file formats and conventions that are part of the indexed file processor system of OS/3 data management. The system comprises three basic methods for processing or accessing disk files without indexes: the sequential access method (SAM); the direct access method (DAM); and a combination of these two, termed simply the "nonindexed method," which has no acronym.

The three methods of processing are explained in detail in Section 15, which describes:

- the logical input/output control system (IOCS), or processor, that OS/3 data management furnishes you for each method;
- the imperative macroinstructions that constitute the actual repertoire of file-processing functions available to you; and
- the *define-the-file* (DTF) declarative macros you will use to inform data management how your files are to be accessed and how you have structured them.

In both this section and Section 15, the term "DTF" is applied both to this type of declarative macro and to the file table that each DTF sets up for data management to use while you are processing your file.

You define your sequentially processed disk files to data management with the DTFSD declarative macro; consequently, these files are often called DTFSD files. Direct access, or randomly processed, files (these terms are synonymous in this manual) are defined by the DTFDA declarative macro and are termed DTFDA files. Files that you want to process by a combination of *both* sequential and random techniques you will define to data management with the DTFNI declarative macro. You may subdivide your DTFNI files into as many as seven file partitions; you define certain details of each partition with yet a fourth type of DTF declarative macro: the DPCA macro.

To summarize:

- Three access methods:

  Sequential access method (SAM)

  Direct access method (DAM)

  Nonindexed access method (a combination of SAM and DAM techniques)

- Four *define-the-file* (DTF) declarative macros:

  DTFSD — defines a sequentially processed, nonindexed disk file

  DTFDA — defines a randomly processed (direct access), nonindexed disk file

  DTFNI — defines a nonindexed disk file that is to be processed sequentially, randomly, or by a combination of both techniques

  DPCA — defines certain particulars of a partition of a DTFNI file

The various access methods and nonindexed disk file descriptions share certain concepts of file organization, file labelling, record formats, and record addressing; these are described generally in this section, although some details are further developed in Section 15. You will notice some differences from the OS/3 indexed sequential access method (ISAM), described in earlier sections of this manual; chief of these, of course, is the absence of any index structure. Another difference is that ISAM does not allow you to have your own header and trailer labels. A third point of difference lies in the concept of *keys* (14.3.3).

14.2. FILE ORGANIZATION

All DTFSD, DTFDA, and DTFNI files in the OS/3 nonindexed processor system may reside on one or more disk volumes and may be termed *multivolume files.* Multivolume DTFNI and DTFDA files must have *all* volumes of the file mounted and online for processing and may consist of no more than eight volumes. Multivolume DTFSD files are maintained in single-volume mode — only one of your disk packs being online at any time; there is no limit imposed by OS/3 on the total number of volumes you may have in a DTFSD file. Data management communicates with the operator for you, automatically, to request and validate mounting of the successive volumes of a multivolume DTFSD file.

All OS/3 nonindexed disk files are terminated by a logical end-of-file (EOF) pointer to the block one *after* the highest block in the file in which you have placed a data record. The address of this block is file- or partition-relative; that is, its numbering is related to the address of the block that begins the file or — if you are considering a partition of a file — begins this partition. The significance of the logical EOF pointer lies in the actions that occur when the address is accessed during sequential processing of an input file.

In DTFSD files (which have only *one* volume mounted online at a time), the EOF pointer also indicates the logical end-of-volume (EOV) for the online volume. When you have finished creating such a file and issue the CLOSE imperative macro to terminate processing, data management stores the address of the block next after the current block as the EOF/EOV pointer or end-of-data ID (EODID) in the DTF file table and on the disk format 2 label for the file. No special flag or notation is written in the data area at this relative disk address. When data management encounters the EOF address during input of your file, it transfers control to a routine you have prepared to handle the end-of-file condition. (This routine, called the EOFADDR routine from the keyword parameter by which you specify it in your DTF, is described in 15.6.4; see 15.7.2 for the CLOSE macro.) You should remember that, for multivolume DTFDA and DTFNI files (all volumes of which are mounted and online at all times when you are processing), there is no EOV in the DTFSD sense and no need for an EOFADDR routine except for input DTFNI files you process sequentially.

14.2.1. Partitioning DTFNI Files

You may subdivide each DTFNI file into as many as seven file partitions, each with its own partition name, record size, block size, and other characteristics. You define the overall file characteristics, name all the file partitions, and describe the first partition in the DTFNI declarative macro; you then use separate DPCA declarative macros for each partition to define the characteristics of the partition. Every DTF file table has a length of 242 bytes, but the table established by the DPCA declarative macro is only 82 bytes long; it is termed the *partition control appendage.* You access a file partition for processing by name, using the SETP macro (15.7.4).

14.2.2. Subfiles in DTFNI Partitions

You may further subdivide each partition of a DTFNI file into as many as 71 serial subfiles, which you must create in sequence. (They need not be processed sequentially, and you may access them for processing in any order, once you have created them.) Unlike a file partition, which may differ in certain characteristics from the basic DTFNI file, a subfile must not differ from the partition in any respect. Another difference between a subfile and a partition is that a subfile has no name; you address one by using its partition-relative serial number as an operand of the SETS macro by which you access your subfiles (the SETS macro is described in 15.7.5). To help it keep abreast of your processing of subfiles, data management maintains subfile tables; it reserves one track of the first volume of a file for these when you so instruct it by specifying the SUBFILE keyword parameter in your DTFNI or DPCA declarative macro (15.6.26). Figure 14—1 depicts the organization of a DTFNI file into partitions and subfiles.

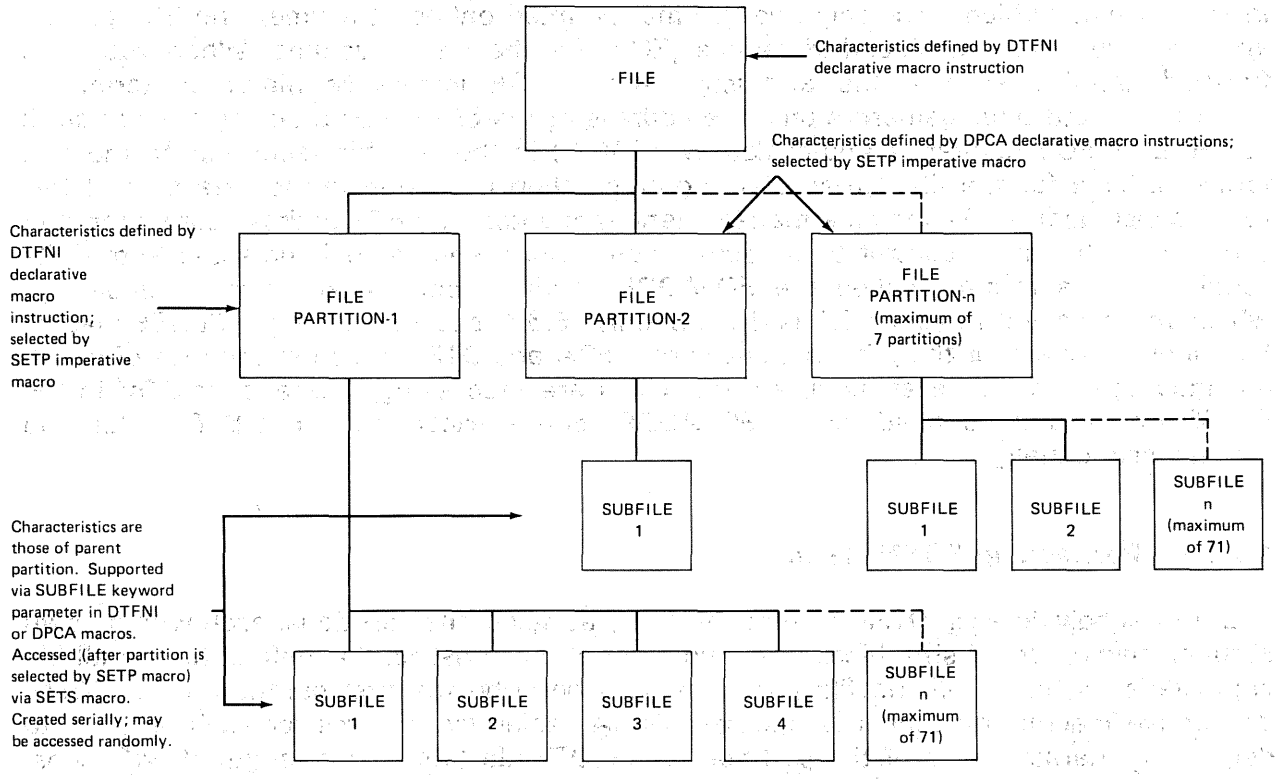


Figure 14—1. Organization of a DTFNI Disk File into Partitions and Subfiles

14.2.3. System Standard Labels for Nonindexed Disk Files

In processing your DTFSD, DTFDA, and DTFNI files, data management uses the OS/3 system standard labels for disk files. The system standard labels comprise the OS/3 standard volume label (VOL1) and seven types of disk format labels, designated format 0, format 1, and so on, located in a directory called the *volume table of contents* (VTOC). Data management accesses these standard labels to retrieve information it needs about your file and to add information on certain file characteristics to them. All are described in Appendix D.

For example, data management retrieves the VOL1 label to find the location of the VTOC. Here, it will read the first record in the VTOC (the format 4 label) to obtain the address of the last active format 1 label and will search the VTOC up to that address to locate the format 1 label for the current file. It needs this format 1 label, and any format 3 labels it may point to, for information about the extent space and secondary allocation for the file.

If your file is an output file, data management will rewrite the format 1 label to add to it some of the file characteristics you have defined in your DTF. If your file is currently an input file, data management retrieves and checks these characteristics. In addition, data management maintains a format 2 label to control file partitions and save information on existing and newly created files.

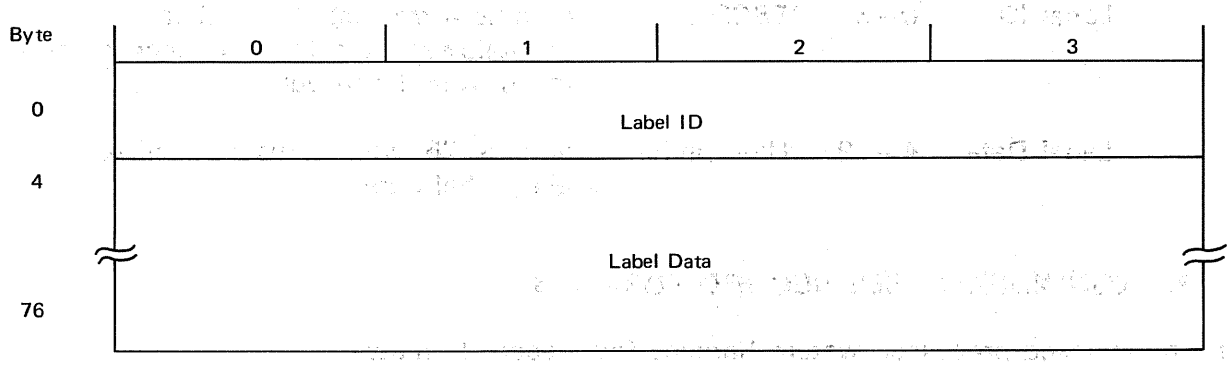
A look at Appendix D will make the foregoing clearer when you need to go into it. For the moment, however, it may be more important for you to know that these actions of data management are automatic and that you will rarely need to be concerned with the details. (Section 16 has further information on management of your disk resources.)

14.2.4. Optional Standard User Labels

Unlike OS/3 ISAM, which does not allow them, the nonindexed file processor system does support standard user header labels (UHL) and user trailer labels (UTL). These are optional, unblocked records, which you may process on the opening or closing of a volume with a routine you make available to data management by specifying its address in the LABADDR keyword parameter in your DTF (15.6.14).

14.2.4.1. User Header Labels

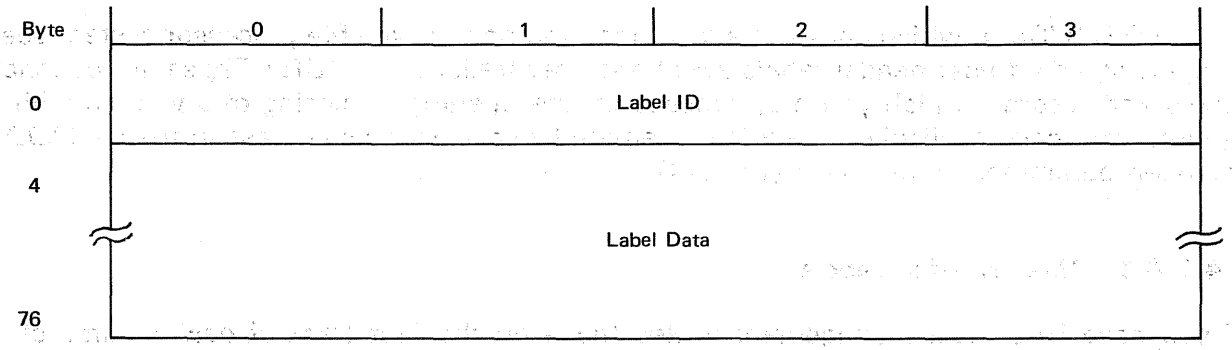
If you have UHL, data management writes these on the first track of *each* volume of a DTFSD file, and on the first track of the *first* volume of a DTFDA or DTFNI file. You may have no more than eight UHL, and they may not be blocked. This is their simple 80-byte format:



Field	Bytes	Code	Description
Label ID	0—3	EBCDIC	Contains 4-byte label identifier: UHL, followed by a label number which ranges from 1 through 8
Label Data	4—79	User option	Contains 76 bytes of user-specified header label data

14.2.4.2. User Trailer Labels

Data management writes your optional UTL on the first track of *each* volume of a DTFSD file and on the first track of the *first* volume DTFDA or DTFNI files; your UTL follow your UHL. You may have no more than eight, and they may not be blocked. This is their 80-byte form:



Field	Bytes	Code	Description
Label ID	0—3	EBCDIC	Contains 4-byte label identifier: UTL, followed by a label number which ranges from 1 through 8
Label Data	4—79	User option	Contains 76 bytes of user-specified trailer label data

14.3. NONINDEXED FILE RECORD FORMATS

The nonindexed processor system handles four record formats:

- Fixed-length, blocked
- Fixed-length, unblocked
- Variable-length, blocked
- Variable-length, unblocked

DTFSD and DTFNI files may comprise records in any of these four formats, but DTFDA files may not be specified as having blocked records (that is, physical blocks containing more than one logical data record, fixed- or variable-length).

It might be well at this point to review three OS/3 definitions:

block
The portion of a file transferred into or out of main storage by a single access.

buffer
An area in main storage for handling a block of data. Must not be smaller than the blocks to be handled.

record
The collection of contiguous characters, designated as such to data management by the user, for handling as a unit. Record size must not exceed block size.

The system does not handle undefined records; if you do not specify the record format in your DTF, data management assumes that your records are fixed and unblocked (that is, you have only one fixed-length logical record in each physical block). No record may occupy more than one block, nor may any record or block span from one disk track to another.

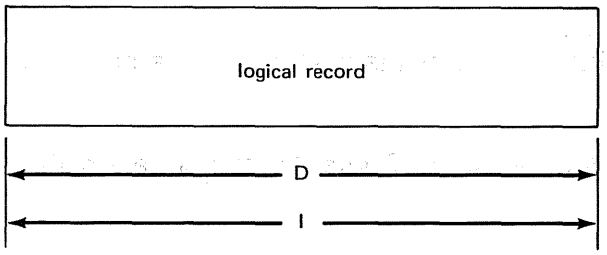
14.3.1. Fixed-Length Records

Fixed-length logical records are all of equal length for a given file or partition. When you specify that your fixed-length records are unblocked (or leave this specification to data management's default assumption, just mentioned), you do not need to specify the RECSIZE keyword parameter because data management takes the number of bytes per records as being what you specified with the BLKSIZE keyword. However, when you block your fixed-length records, you must specify both the BLKSIZE and RECSIZE keywords, and the value of BLKSIZE must be an exact multiple of RECSIZE.

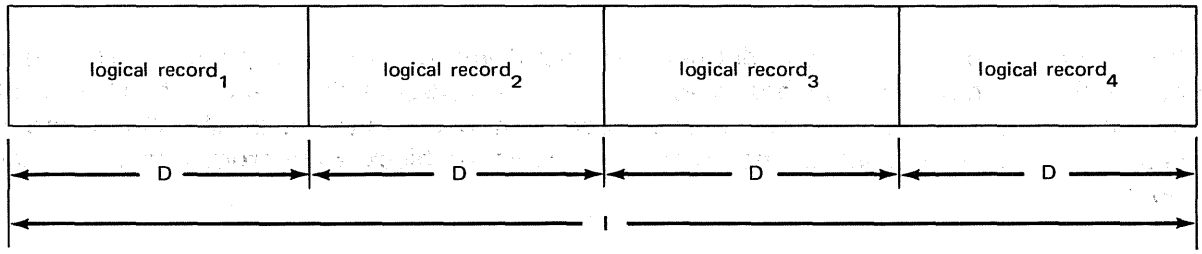
When you are using the variable-sector SPERRY UNIVAC 8411, 8414, 8424, 8425, 8430, or 8433 Disk Subsystems, your BLKSIZE specification governs exactly the number of bytes of data read or written. On the other hand, if you are using the fixed-sector SPERRY UNIVAC 8415, 8416, or 8418 Disk Subsystem and do not specify a blocksize that is a multiple of 256 bytes, the block written will always be larger than the specified block size; this is because the system writes or reads only in multiples of 256 bytes to the 8416 disk. Your BLKSIZE specification controls the number of bytes treated as *data* within a logical block, which is a multiple of 256. (For example, if you specify BLKSIZE=400 for a file that is to reside on an 8416 disk, data management will write out a block of 512 bytes.) You must be sure to reserve adequate buffer space for this circumstance, because a 512-byte block will be read into the buffer during retrieval.

Figure 14—2 illustrates the two fixed-length record formats and their relationship to the I/O area and DTF keyword parameters.

FIXED-LENGTH, UNBLOCKED RECORD



FIXED-LENGTH, BLOCKED RECORDS



LEGEND:

D Length of data in each logical record; you specify this length only when your records are blocked, using the RECSIZE keyword parameter in your DTF macro.

I Length of I/O area; you always specify this with the BLKSIZE keyword parameter of your DTF macro instruction. If your file is to reside on an 8416 disc, I must be a multiple of 256 bytes, and therefore the length of the block may exceed the nearest multiple of fixed-record length.

NOTE:

In preparing the illustration of blocked records, the arbitrary choice was made to show four logical records per physical block. The actual number you choose is a matter of file design. Remember that blocking is not specifiable for DTFDA files.

Figure 14—2. Fixed-Length Physical Record Formats, Nonindexed Disk Files without Keys

14.3.2. Variable-Length Records

When your records are variable-length, OS/3 data management preempts the first four bytes of every block for use as a *block descriptor word* (BDW). Data management calculates the effective length of the block and inserts this for you into the first two bytes of the BDW; it reserves the last two bytes of the BDW for its own use. You need not be concerned with the BDW, therefore, other than to allow for the fact that it reduces by four bytes the block space available for your logical records.

You are concerned, however, with the first four bytes of every logical record; these are also needed by data management for control and constitute the *record descriptor word* (RDW).

Data management, again, reserves the final two bytes of the RDW for its own use, but the first two bytes must contain the length of the record of which the RDW is a part. Before submitting a new record to be blocked, you must place the proper binary value in this 2-byte field.

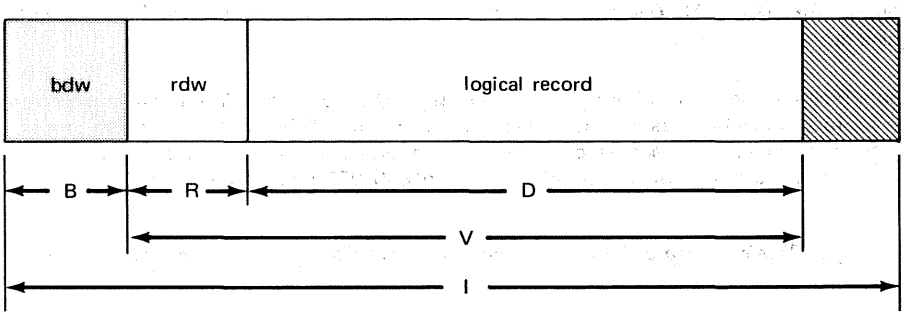
When you specify that your records are variable and unblocked, data management will write out one block for each logical record you submit, regardless of the amount of available space remaining in the I/O area. If you have specified blocked records, data management will pack as many complete records as possible into each block before writing it. In either case, the length of the block it writes will be the number of bytes you have specified with the BLKSIZE keyword parameter, unless more must be written because of roundup for the 8416 disk.

Unless your logical records follow some unusual pattern, there will always be varying amounts of unused space at block ends.

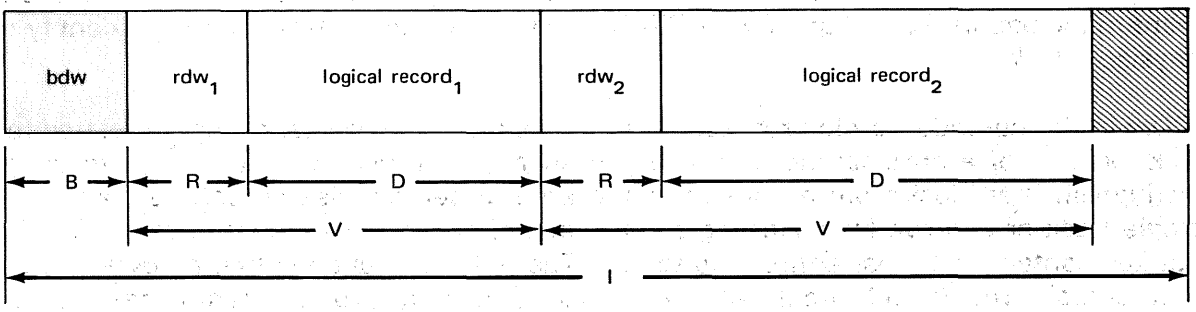
You must not use the RECSIZE keyword parameter in your DTF for a file containing variable-length records because data management expects to find the record size in the first two bytes of the RDW.

Figure 14—3 illustrates the layout of both formats for variable records and relates these to the length of the I/O area.

VARIABLE-LENGTH, UNBLOCKED RECORD



VARIABLE-LENGTH, BLOCKED RECORDS



LEGEND:

B Block descriptor word, four bytes. You must reserve space for this in your I/O area; it is also part of the block on disc. Data management calculates the block length in bytes and writes this in the first two bytes of the BDW; the last two bytes are reserved.

R Record descriptor word, always first four bytes of each variable-length record. You determine the length of each logical record in bytes, including this 4-byte RDW, and place it in the first two bytes of the RDW. The last two bytes are reserved.

Figure 14—3. Variable-Length Physical Record Formats, Nonindexed Disk Files without Keys (Part 1 of 2)

LEGEND (cont):

D Variable length of the data portion of your record.

V Length of the variable record (measured in bytes); includes four bytes for the RDW. You insert this number into the first two bytes of the RDW, in binary form.

I Length of the physical block, both on disc and in the I/O area. The I/O area length you specify via the BLKSIZE keyword parameter must accommodate the largest variable-length block in your file.

Unused space

Supplied by data management

NOTE:

In preparing this illustration of blocked records, an arbitrary choice was made to show two logical records per physical block. The actual number you choose is a matter of file design. Remember that you may not specify blocked record format for DTFDA files. This does *not* mean that OS/3 DAM does not handle the block format in which the unblocked variable-length record is shown here and actually exists on disc. It does. The point is that DAM does not block or deblock records for you. If what you can only describe to DAM via the DTFDA macro as an unblocked record is actually a block of logical records, you must provide for blocking and deblocking them yourself.

You will probably find the DTFNI file the better alternative; for one thing, the PUT and GET imperative macros may be issued to a DTFNI file for record level access; but they cannot be issued to a DTFDA file. In random processing of a DTFNI input file containing fixed, blocked records, data management provides you with the displacement of the desired record in the block, loading this into *filenameD* after the READ/WAITF macro sequence.

Figure 14—3. Variable-Length Physical Record Formats, Nonindexed Disk Files without Keys (Part 2 of 2)

14.3.3. Optional Key Fields with Nonindexed Files

Up to this point, our discussion of record formats has ignored an optional feature you may want to include in the design of your DTFDA and DTFNI files: a leading *key* to identify each physical block.

A key in the nonindexed file processor system is simply a character string, unrelated to the disk location of a physical block, which you specify and write with the block to uniquely distinguish that block from all others in the area of search. Search can be confined to a single track or can run from starting point to end of cylinder. Key length has certain limits, but key content is almost entirely up to you. The only restriction is that no byte of any key may contain the hexadecimal value FF. This value may produce erratic results during keyed retrieval. Otherwise, keys to your files may be constructed according to any scheme meaningful in the context of your file-processing needs, although uniqueness within the search area is assumed to be necessary because you will search for one specific block in the area, seeking to identify it by its key alone. Keys may be used with DTFDA and DTFNI files, and partitions of DTFNI files; they are not used with DTFSD files.*

**OS/3 data management does not provide you with a means for using keys in processing your DTFSD files because, in sequentially constructed files like these, each block is already uniquely identified by its relative location. OS/3 assumes that your reasons for organizing the file for sequential access only do not include a need to identify a block by some other means.*

If you are familiar with OS/3 ISAM, you might note at this point that, in the nonindexed file processor system, you associate a key only with a physical *block,* whereas in ISAM each of your logical *records* may have a key.

Sequential processing does not necessarily preclude the presence of keys. A DTFNI file or partition, for example, which you may always process sequentially, may have a key associated with each block of data. If it does, moreover, you must take the key into account when you issue the sequential access PUT and GET imperative macros to process the file. (This point is developed further in 15.7.9.5.)

A point to note here, perhaps, is that processing a file with the OS/3 sort/merge program is quite different from sequential processing in the data management sense. With sort/merge, you are concerned only with *sequential* organization of files and use the term *key* in a very different sense: a record may contain a number of fields, located anywhere within it, which sort/merge uses to resequence the file. Each of these fields is considered a sorting *key;* these may overlap and need not be unique. In the data management nonindexed system, however, keys are used only in direct access files and only with blocks; each block has only *one* key, always unique, and always located in front of the block.

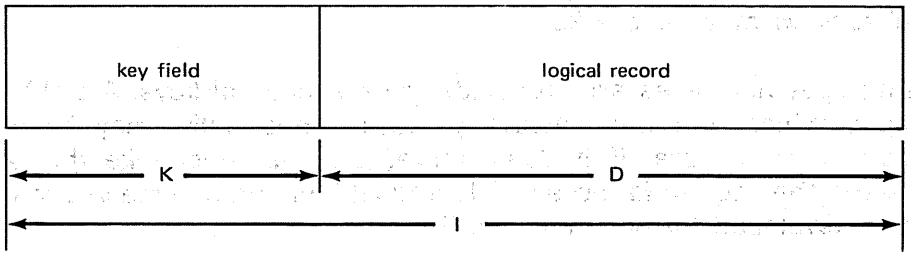
You have noted that the length of the key has certain limits in the nonindexed processor system: the minimum key length is 3 bytes, and the maximum is 255 bytes. Another point to remember is that all keys in any one file must have the same length; the only exception to this rule is the *partitioned* DTFNI file, in which each partition may have its own unique key length, uniform throughout the partition. You may not include blocks without keys in a file of keyed blocks.

When you are processing blocks with keys, you inform data management of the actual key length it will find or place with your blocks on disk by specifying the KEYLEN keyword parameter in the DTFDA, DTFNI, or DPCA declarative macro.

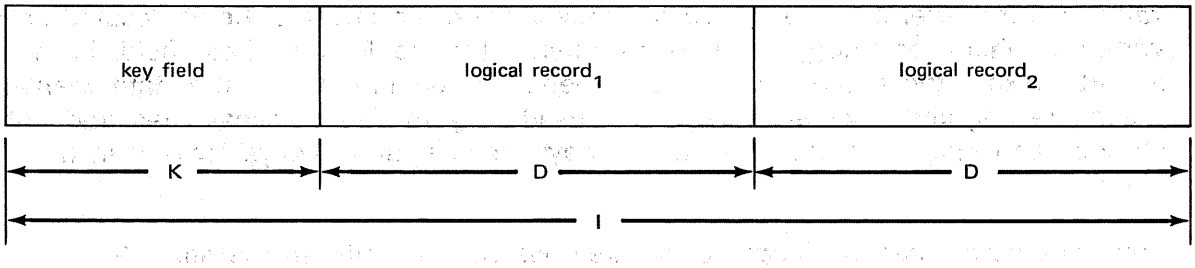
The various forms of the READ and WRITE imperative macros are designed to give you a useful set of options for retrieving records by key and writing or updating the key and data portions; these are described in Section 15.

The effects of keys on the physical block length and, consequently, on the layout of your blocks on disk and in the I/O buffer are illustrated in Figure 14—4 for both fixed and variable records.

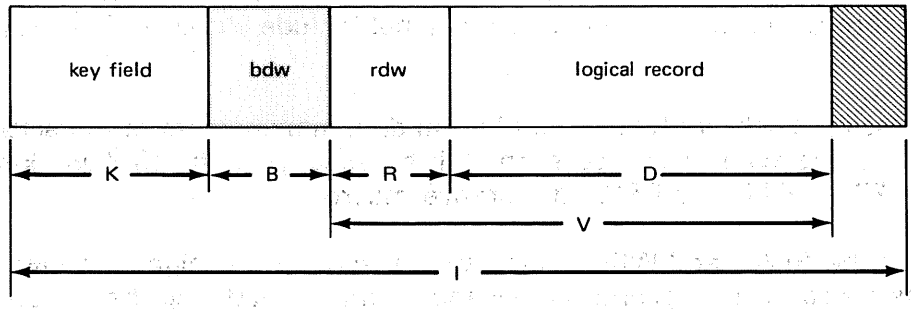
FIXED-LENGTH, UNBLOCKED RECORD



FIXED-LENGTH, BLOCKED RECORDS



VARIABLE-LENGTH, UNBLOCKED RECORD



VARIABLE-LENGTH, BLOCKED RECORDS

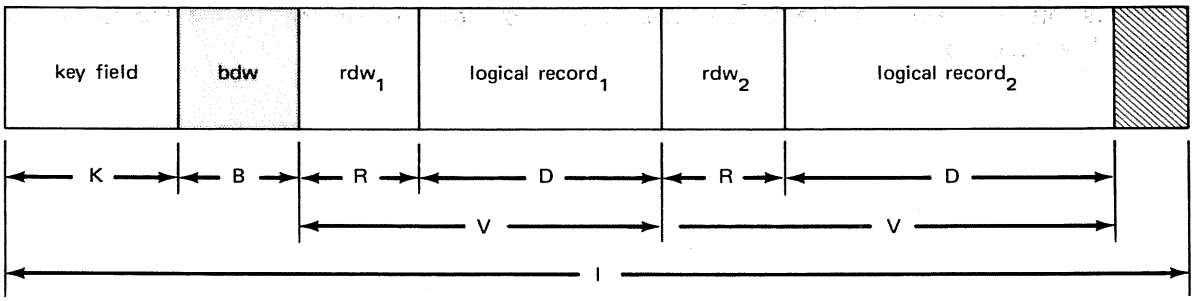


Figure 14—4. Keyed Fixed- and Variable-Length Physical Record Formats, Nonindexed Disk Files (Part 1 of 2)

LEGEND:

K Key field, supplied by you. Minimum 3 bytes, maximum 255. All keys in same partition or file must have same length. Notice that only a block has a key; individual records within a block do not.

B Block descriptor word, always four bytes. You reserve space for this at the head of the block, after the keyfield; data management calculates and inserts block length into first two bytes. Last two bytes are reserved.

R Record descriptor word, always the first four bytes of your variable-length record. You determine the length of each logical record in bytes, including this 4-byte RDW, and place it in the first two bytes of the RDW. The last two bytes are reserved.

D Length of the data portion of your logical record; varies for variable-length records. Always the same for each fixed-length record throughout file.

V Length of a variable record; includes four bytes for the RDW. You insert this number (measured in bytes) into the first two bytes of the RDW, in binary form.

I Length of the physical block, both on disc and in your I/O area. The I/O area length you specify via the BLKSIZE keyword parameter must accommodate the largest physical block in a file of variable-length records.

Unused space, if any

Supplied by data management

NOTE:

In preparing these illustrations of blocked records, an arbitrary choice was made to show two logical records per physical block, in both the fixed- and variable-length formats. The actual number you choose is a matter of file design. You may not specify that records are blocked for DTFDA files.

Figure 14—4. Keyed Fixed- and Variable-Length Physical Record Formats, Nonindexed Disk Files (Part 2 of 2)

15. Nonindexed File Access Methods: Function and Operation

15.1. GENERAL

This section describes the nonindexed file processor system used with OS/3 data management. The nonindexed file processor system is a generalized input/output control system (IOCS) that enables you to create and maintain data files on all disk subsystems supported by OS/3 and to process these files in a sequential order, a random order, or by a combination of both sequential and random file processing techniques. It offers standard techniques for processing sequential access method (SAM) files and direct access method (DAM) files, as well as enhanced capabilities for processing nonindexed files by a combination of both methods.

The nonindexed processor system operates on disk files that you describe to OS/3 data management through DTF (define the file) declarative macroinstructions:

- SAM files are defined by the DTFSD declarative macro;
- DAM files by the DTFDA declarative macro; and
- nonindexed files through the DTFNI macro.

You provide descriptive information to the declarative macros by specifying the keyword parameters associated with each. From these, data management builds the appropriate DTF file table for the type of file you have selected.

You perform I/O operations upon your file by issuing imperative macroinstructions in your basic assembly language (BAL) program. These, providing the essential interface between your program and OS/3 data management, are what you use to access and generate data within your file and to position it for more effective processing. Data management communicates directly with the resident systems access technique (SAT) of the OS/3 supervisor for access to the physical input/output control system (PIOCS).

The transient routines invoked by the OPEN macro, for example, complete setting up your DTF file definition tables and perform certain checks to ensure the integrity of your definition. Once you have opened a file, you may then access it via the imperative processing macros (GET, PUT, CNTRL, READ, WRITE, etc). You terminate your file processing by issuing a CLOSE macro and can then do no further processing on the data in the file until you reopen it by issuing another OPEN macro.

When you are processing files sequentially, the GET macro reads data blocks and delivers individual records to you, one at a time. You output records to the disk file with the PUT macro, which you may also use to update records. Data management provides buffering via blocked records and automatically blocks and deblocks them for you. You may overlap your I/O with your sequential processing by specifying a work area, or a second I/O area, in addition to the one I/O area that is always required for each file. As is true throughout OS/3 data management, your I/O areas must be half-word aligned.

For randomly processing files, data management offers you the capability to erase data from an expired or newly allocated file, to create blocks in a file, or to expand a file by generating data blocks in space newly allocated to it. You retrieve records via the READ macro and output them via the WRITE macro; each of these macros has several forms, which you specify as required, to access a block by its relative disk address, or by a key to be matched via search and an address for starting the search.

You may subdivide each nonindexed file, defined by the DTFNI declarative macro, into as many as seven file partitions, each having its own I/O area and characteristic block size, record size, and key length. Each partition of a DTFNI file may be further subdivided into as many as 71 serial subfiles. You may gain access to the specific partition and subfile required by issuing special imperative macros (SETP, SETS, for example) provided for this purpose.

For all three file types, OS/3 data management gives you the option of generating and processing your own standard user header and trailer labels. You may accompish this by coding a special user label processing routine, which receives control from the OPEN and CLOSE transients. You should remember that user standard labels are associated with your *file* and are not maintained at the partition or subfile level. Your label processing routine, furthermore, may not issue any of the file processing imperative macros, although a special macro, LBRET, is available for use in this routine.

It is important for you to keep in mind, as you study this section, that differences exist between OS/3 data management and the data management of SPERRY UNIVAC or other systems you may be acquainted with. Another point to remember is that some macros are used for other types of OS/3 data management files than those described in this section, with slight differences in format or effect. For example, the imperative macro SETF (15.7.8) is also used for magnetic tape files defined by the DTFMT declarative macro, but in its tape application has no UPDATE positional parameter.

Following functional descriptions of each of the three file processing methods, this section of the manual presents a discussion of the DTF declarative macros. A summary of the 37 keyword parameters (Table 15—7) and a description of their use follow these. The next paragraphs present a summary of the 18 imperative macros used in processing your files, and a detailed description of the use of each follows the summary. The final paragraphs of this section discuss the error and exception handling features of the nonindexed file processor system. You will find numerous coding examples throughout.

15.2. FUNCTION DESCRIPTION, OS/3 SAM

As you have just noted, you use the DTFSD declarative macro to define disk files that data management is to process as sequential access files. During your processing of DTFSD files, you should keep in mind that multivolume DTFSD files are maintained in single-volume-mounted mode, only the current volume being online at any one time. Data management takes care of communicating with the operator to request and validate the mounting of subsequent DTFSD volumes automatically. Files defined as sequential access by the DTFSD declarative macro may contain fixed- or variable-length records, blocked or unblocked.

For input operations, you use the GET macro; data management reads the input data and, deblocking it automatically if required, delivers the records, one at a time, to you. For output, you deliver records to data management one at a time via the PUT macro; when a full block of records is ready, data management writes it to disk. You may process input or output records in a work area or directly in the I/O area.

If you use a work area, data management moves an output record from the work area to the I/O area; it moves an input record from the I/O area to the work area. If your records are blocked, or if you provide two I/O areas without a work area, you must supply an index register (via the IOREG keyword parameter of your DTF), into which data management places the starting address of the current record position in the I/O area.

You may overlap your I/O operations with your other processing if you provide one or more work areas or a second I/O area in addition to the one I/O area you must always provide.

When you are processing variable-length, blocked records in the output mode and do not provide a work area, you must yourself test whether the next record will fit in the space remaining in the current output area. When it does not, you issue a TRUNC imperative macro to output a truncated block to disk. Data management provides you with the number of bytes remaining in the I/O area in a register that you specify via the VARBLD keyword parameter of your DTFSD declarative macro.

When you are processing input records from a DTFSD file, you may reach a point where you want to omit processing the records remaining in the current block or volume and begin with the first record of the next. You may accomplish either of these actions by issuing the RELSE imperative macro for skipping the remaining records in the current block, or the FEOV macro for skipping the remainder of the current volume.

When necessary, you may use the PUT imperative macro to update records you have retrieved, but you may not change their record length. You issue the PUT macro for the record to be updated following the GET macro by which it is retrieved.

In addition, you may extend a DTFSD file beyond the current end-of-file (EOF) address of the last volume of the file. There are two ways of doing this: in the *update* mode, you provide for this extension by coding it in your EOF routine; for *output* files, you specify the extend mode of processing in the LFD job control statement of the device assignment set by which you allocate the file. (The details of these two methods are developed under the discussion of the PUT macro; see 15.7.9.2 and 15.7.9.3.)

To use a DTFSD file as a disk work file, creating and then processing it under the same DTF file descriptor, you initially specify TYPEFLE=INOUT in the DTF, open the file for output and create it, and then close it. Changing the file processing direction to input via the SETF imperative macro, you then reopen the file, using the same DTF, to retrieve and, optionally, update your records. The details of this method are developed under the discussions of the SETF macro (15.7.8) and the PUT macro (15.7.9.1).

15.3. FUNCTIONAL DESCRIPTION, OS/3 DAM

You define DAM files, which you process at random with OS/3 data management, by means of the DTFDA declarative macro.

Another way in which OS/3 DAM differs from OS/3 SAM and OS/3 nonindexed file access method is that DAM supports fixed- and variable-length records in *unblocked* format only, whereas the other two also support blocked records. Consequently, you have no need of a work area or a second I/O area, as you do for processing sequential blocked records or for overlapping I/O with processing, and therefore the DTFDA declarative macro does not have IOAREA2, IOREG, RECSIZE, nor WORKA keyword parameters associated with it.

A third point of difference to remember between SAM and DAM is that all volumes of a multivolume DAM file are kept online when you are processing; there is no EOF concept in the SAM sense and consequently no EOFADDR keyword parameter associable with the DTFDA declarative macro, nor is there a combined input/output file type (TYPEFLE=INOUT), or need for an UPDATE keyword parameter.

OS/3 DAM provides you with a means for creating records in a file, erasing data from an old or newly allocated file, and expanding a file by generating records in space newly allocated to it.

The input work horse for direct access files is the block-level READ imperative macro, which has two forms (READ,KEY and READ,ID) for accessing a block through a search on key (starting the search at an address you specify) and for accessing a block directly at a relative disk address (ID) that you provide to data management. You indicate to data management which of these forms of the READ macro you intend to use by specifying the READKEY and READID keyword parametes in your DTFDA declarative, and you provide

data management with READ-related information in a number of other DTF keyword parameters, discussed in 15.7.14. To give data management time to locate the block you want and to move it to the I/O area, you must issue a WAITF imperative macro after every READ macro issued to a DTFDA file, before you may issue another imperative to the file. This assures you that the data transfer has taken place as specified before you proceed further (15.7.16).

Another imperative macro for DTFDA files that must be paired with a following WAITF macro — and for the same reasons — is the direct access output processing macro, WRITE, which, like READ, is also a block-level macro. The WRITE macro has five forms that may be used with each other and with the two READ macro forms to create, update, and erase direct access disk files. Two of these forms (WRITE,RZERO and WRITE,AFTER,EOF) do not actually output a block to disk. You use these to reposition the disk access arm to a new track, or to record the ID returned after the last block was written as the end of data in the file. These forms of the WRITE command, and the WRITE-related DTFDA keyword parameters, are developed in detail in 15.7.11 and 15.6.

15.4. FUNCTIONS OF THE OS/3 NONINDEXED FILE ACCESS METHOD

The logical IOCS processor, which processes nonindexed files you define with the DTFNI declarative macro, will also process sequential files defined by the DTFSD macro and direct access files you define with the DTFDA macro. It supports all OS/3 random access file processing imperative macros and all the sequential processing macros; however, only files you define by the *DTFNI* macro may be processed by the combination of *both* direct and sequential processing techniques. Similarly, it is important to remember that only to *DTFNI* files may you issue the following five imperative macros: NOTE, POINT, POINTS, SETP, and SETS.

The DTFNI declarative macro has a number of unique keyword parameters you will specify to realize other extensions to file processing techniques which apply to nonindexed disc files under OS/3 data management: the PCA and SUBFILE keyword parameters, for example, by which you specify that your DTFNI file is subdivided into file partitions and that these, in turn, are subdivided into partition subfiles. You will note also the SIZE and UOS (*unit of store*) keyword parameters, which you use for dynamic allocation and extension of DTFNI files to the partition level.

You may subdivide a DTFNI file into as many as seven file partitions, each of which has its own I/O area or areas and its own characteristic record size, block size, and record format. Each partition you may further subdivide into a maximum of 71 subfiles, having the same characteristics as the partition of which it is a part. You define a DTFNI file as being partitioned by specifying two or more PCA keyword parameters in the DTF and by coding a DPCA (*define partition control appendage*) declarative macro for the second through the seventh partitions of the file. In the DPCA declaratives, you specify a keyword parameter for each aspect in which the partition differs from the parent DTFNI file; for this reason, you do not prepare a separate DPCA description for the first partition — it is already fully described in the DTFNI macro.*

**It is not mandatory, of course, that each partition of a DTFNI file have different record lengths or formats; however, even though a partition may be the same as the parent DTFNI file in all other respects, if it is a partition it has its own separate name, blocksize, and I/O area and requires its own DPCA declarative macro for separate identity and accessibility.*

When you open the DTFNI file for processing, only the *first* partition is set active and available to you for processing; to gain access to some other partition, you issue the SETP imperative macro to the file and specify the partition you want to process by name. Subsequent processing is carried out on this partition until you select another by issuing another SETP macro. When you have selected the partition you want to process, you issue a SETS imperative macro to specify whichever subfile it is you want to process, having previously indicated that data management is to support subfiles for this file via the DTFNI or DPCA keyword parameter SUBFILE. These procedures are detailed further in the descriptions of the DTFNI and DPCA declaratives (15.5.3 and 15.5.4) and the SETP and SETS imperative macros (15.7.4 and 15.7.5).

OS/3 data management will allocate and extend DTFNI space automatically for you, as you need it, using the disk space management routines of the OS/3 supervisor and taking its cues from both your job control statements and your DTFNI/DPCA keyword parameters. It satisfies the space requirement for the first partition from the first available area of the extents you specified in the device assignment set of job control statements by which you initially allocate the file; to the first partition it allocates that percent of the total file allocation you have specified in the SIZE keyword parameter of the DTFNI declarative macro.

You use the UOS keyword parameter to indiate the percentage of additional space data management will dynamically allocate to this partition, should you ever issue a WRITE or PUT output imperative macro referencing a block or record that lies beyond the current maximum relative block address for the partition. The UOS keyword parameter specifies the percentage of the secondary allocation data management may assign. (You will have specified the total number of tracks or cylinders by which the file may be extended dynamically in the third positional parameter of the EXT job control statement in your device assignment set for the file.)

Further details on dynamic extension are given in descriptions of the SIZE and UOS keyword parameters (15.6.24 and 15.6.29); each DPCA declarative macro has its unique SIZE and UOS keyword parameters. You should also remember that DTFNI files will not be dynamically extended beyond the volumes on which they initially reside.

Should you ever want to expand a growing DTFNI file beyond the physical volumes you originally allocated to it (assuming that these volumes are still full after your efforts to reorganize the file and scratch expired or unused portions of it), you will need to copy the file sequentially off to tape or other disk devices, using one of the OS/3 data utility programs, redefine the file with a new DTFNI macro, reallocate this to new devices, and, copying it back, effectively create a new file. (The OS/3 data utility programs are described in the data utilities user guide, UP-8069 (current version).

Other enhancements that apply only to processing your DTFNI files are the NOTE and POINT imperative macros. You use the NOTE macro to access the partition-relative address of the current record or block, which data management places in the DTFNI file table for you to reference. You may then use the current address for file positioning via the POINT macro, which updates the current record address in the DTF as you direct; your subsequent file processing proceeds from this updated address. (Further details are developed on the NOTE and POINT macros in 15.7.17 and 15.7.18.)

The imperative macro POINTS is useful to you for initializing the relative block address of the current partition; you select the current partition, as previously described, with the SETP macro. (The POINTS macro is fully described in 15.7.6.)

15.5. NONINDEXED DISK FILE DECLARATIVE MACROS

You will use the four DTF declarative macros described in the following paragraphs to define disk files and partitions to OS/3 data management. Note that, although the DTF keyword parameters listed in the following statement formats are presented in alphabetic order, you may code these in any convenient order, just so you separate them with commas.

Following the statement formats are tables summarizing the required and optional keyword parameters; the detailed descriptions of the keyword parameters are presented in 15.6 and a table recapitulating all of them follows the descriptions. Except for the LACE keyword (15.6.8) and the LOCK keyword (15.6.36), the keywords are described in alphabetic order.

Refer to the Preface of this manual to review OS/3 statement conventions, and to 1.6 for a general discussion of DTF macros and BAL rules for coding their operands.

In the declarative macroinstruction format delineations that follow, a comma is shown preceding each keyword parameter except the first, to remind you that all keywords coded in a string must be separated by commas. However, a comma must not be coded in column 16 of a continuation line, nor follow the last keyword in the string. Refer to the coding examples that follow.

DTFSD

15.5.1. Defining a Sequential Disk File (DTFSD)

Function:

You will use the DTFSD declarative macro instruction to define input and output disk files that you intend to process sequentially. The DTFSD macro establishes a 242-byte file table. Table 15—1 summarizes the DTFSD keyword parameters. These are described in detail in 15.6. A coding example follows Table 15—1.

Format:

LABEL	ΔOPERATIONΔ	OPERAND
filename	DTFSD	[ACCESS= {EXC, EXCR, SRDO, SRD}]
		BLKSIZE=n
		,EOFADDR=symbol
		[,ERROPT= {IGNORE, SKIP}]
		[,ERROR=symbol]
		,IOAREA1=symbol
		[,IOAREA2=symbol]
		[,IOREG=(r)]
		[,LABADDR=symbol]
		[,LACE=n]
		[,LOCK=NO]
		[,OPTION=YES]
		[,RECFORM= {FIXBLK, FIXUNB, VARBLK, VARUNB}]
		[,RECSIZE=n]
		[,SAVAREA=symbol]
		[,TRLBL=YES]

(continued)

LABEL	Δ OPERATION Δ	OPERAND
	DTFSD (cont)	[,TYPEFLE= {INOUT / INPUT / OUTPUT}] [,UPDATE=YES] [,VARBLD=(r)] [,VERIFY=YES] [,WORKA=YES]

Table 15–1. Summary of Keyword Parameters for DTFSD Macro Instruction (Part 1 of 2)

Keyword	Specification	Files			Remarks
		Input	Output	In/Out	
ACCESS*	EXC	X	X	X	Request exclusive use of file for associated DTF
	EXCR	X	X	X	Request exclusive use of file for associated DTF while permitting read use for other jobs
	SRD	X			Request read function for file associated with DTF while permitting read/write for other jobs
	SRDO	X			Request read function for both files associated with DTF as well as other jobs. No writing permitted.
BLKSIZE*	n=maximum block size	R	R	R	The maximum block size, in bytes
EOFADDR	symbolic label	R		R	Identifies EOF routine
ERROPT	IGNORE	X	X	X	Ignore parity error
	SKIP	X	X	X	Bypass parity error
ERROR	symbolic label	X	X	X	Address of user's unrecoverable error routine
IOAREA1	symbolic label	R	R	R	Address of I/O area
IOAREA2	symbolic label	X	X	X	Address of alternate I/O area
IOREG	(r)=general register	X	X	X	Required if records are processed in the I/O area and records are blocked
LABADDR	symbolic label of user's label routine		X	X	Required if user header or trailer labels are to be created
		X			Required if user header or trailer labels are to be retrieved
LACE*	n=lace factor	X	X	X	Specifies factor for record interlace
LOCK	NO	X	X	X	Specifies that file lock is not to be set on a lockable file at OPEN, permitting read-only access
OPTION	YES	X			Specifies file not always to be processed

Table 15–1. Summary of Keyword Parameters for DTFSD Macro Instruction (Part 2 of 2)

Keyword	Specification	Files			Remarks
		Input	Output	In/Out	
RECFORM*	FIXBLK	Y	Y	Y	For fixed-length, blocked records
	FIXUNB	X	X	X	For fixed-length, unblocked records; assumed
	VARBLK	Y	Y	Y	For variable-length, blocked records
	VARUNB	Y	Y	Y	For variable-length, unblocked records
RECSIZE*	n=number of bytes in record	X	X	X	For fixed-length, blocked records
SAVAREA	symbolic label	X	X	X	Specifies address of save area for contents of general registers
TRLBL	YES	X	X	X	Read or write user trailer labels when CLOSE issued to file. (Specify LABADDR also.)
TYPEFLE	INOUT			R	For I/O files
	INPUT	X			For input files; assumed
	OUTPUT		R		For output files
UPDATE	YES	X		X	Required if records are to be written back to the same location from which they were read
VARBLD	(r)=general register		X	X	Required for variable-length, blocked records built in output area; register contains number of bytes left in output area.
VERIFY	YES	X	X	X	Check parity after records have been written.
WORKA	YES	X	X	X	Process records in work area.

LEGEND:

R Required
X Optional
Y One option required

Assumed parameter, if none specified
* Parameter may be changed on DD job control statement.

Example:

```
LABEL     ΔOPERATIONΔ     OPERAND
1         10              16                                  72  80
ACCNTS    DTFSD           BLKSIZE=800,                        X
                          IOAREA1=READ,                       X
                          EOFADDR=FINIS,                      X
                          RECFORM=FIXBLK,                     X
                          RECSIZE=100,                        X
                          UPDATE=YES,                         X
                          WORKA=YES
```

This example defines a file ACCNTS that is a SAM input file (by TYPEFLE default option). The required I/O area is designated symbolically as READ. The block size is 800 bytes, record size is 100 bytes, and records are fixed and blocked. An end of file routine FINIS has been specified to handle that occurrence. No special label handling or error routines are provided; therefore, errors will return inline.

DTFDA

15.5.2. Defining a Direct Access Disk File (DTFDA)

Function:

You will use the DTFDA declarative macroinstruction to define files that are to be randomly processed. The DTFDA macro establishes a 242-byte file table. A summary of DTFDA keyword parameters is presented in Table 15—2; these are described in detail in 15.6. A coding example follows the table.

Format:

LABEL	Δ OPERATION Δ	OPERAND
filename	DTFDA	[ACCESS= {EXC, EXCR, SRD, SRDO}]
		[AFTER=YES]
		,BLKSIZE=n
		[,ERROR=symbol]
		[,IDLOC=symbol]
		,IOAREA1=symbol
		[,KEYARG=symbol]
		[,KEYLEN=n]
		[,LABADDR=symbol]
		[,LACE=n]
		[,LOCK=NO]
		[,READID=YES]
		[,READKEY=YES]
		[,RECFORM= {FIXUNB, VARUNB}]
		[,RELATIVE= {R, T}]
		[,SAVAREA=symbol]
		,SEEKADR=symbol
		[,SRCHM=YES]
		[,TRLBL=YES]

(continued)

LABEL	Δ OPERATION Δ	OPERAND
	DTFDA (cont)	[,TYPEFLE= {INPUT / OUTPUT}] [,VERIFY=YES] [,WRITEID=YES] [,WRITEKEY=YES]

Table 15–2. Summary of DTFDA Keyword Parameters (Part 1 of 2)

Keyword	Specification	Files		Remarks
		Read	Write	
ACCESS*	EXC	X	X	Request exclusive use of file for associated DTF
	EXCR		X	Request exclusive use of file for associated DTF while permitting read use for other jobs
	SRD	X		Request read function for file associated with DTF while permitting read/write for other jobs
	SRDO	X		Request read function for both files associated with DTF as well as other jobs. No writing permitted.
AFTER	YES		X	A capacity record on each track is assumed.
BLKSIZE*	n=maximum block size	R	R	Length of IOAREA1, in bytes
ERROR	symbolic label	X	X	Address of user error routine
IDLOC	symbolic label	X	X	Address of field containing the record ID
IOAREA1	symbolic label	R	R	Name of I/O area defined by user
KEYARG	symbolic label	X	X	Address of field for key used for key search
KEYLEN*	n=key length	X	X	Length of the key in bytes
LABADDR	symbolic label	X	X	Address of user label handling routine
LACE*	n=lace factor	X	X	Specifies factor for record interlace
LOCK	NO	X	X	Specifies that file lock is not to be set on a lockable file at OPEN, permitting read-only access.
READID	YES	X		Record referenced by ID
READKEY	YES	X		Record referenced by KEY
RECFORM*	FIXUNB	Y	Y	For fixed-length records
	VARUNB	Y	Y	Variable-length records
RELATIVE	R	X	X	Relative addressing – record
	T	X	X	Relative addressing – track

Table 15–2. Summary of DTFDA Keyword Parameters (Part 2 of 2)

Keyword	Specification	Files		Remarks
		Read	Write	
SAVAREA	symbolic label	X	X	Specifies the address of a save area for contents of general registers
SEEKADR	symbolic label	R	R	Address of track reference field
SRCHM	YES	X	X	Search multiple tracks. (If specified, file must be allocated on a cylinder basis.)
TRLBL	YES	X	X	User standard trailer labels are to be read or written when CLOSE issued to file. Specify LABADDR also.
TYPEFLE	INPUT	Y	Y	Check standard labels
	OUTPUT	Y	Y	Write standard labels
VERIFY	YES		X	Records are to be check-read
WRITEID	YES		X	Output record is located by means of its relative disc address (ID).
WRITEKEY	YES		X	Output record is located by key.

LEGEND:

R Required
X Optional
Y One option required
Assumed parameter, if none specified
* Parameter may be changed on DD job control statement.

Example (DAM Input File):

```
LABEL     OPERATION     OPERAND                    COMMENTS
1         10            16                                                    72
*    DEFINE THE FORMAT OF A DIRECT ACCESS INPUT FILE
*         LABELED INFILE
INFILE    DTFDA    BLKSIZE=1111,     INAREA IS 1111 BYTES LONG              1
                   IOAREA1=INAREA,   INAREA IS THE INPUT/OUTPUT AREA        4
                   KEYLEN=20,        EACH RECORD HAS A 20-BYTE KEY          5
                   READID=YES,       READ INFILE,ID WILL BE ISSUED          6
                   RECFORM=FIXUNB,   ALL RECORDS ARE THE SAME SIZE          7
                   SEEKADR=IDADR     IDADR IS A 4-BYTE FIELD - DISC ID
```

Example (DAM Output File):

```
OUTFILE   DTFDA    BLKSIZE=512,                                             X
                   IOAREA1=OUTPUT,                                          X
                   TYPEFLE=OUTPUT,                                          X
                   VERIFY=YES,                                              X
                   KEYLEN=16,                                               X
                   WRITEID=YES,                                             X
                   READID=YES
```

DTFNI

15.5.3. Defining a Nonindexed Disk File (DTFNI)

Function:

You will use the DTFNI declarative macro instruction to define disk files that are to be processed sequentially, randomly, or by a combination of sequential and direct access processing techniques. The DTFNI macro establishes a 242-byte file table. See 15.5.4 for a description of the DPCA declarative macro, which you use to define the second and all subsequent partitions of a partitioned DTFNI file. Table 15—3 summarizes the DTFNI and DPCA keyword parameters; coding examples follow the table. Keyword parameters are detailed in 15.6.

Format:

LABEL	Δ OPERATION Δ	OPERAND
filename	DTFNI	[ACCESS= {EXC, EXCR, SRD, SRDO}] [AFTER=YES] ,BLKSIZE=n [,ERROPT= {IGNORE, SKIP}] [,EOFADDR=symbol] [,ERROR=symbol] [,IDLOC=symbol] ,IOAREA1=symbol [,IOAREA2=symbol] [,IOREG=(r)] [,KEYARG=symbol] [,KEYLEN=n] [,LABADDR=symbol] [,LACE=n] [,LOCK=NO] [,OPTION=YES] [[,PCA1=symbol],...,[PCA7=symbol]] [,READID=YES] [,READKEY=YES]

LABEL	Δ OPERATION Δ	OPERAND
	DTFNI (cont)	[,RECFORM= {FIXBLK / FIXUNB / VARBLK / VARUNB}] [,RECSIZE=n] [,RELATIVE= {R / T}] [,SAVAREA=symbol] ,SEEKADR=symbol [,SRCHM=YES] [,SIZE=n] [,SUBFILE=YES] [,TRLBL=YES] [,TYPEFLE= {INOUT / INPUT / OUTPUT}] [,UOS=n] [,UPDATE=YES] [,VARBLD=(r)] [,VERIFY=YES] [,WORKA=YES] [,WRITEID=YES] [,WRITEKEY=YES]

DPCA

15.5.4. Defining a Partition Control Appendage (DPCA)

When you have a DTFNI file that is to contain two or more partitions, you use a DPCA declarative macro to define separately the second and each of the subsequent partitions (that is, partitions 2 through 7). Descriptive information for the first partition (and for the parent file as a whole) is contained in the 242-byte DTFNI file table. The DPCA declarative macro sets up an auxiliary file table, called a *partition control appendage,* which defines the aspects in which each partition differs from the first and occupies only 82 bytes of main storage.

Notice that the label of each DPCA macro is **partition-name**; this is the symbolic label of the partition you have specified in the corresponding PCA keyword parameter of the parent DTFNI macro.

You should also note the absence of all strictly *file*-relative keyword parameters from the DPCA macro: ERROPT, ERROR, and LABADDR, for example. The reason for this is to simplify matters for you: OS/3 data management relies upon the DTFNI file table for all file-relative keyword parameters, and you do not specify these again in the DPCA macros.

Following the format statement, Table 15—3 summarizes the DTFNI and the DPCA keyword parameters; these are, in turn, detailed in 15.6. Coding examples follow the table.

Function:

You use the DPCA macro to define the second and all subsequent partitions of a multipartitioned DTFNI file. A maximum of seven partitions may be defined in all. The DTFNI file table contains descriptive information for the first partition. Separate DPCA macros establish *partition control appendages* defining each of the others (partitions 2 through 7).

Format:

LABEL	Δ OPERATION Δ	OPERAND
partition-name	**DPCA**	**BLKSIZE=n** **[,EOFADDR=symbol]** **,IOAREA1=symbol** **[,IOAREA2=symbol]** **[,IOREG=(r)]** **[,KEYARG=symbol]** **[,KEYLEN=n]** **[,LACE=n]**

LABEL	Δ OPERATION Δ	OPERAND
	DPCA (cont)	[,RECFORM= {FIXBLK / FIXUNB / VARBLK / VARUNB}] [,RECSIZE=n] [,SIZE=n] [,SUBFILE=YES] [,UOS=n] [,VARBLD=(r)] [,WORKA=YES]

Table 15–3. Summary of DTFNI and DPCA Keyword Parameters (Part 1 of 3)

Keyword	Specification	Used for DPCA	Files			Remarks
			Input	Output	In/Out	
ACCESS*	EXC	No	X	X	X	Request exclusive use of file for associated DTF
	EXCR	No	X	X	X	Request exclusive use of file for associated DTF while permitting read use for other jobs
	SRD	No	X			Request read function for file associated DTF while permitting, read/write for other jobs
	SRDO	No	X			Request read function for both files associated with DTF as well as other jobs. No writing permitted
AFTER	YES	No		X	X	A WRITE, AFTER macro will be issued.
BLKSIZE*	n=maximum block size	R	R	R	R	Length of IOAREA1, in bytes
EOFADDR	symbolic label	X	X	X	X	Address to which control is passed when end of sequentially processed file is reached
ERROPT	IGNORE	No	Y	Y	Y	Ignore parity error
	SKIP	No	Y	Y	Y	Bypass parity error
ERROR	symbolic label	No	X	X	X	Address of user error routine
IDLOC	symbolic label	No	X	X	X	Address of field containing the record ID
IOAREA1	symbolic label	R	R	R	R	Name of I/O area defined by user; always half-word aligned
IOAREA2	symbolic label	X	X	X	X	Name of alternate I/O area
IOREG	(r) general register	X	X	X	X	Required if records are processed in the I/O area and records are blocked
KEYARG	symbolic label	X	X	X	X	Address of field for key used for key search
KEYLEN*	n=key length	X	X	X	X	Length of the key in bytes
LABADDR	symbolic label	No	X	X	X	Address of user label-handling routine
LACE*	n=lace factor	X	X	X	X	Specifies the factor for record interlace

Table 15–3. Summary of DTFNI and DPCA Keyword Parameters (Part 2 of 3)

Keyword	Specification	Used for DPCA	Files			Remarks
			Input	Output	IN/Out	
LOCK	NO	No	X	X	X	Specifies that file lock is not to be set on a lockable file at OPEN, permitting read-only access
OPTION	YES	No	X	X	X	Specifies an optional file
PCAn	symbolic label	No	X	X	X	Specifies the address of partitions (1 to 7) used to subdivide a file
READID	YES	No	X		X	A READ, ID macro will be issued.
READKEY	YES	No	X		X	A READ,KEY macro will be issued.
RECFORM*	FIXBLK	Y	Y	Y	Y	Fixed-length, blocked records
	FIXUNB	Y	Y	Y	Y	Fixed-length records, unblocked
	VARBLK	Y	Y	Y	Y	Variable-length, blocked records
	VARUNB	Y	Y	Y	Y	Variable-length records, unblocked
RECSIZE*	n=number of bytes in record	X	X	X	X	Specifies record size
RELATIVE	R	No	X	X	X	Relative addressing – record
	T	No	X	X	X	Relative addressing – track
SAVAREA	symbolic label	No	X	X	X	Specifies save area for contents of general registers
SEEKADR	symbolic label	No	R	R	R	Address of track reference field
SIZE*	n=percent	X	X	X	X	Specifies percentage of total file allocation to be initially assigned to partition
SRCHM	YES	No	X	X	X	Search multiple tracks. (If specified, file must be allocated on a cylinder basis.)
SUBFILE	YES	X	X	X	X	Support division of file partitions into subfiles
TRLBL	YES	No	X	X	X	Read or write user trailer labels when CLOSE issued to file. (Specify LABADDR also)
TYPEFLE	INOUT	No			R	This file may be used as either an input file or an output file.
	INPUT	No	X			Read and check standard labels for file. If file is processed sequentially, the PUT macro may not be issued for this file unless UPDATE=YES has also been specified in the DTF.
	OUTPUT	No		R		Write standard labels for this file. If file is processed sequentially, the GET macro instruction may not be issued.
UOS*	n=percent	X	X	X	X	Specifies percentage of secondary allocation to be used for extending partition
UPDATE	YES	No	X	X	X	Write records back into same location from which they were read.
VARBLD	(r)=general register	X	X	X	X	Required for variable-length, blocked records that are built in output area; register contains number of bytes left in output area.
VERIFY	YES	No		X		Records are to be check-read.

Table 15–3. Summary of DTFNI and DPCA Keyword Parameters (Part 3 of 3)

Keyword	Specification	Used for DPCA	Files			Remarks
			INPUT	OUTPUT	IN/OUT	
WORKA	YES	Yes	X	X	X	Process records in work area
WRITEID	YES	No		X	X	A WRITE, ID macro will be issued
WRITEKEY	YES	No		X	X	A WRITE, KEY macro will be issued

LEGEND:

R Required
X Optional
Y One option required
▒ Assumed parameter, if none specified
* Parameter may be changed on DD job control statement.

Examples of DTFNI declarative macros used to define a nonpartitioned, nonindexed file and a nonindexed file containing two partitions.

Example:

```
LABEL     OPERATION     OPERAND
1         10            16                                  72        80
NONPART   DTFNI         BLKSIZE=512,                        X
                        RECFORM=VARBLK,                     X
                        IOAREA1=BUFFER,                     X
                        VARBLD=(4),                         X
                        RELATIVE=R,                         X
                        SEEKADR=RELREC,                     X
                        VERIFY=YES,                         X
                        ERROPT=SKIP,                        X
                        ERROR=ERREXT
PARTIT    DTFNI         BLKSIZE=256,                        X
                        RECFORM=FIXBLK,                     X
                        RECSIZE=128,                        X
                        PCA1=ORGANS,                        X
                        PCA2=INFILE2,                       X
                        IOAREA1=BUFFER1,                    X
                        IOAREA2=BUFFER2,                    X
                        IOREG=(4),                          X
                        SIZE=5,                             X
                        UOS=IO,                             X
                        TYPEFLE=INOUT
```

Examples of DPCA declarative macros used to define partitions of a multipartitioned DTFNI file:

```
LABEL     ΔOPERATIONΔ     OPERAND                          72
1         10              16                               |
                                                           |
OUTFILE2  DPCA            BLKSIZE=512,                     X
                          EOFADDR=ENDJOB,                  X
                          LACE=2,                          X
                          IOAREA1=PRIME,                   X
                          IOAREA2=SECOND,                  X
                          IOREG=(3),                       X
                          RECFORM=FIXBLK,                  X
                          RECSIZE=128,                     X
                          SIZE=2

INFILE2   DPCA            BLKSIZE=1024,                    X
                          IOAREA1=INAREA1,                 X
                          IOAREA2=INAREA2,                 X
                          RECSIZE=256,                     X
                          WORKA=YES
```

15.6. KEYWORD PARAMETERS FOR DECLARATIVE MACROS

The following paragraphs describe each of the 36 required and optional keyword parameters used as the operands of the declarative macros that are part of the nonindexed file processing system of OS/3 data management. These declarative macros include the DTFSD, DTFDA, DTFNI, and DPCA declarative macros, which you use to define your disk files and partitions. The keyword parameters are presented in alphabetic order for ease of reference. Subsection 15.6.36 lists certain nonstandard forms of these parameters that are supported by OS/3 data management so that existing programs prepared for other systems may be run under OS/3 with minimum change.

Table 15—7 shows which keyword parameters are used with which macro; if you code a keyword parameter for a declarative macro that does not support it, this error will be noted at assembly time, and flagged in your assembly listing with an appropriate, self-explanatory error message.

The following descriptions also include notations as to which declarative macros support each keyword parameter and point out any difference in meaning of a keyword when it is applied to the various file definitions or processors.

15.6.1. Specifying File Accessing Options (ACCESS)

For a description of the ACCESS keyword parameter, see 11.4.1.

Records added by the writer (ACCESS=EXCR) to a file, in a shared environment that permits one writer and any number of readers, are not available to the reader (ACCESS=SRD). Once the writer closes the job, any added records will be available to users who subsequently open the file.

15.6.2. WRITE,AFTER or WRITE,RZERO Macro Issue (AFTER)

When you intend to issue one of the following forms of the WRITE macro to a DTFDA file or to a DTFNI file you are processing randomly, you must specify the AFTER keyword parameter in your DTF:

WRITE,AFTER	(15.7.11.1)
WRITE,AFTER,EOF	(15.7.11.3)
WRITE,RZERO	(15.7.11.2)

If you specify the AFTER keyword parameter in the DTF for a file that you are creating on a variable-sector 8411, 8414, 8424, 8425, 8430, or 8433 disk, data management expects that you will do so using these macros and does not preformat the file at OPEN time. Because the WRITE,ID macro conducts a search and relies on preformatting, you may not issue this macro to a file on a variable-sector disk when you have specified the AFTER keyword parameter; data management can find no relative disk addresses to match against the content of your SEEKADR field. See the description of the WRITE,ID macro (15.7.11.4).

The foregoing restriction does not apply to a file on a fixed-sector 8416 disk, which is preformatted by definition.

Keyword Parameter AFTER:

AFTER=YES
Specified if a subsequent WRITE macroinstruction contains an AFTER or an RZERO positional parameter. This keyword parameter is used only when creating or adding blocks sequentially, marking the end of data in the file, or repositioning the disk head to a new track.

15.6.3. Specifying Block Length (BLKSIZE)

Each input or output file and partition must have at least one I/O area (buffer) reserved for its individual use. The symbolic address of this area is defined by the IOAREA1 keyword parameter, which is required for all DTFs (15.6.10), but you also need the BLKSIZE keyword parameter to specify the maximum length of the blocks placed within it. The BLKSIZE keyword parameter is therefore required for DTFSD, DTFDA, DTFNI, and DPCA declarative macros.

These factors determine the size of your I/O area, which you specify elsewhere in your program with the BAL *define constant* (DC) or *define storage* (DS) statement:

1. The length of the data to be read or written — If the records in the file or partition are variable-length, the block size and buffer must accommodate the length of the largest record, which includes the data length and the four bytes reserved at the head of each record for the record descriptor word (RDW). Fixed-length records do not contain an RDW (14.3.2).

2. The track capacity of the device on which the file is written — This must not be exceeded by fixed- or variable-length records. (See Appendix A for the operational characteristics of the disk subsystems supported by OS/3.)

3. Whether records are variable-length — For blocked or *unblocked* variable records, your block size and buffer length specifications must include the length of the largest logical record plus the four bytes required for the block descriptor word (BDW) at the head of each block. This point is important for you to remember when you are processing files defined by the DTFDA macro, which does not support blocked record formats — a BDW is calculated by data management for the blocks of these records as well. Look for this in Figure 15—1.

4. The length of the record key — You specify the key length of the KEYLEN keyword parameter when you are referencing blocks by key, generating new blocks with keys, or updating or reading both key and data areas of blocks (15.6.14). Therefore, whenever you specify the KEYLEN keyword parameter, your block size must also include the length of the key.

5. Whether you are processing optional user labels — User header labels (UHLs) and user trailer labels (UTLs) have a standard length of 80 bytes. The I/O area must be at least 80 bytes long when these labels are to be processed in your program; in OS/3, UHLs and UTLs are handled only in the I/O areas, not in a work area. Refer to 14.2.4, 15.6.15, and 15.7.3.

An important point to note is that, although you must always reserve an 8-byte field immediately *preceding* the I/O area for data management use when you are processing output files, these eight bytes are not included in your block size specification. Remember, also, that the I/O area is always half-word aligned; this is true throughout OS/3 data management.

For example, it would be incorrect to reserve a 68-byte field when your BLKSIZE specification is 60 bytes; the following coding example shows one way to do this in OS/3. Other systems you may be familiar with do it differently.

Example:

	LABEL (1)	OPERATION (10)	OPERAND (16)	72	
1.	SAMPFLE	DTFNI	BLKSIZE=60,	X	
2.			IOAREA1=IODAM,	X	
3.			TYPEFLE=OUTPUT,	X	
4.			.	X	
			.	X	
			.		
5.	ALIGNHAF	DS	OH		
6.	DMGTUSE	DS	CL8		
7.	IODAM	DS	CL60		

1. SAMPFLE is a DTFNI file; block size is 60 bytes.

2. I/O buffer's symbolic address is IODAM.

3. SAMPFLE is an output file; data management requires eight bytes for its use immediately preceding the buffer.

4. Other keyword parameters are not shown. The following *define storage* (DS) statements must be coded elsewhere in your program, not as part of the DTFNI macro call.

5. Half-word alignment required for all I/O buffers in OS/3 data management. No label required.

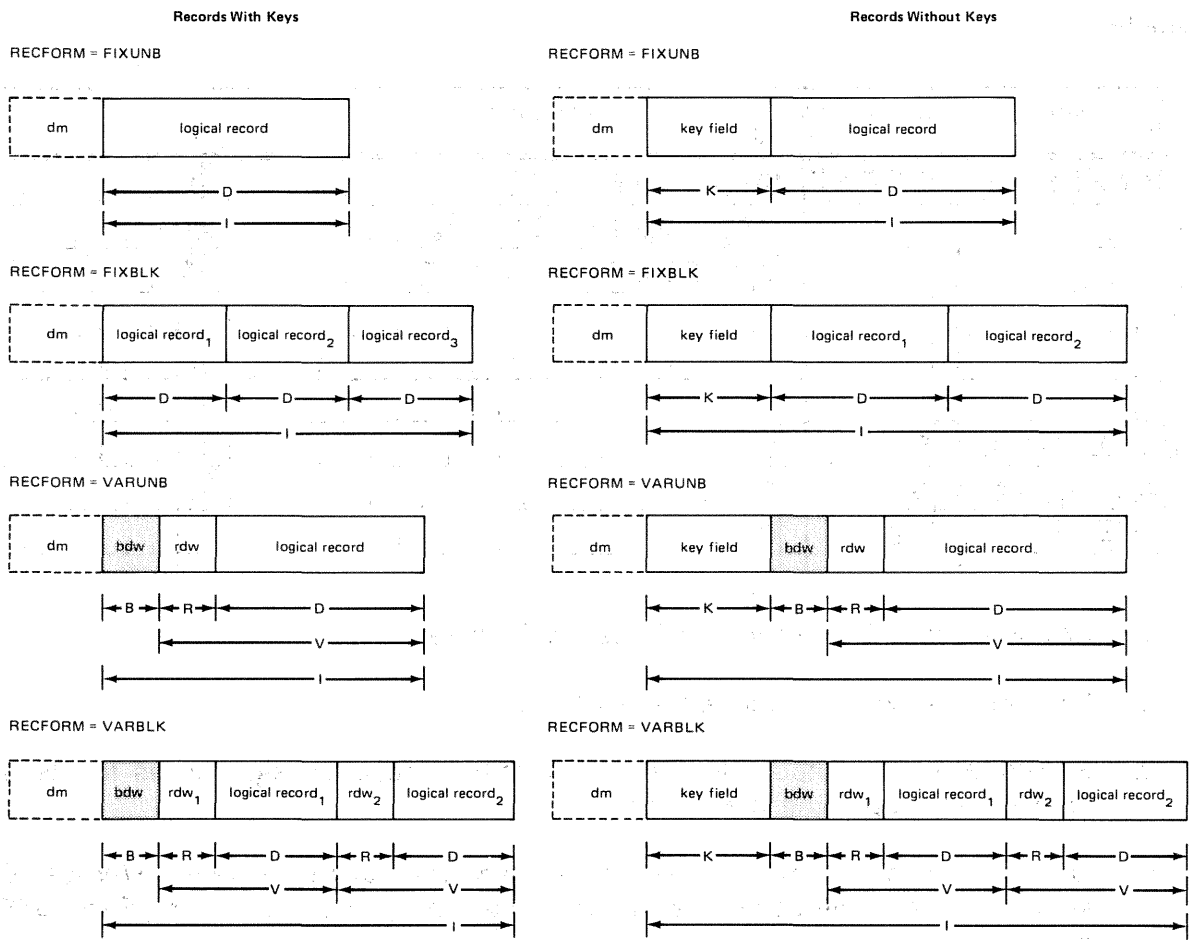
6. Eight bytes are required for data management use immediately preceding the I/O buffer. No label required.

7. Storage area IODAM defined as 60 bytes in length, not 68.

NOTE:

Lines 5 and 6 were written as shown merely to document two separate points explicitly. Most BAL programmers will probably replace these two lines of coding with a single define storage *(DS) statement having simply a* D *(double word) for an operand. Such an instruction will reserve eight bytes and force half-word alignment — a neater way to discharge both requirements at once.*

Figure 15—1 illustrates the contents of the I/O area under some of the situations just described; Table 15—4 summarizes what data management moves into the I/O area when you specify certain combinations of the keyword parameters in your DTF.



LEGEND:

D Data portion of logical record. Data Management assumes this equals block size when records are fixed, unblocked.

I Length of IOAREA1; specified by BLKSIZE. I/O area is always half-word aligned.

R Record descriptor word (RDW), a 4-byte record-length field containing the length of the record in the first two bytes; inserted by user.

B Block descriptor word (BDW), 4-byte block length field containing the length of the block in the first two blocks; calculated by data management.

V Variable-length record; length is specified by RECSIZE and is contained in first two blocks of RDW.

K Keyfield, from 3 bytes to 255 bytes in length; its actual length is specified by KEYLEN except when you want keys ignored. (In this event, you specify KEYLEN=0 or omit the keyword.) Note that only a block, not a record, has a key.

dm Eight-byte field that you reserve for data management use with output files; immediately precedes I/O buffer but its length is not included in BLKSIZE. I/O area is always half-word aligned.

▒ Calculated by data management; user supplies space.

□ User-supplied.

Figure 15—1. Record Formats and I/O Area Contents for Nonindexed Disk Files

Table 15—4. IOAREA1 Contents

Keyword Specification	KEYLEN	AFTER	READID or WRITEID	I/O Area Contents
AFTER=YES	= 0	= YES	—	Data
	≠ 0	= YES	—	Key, Data
READID=YES or WRITEID=YES	≠ 0	= YES	= YES	Key, Data
	≠ 0	Not specified	= YES	Key, Data
	= 0	= YES	= YES	Data
	= 0	Not specified	= YES	Data
READKEY=YES or WRITEKEY=YES	≠ 0	= YES	= YES	Key, Data
	≠ 0	= YES	Not specified	Key, Data
	≠ 0	Not specified	= YES	Key, Data
	≠ 0	Not specified	Not specified	Key, Data

The BLKSIZE and RECSIZE parameters must be specified such that there are no more than 255 records in a block for FIXBLK files defined by the DTFSD or DTFNI declarative macroinstruction. If this maximum blocking factor is exceeded, an attempt to open the file results in control being returned to your error routine (or inline if none was specified) and bit 2 (invalid DTF) and bit 4 (error found in open) set in byte 0 of filenameC.

Keyword Parameter BLKSIZE:

BLKSIZE=n

Specifies the length of the I/O area, where *n* is the maximum size of the block in bytes. If the records in the file or partition are variable-length, *n* must include four bytes for the BDW. If the KEYLEN keyword parameter is specified, you must also include in *n* the specified length of the key.

15.6.4. Address for Routine on End-of-Input File or Partition (EOFADDR)

You must supply the address of the routine you code to handle end-of-data processing for input files processed sequentially by specifying the EOFADDR keyword parameter. This keyword is required for input files defined by the DTFSD declarative macro and for sequentially processed input files defined by the DTFNI macro. You may not need it for a sequentially processed partition defined by a DPCA macro unless you have special end-of-data requirements for that partition: if you do not specify the EOFADDR keyword parameter in a DPCA declarative macro, data management will transfer control, on sensing end of data, to the address specified by the EOFADDR keyword parameter of the parent DTFNI macro.

In addition to performing normal file termination procedures by using the CLOSE imperative macro in your end-of-file or end-of-partition routine (15.7.2), you may optionally extend your file or partition beyond the logical end-of-file; this option is open to you only for sequentially processed input files in the update mode (that is, you have specified UPDATE=YES in the DTFSD or DTFNI declarative macro). The details on this method of extension are developed under the PUT imperative macro (15.7.9).

Keyword Parameter EOFADDR:

EOFADDR=symbol

Specifies the address of a routine you have coded to handle end-of-data for a DTFSD input file or a sequentially processed DTFNI input file, where *symbol* is the symbolic address to which data management transfers control on sensing end of data. Required for DTFSD input files and for sequentially processed DTFNI input files.

15.6.5. Handling Parity Errors on Sequential Disk Files (ERROPT)

Your may use the ERROPT keyword parameter to specify action data management is to take when it is informed of a parity error from which the physical IOCS has tried unsuccessfully to recover. Its use is optional for sequentially processed input or output files defined by the DTFSD or DTFNI declarative macros. It is not supported by the DTFDA macro.

Keyword Parameter ERROPT:

ERROPT=IGNORE

Specifies that data management is to make the block or record available to you in the I/O area as if no parity error had occurred.

ERROPT=SKIP

Specifies that data management is to bypass or skip over an input block or logical record, which it does not make available to you for processing. For output records, data management ignores the block or record as if it were written correctly.

If you omit the ERROPT keyword parameter, on detecting a parity error, data management transfers control to your error-handling routine if you have specified one; otherwise control returns to you inline.

15.6.6. Error Processing (ERROR)

You may have data management transfer control, on detecting an error or exception condition, to a special error processing routine you have defined by the ERROR keyword parameter. The ERROR keyword parameter may be used with input or output files defined by the DTFSD, DTFDA, or DTFNI declarative macros. You do not specify it in a DPCA declarative: on detecting an error or exception condition while processing a partition, data management transfers control to the address specified by the ERROR keyword parameter of the DTFNI macro defining the file to which the partition belongs.

An exception condition, as distinguished from a hardware or detectable logic error, is not necessarily an unforeseen result in processing your file. It may simply signal the completion of I/O or an anticipated end of data, cylinder, track, or volume condition; on the other hand, it may also indicate that the expected record was not found.

Before data management transfers control to your error handling routine, it displays and/or logs an error message. Data management error messages are documented in the system messages operator/programmer reference, UP-8076 (current version) and briefly described in Appendix B.

When your error routine receives control, data management will have already made other information available to you in certain fields of the DTF file table. One field, which your error routine should access dynamically to take appropriate action, is *filenameC*. This field of the DTF contains four contiguous bytes; data management sets certain bits of these to binary 1 as flags to indicate *specific* error conditions; refer to Appendix B (Table B—3) for the significance of the flags. You may address this field in your program by concatenating the character C to your 7-character logical file name.

Another field in the DTF file table, in which data management informs you of the *general* error/status condition, is more useful to examine in debugging a program than to access dyamically in your error routine. This field, designated *filenameE* and always decimal byte 56 in the DTF, is loaded by data management with a hexadecimal error message code: the numeric component of the numbered data management error message to which it corresponds. Note these in the leftmost column in Table B—1. *FilenameE* can be quickly located by the tag generated in the expansion of your DTF declarative macro.

You should realize that not all of the flags set in *filenameC* represent error conditions causing transfer of control to your error routine. The two exceptions in the nonindexed disk file processor system are:

- *last block on track accessed* (byte 0, bit 0); and
- *I/O completed* (byte 1, bit 0).

These two do not represent errors; they signal conditions you might expect to use in the normal course of processing to determine the need to branch in your program. If you use either of these flags in this way, your must test for them *inline*: you will miss your cue if you test only in your error routine.

It is your responsibility to interrogate the error/status codes and take appropriate action. If you choose to continue processing, however, it is useful to remember that data management provides you an inline return address in general register 14; the inline return is to the instruction in your program next following the imperative macro that initiated the transfer of control to your error routine.

If you do not specify the optional ERROR keyword parameter in your DTF, data management returns control to you inline, when an error is detected, to the instruction next following the imperative macro. In this situation, of course, it is up to you to interrogate error/status codes inline and to take appropriate action inline.

Keyword Parameter ERROR:

ERROR=symbol
Specifies the address of your error handling routine, to which data management transfers control upon detecting an error condition, where *symbol* is the symbolic address of this routine.

If you omit specification of this optional keyword parameter, data management returns control to your program inline, at the instruction next after the imperative macro that initiated transfer of control to your error routine.

15.6.7. Specifying Field for Return of Relative Disk Address (IDLOC)

When you want data management to return to you the relative disk address (or ID) of a block after you have issued a READ or WRITE imperative macroinstruction, you specify the IDLOC keyword parameter in your DTF. The ID return is made by the WAITF macro you issue following the READ or WRITE macro.

If you issue a READ,ID macro or a WRITE macro with the ID, AFTER, or RZERO positional parameter, data management returns to you the ID of the *next* block in the file or partition.

On the other hand, if you issue a READ,KEY or a WRITE,KEY macro, data management returns the same ID as is supplied in the SEEKADR field.

The *form* in which the ID is stored for you (as a relative record or a relative track address) is governed by how you specify the RELATIVE keyword parameter (15.6.22).

The IDLOC keyword is specified only for files defined by the DTFDA declarative macro or for randomly processed files defined by the DTFNI macro. (When you are processing a *partition* of a DTFNI file, data management uses the IDLOC keyword you specified in the DTFNI macro defining the file to which the partition belongs.)

When you specify the IDLOC keyword parameter, you provide the symbolic address of the field to which data management makes its return. You should remember that OS/3 data management assumes that the size and format of this field are the same as those of the field you have supplied via the SEEKADR keyword parameter, which you are required to specify for DTFDA files and randomly processed DTFNI files. In some situations, you will find it an advantage to make the IDLOC and the SEEKADR fields (15.6.26) physically one and the same, so that data management automatically updates the SEEKADR field for you.

For example, if you are creating an output file with the WRITE,ID macro (15.7.11.4), you must provide data management with the relative disk address, or ID, to which each block is to be written by moving this ID into the SEEKADR field before issuing the macro: the ID is what guides the macro. After successful execution of the WRITE,ID macro, the following WAITF macro (15.7.16) returns to your IDLOC field the ID of the *next* block in physical sequence in your file. If this address is indeed where your next record goes, you move the contents of the IDLOC field to the SEEKADR field — but, if you had made these two fields the same area in main storage, then data management has in effect updated the SEEKADR field for you automatically, and you can issue another WRITE,ID macro.

Consider, on the other hand, the WRITE,AFTER imperative macro (15.7.11.1). This macro does not require you to preload the SEEKADR field to guide it, but relies upon your having positioned your file to where it is to write. Yet, to specify both the IDLOC and the SEEKADR keywords and to make the fields the same simplifies your handling of the end-of-track occurrence. The ID returned by the WAITF macro that follows each WRITE,AFTER macro is (as with the WRITE,ID macro) the relative disk address of the next block. If you have specified *relative track* addressing (with RELATIVE = T), the ID is not automatically incremented to become the address of the first block on the new track unless you have moved each previously returned ID to the SEEKADR field. To avoid this, and to avoid testing inline for setting of the *last-block-on-track-accessed* bit in *filenameC* after each WRITE,AFTER issue and repositioning your file to the head of the next track with a WRITE,RZERO macro (as described in 15.7.11.1), you may rely on the automatic incrementing that data management provides, by making the IDLOC and SEEKADR fields the same area.

On the other hand, if you are updating your file with the READ,ID/WRITE,ID macro combination, you would *not* want to have data management automatically update the SEEKADR field with the ID return for you. This is because the ID returned after the WAITF that follows the READ,ID macro is the address of the *next* block in the file. You would be overwriting that block with the information intended for the current block if you had made the IDLOC and the SEEKADR fields physically one and the same — not a good practice in update mode.

Table 15—5 recapitulates the situations just described.

Table 15—5. Relative Disk Address (ID) Returned after a READ or WRITE Macroinstruction when IDLOC Keyword Is Specified

Imperative Macros	DTF Keyword Parameters	ID Returned by WAITF Macro	Form* of ID Returned, if:	
			RELATIVE=R	RELATIVE=T
READ, KEY	READKEY=YES	ID of the block retrieved	rrrr	tttr
WRITE, KEY	WRITEKEY=YES	Same ID as that of block retrieved		
READ, ID	READID=YES	ID of *next* block in the file	rrrr	tttr
WRITE,ID	WRITEID=YES			
WRITE, AFTER	AFTER=YES			
WRITE, AFTER, EOF	AFTER=YES	None	—	—
WRITE, RZERO	AFTER=YES	None	—	—

*Discontinuous binary, where:

rrrr is the 4-byte relative block number.

ttt is the relative track number.

r is the absolute block number on the relative track.

Keyword Parameter IDLOC:

IDLOC=symbol

Specifies the field to which data management returns the relative disk address (ID) after the execution of a READ or WRITE macro, where *symbol* (label) is the address of the field. Data management assumes that size and format of the field are the same as specified in the SEEKADR keyword parameter (15.6.23). Form in which ID is returned is governed by specification of RELATIVE keyword parameter (15.6.22); the block whose ID is returned is governed by the positional parameter used with the READ or WRITE macro.

15.6.8. Specifying the Factor for Record Interlace (LACE)

Supported for input and output files defined by the DTFSD, DTFDA, and DTFNI macros and for partitions defined by the DPCA declarative, the optional keyword parameter LACE allows you to specify the lace factor you need when you want to take advantage of record interlace operations.

The record interlace feature of OS/3 data management is both a data mapping and a tuning technique for increasing your throughput when you are processing input or output disk files by permitting you to access successive blocks within a predetermined interval of time. If this interval, which depends upon your program, is less then the time the disk takes to complete a turn, you will retrieve more than one block per disk revolution. Record interlace thus reduces the effect of rotational delay on your overall disk processing time; you benefit most when you are processing files or partitions sequentially and least when you are processing randomly.

This does not mean that the LACE keyword has no role in processing a direct access DTFDA or DTFNI file; the record interlace technique will never degrade random processing of such a file and is of use whenever sequential operations against the file are significant. For example, if you create a direct access file by a substantial sequential loading operation, using record interlace will make for a more efficient creation program. A massive update operation, using the READ,ID/WRITE,ID combination, might also be a recurring program in your application; this, too, could make advantageous use of record interlace. A third example of a use of record interlace that makes sense in a file organized for direct access is with a report-generating program in which random access is used to one or more points in the file, with sequential processing required thereafter for the reports.

You must first physically arrange your blocks on disk using record interlace before you can achieve the increased processing speed that the technique offers you when you are reading from a laced input file; on the other hand, you are able to write your output file more rapidly by using record interlace. It is important to remember that, once you create a file with interlace, this physical characteristic remains with the file, and it must always thereafter be accessed by programs that specify the same lace factor you used to create it.

Figure 15—2 shows how record interlace works to your advantage. Assume that your input file contains ten 1024-byte blocks per track. If these are located on disk in physical sequential order, as shown in the lefthand portion of the figure, you would need 10 disk revolutions to retrieve all 10 blocks sequentially. (If the disk you are using has a rotation speed of 21.4 ms per revolution, for example, accessing the blocks in this way would take 214 ms.)

On the other hand, if you could space your blocks on the track so that the next one to be retrieved arrived under the disk head just as our I/O order to retrieve it took effect, you could save time — possibly enough time to retrieve more than one block per disk revolution. The physical interval between your blocks would need to be little more than the distance the disk rotates during the period of time it takes you to process each block and to issue a new I/O order. The components of this time slice are your processing overhead and the data management overhead involved in issuing your I/O order; your processing time overhead depends primarily on what your program does. When you use the record interlace technique, data management uses SAT (through which it physically accesses each disk subsystem) to establish the physical interval between your blocks and to arrange them on the track so that as many may be retrieved as your program is capable of handling before the next access time comes.

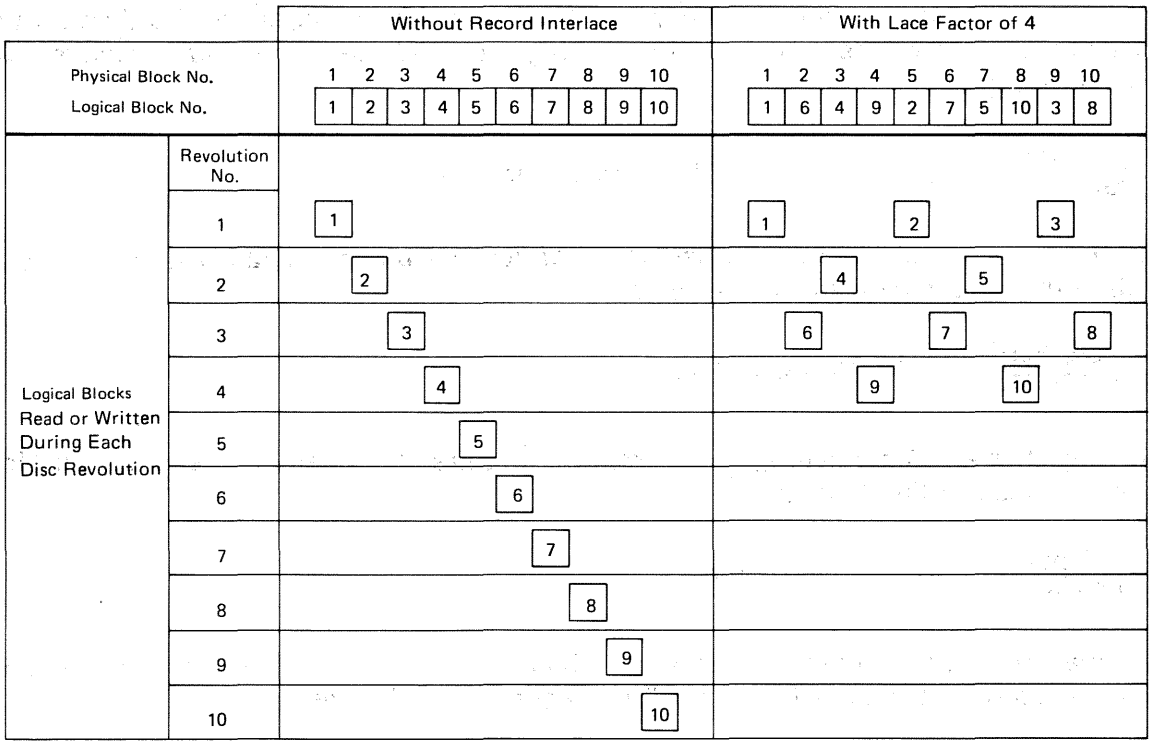


Figure 15—2. Reading a Sequential Disk File With and Without Record Interlace

The righthand portion of Figure 15—2 shows an arrangement of the same 10 blocks, using an interlace factor of 4. How the factor is derived from the estimated program time frame will be developed in what follows; note at this point that their physical arrangement allows all 10 to be retrieved before the completion of the fourth revolution of the disk — less than 85.6 ms, instead of 214 ms as before. Note also that *three* blocks are skipped between successive accesses; the lace factor is always 1 more than the number of blocks skipped.

To achieve this saving of processing time, you specify the lace factor to data management; calculating it is a simple 2-step procedure. You need only two figures: your block size and an estimate of the time interval between your I/O orders.

The first step is to calculate the *sector time*; this is the number of milliseconds each block requires to pass under the disk head. Simply divide your block size, measured in bytes, by 256 bytes and multiply by a standard factor, 0.535 ms:

$$\frac{\text{user block size}}{256 \text{ bytes}} \times 0.535 \text{ ms} = \textit{calculated sector time}$$

The second step yields the *lace factor*, which you will specify as a decimal integer with the LACE keyword parameter. First, you make an estimate of the time frame your program will need for processing between issuing your I/O orders. Then you divide the time frame by the calculated sector time, add 1, and round high to yield the *lace factor*, which is always a decimal integer:

$$\frac{\textit{time frame ms}}{\textit{calculated sector time} \text{ ms}} + 1 \text{ (rounded high)} = \textit{lace factor}$$

Following the illustrated example, assume that you are using a 1024-byte block size; you calculate the *sector time* in milliseconds:

$$\frac{1024 \text{ bytes}}{256 \text{ bytes}} \times 0.535 \text{ ms} = 2.14 \text{ ms}$$

Estimating that a 6.0 ms average *time frame* is required between accesses to your blocks, you then calculate the *lace factor*:

$$\frac{6.0 \text{ ms}}{2.14 \text{ ms}} = 2.9 + 1 = 3.9$$

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 calculations.

You may estimate the average time frame your program needs to process records between passes by using the GETIME macro of the OS/3 supervisor, described in the supervisor user guide, UP-8075 (current version). Or, arbitrarily specifying successively larger lace factors, and noting the running time for your program with each factor, you may pinpoint the optimum lace factor for this program: the one giving the sharpest decrease in running time, followed by a quick increase when the next higher factor is used.

Note that the average time frame will be different for every program accessing the laced file. If several programs at your installation are to use the same file but have different time frames, the file should be laced with a factor that is the best compromise you can establish, taking into consideration the running times of the programs, the frequency with which they will access the file, and the relative importance of the programs to your installation.

It is important to remember that all programs accessing the same laced file must specify in the DTF the actual lace factor used in constructing it; an erroneous specification will cause data management to reset the LACE specification to the value used when the file was created. The lace factor is recorded in the disk format 2 label (D.3.2), and it is one of the items listed by the system utility (SU) symbiont when you request a VTOC print. (The SU symbiont is documented in the system service programs (SSP) user guide, UP-8062 (current version).)

It may be evident to you that the standard factor 0.535 ms and the 256-byte divisor contained in the first formula do not apply to all disk subsystems supported by OS/3. This is true only in part: these figures actually derive from the operatonal characteristics of the 8416 disk subsystem, which was selected quite arbitrarily to provide one set of standard factors for you to use in this calculation. Nevertheless, you use these same figures for *whatever* disk subsystem contains your file; OS/3 data management, through SAT, will adjust the *lace factor* automatically to the characteristics of the actual device in use. (As you know, you never specify the actual device to data management, but SAT and data management know the device type from OS/3 job control.)

Whenever you specify the KEYLEN keyword parameter (15.6.14), you imply random processing, and the LACE keyword parameter is ignored if you specify both.

Keyword Parameter LACE:

LACE=n

Specifies the *lace factor*, a decimal integer, for data management use in applying the record interlace technique to sequentially processed input or output files defined by the DTFSD, DTFDA, or DTFNI macro, or partitions defined by the DPCA declarative macro and processed sequentially. Is ignored when the KEYLEN keyword parameter is specified.

15.6.9. Specifying Input/Output Buffer (IOAREA1)

Each input or output disk file and partition must have at least one input/output area reserved for its individual use. You define this area by specifying the IOAREA1 keyword parameter, which is required for files defined by the DTFSD, DTFDA, and DTFNI declarative macros and for file partitions defined by the DPCA macro.

You define the length of the I/O area by means of the BLKSIZE keyword parameter (15.6.3); as is true throughout OS/3 data management, the I/O area must be half-word aligned. You must reserve an 8-byte area for data management use immediately preceding the I/O area for output files; these eight bytes are *not* included in the blocksize specification. (See Figure 15—1.)

In order to achieve device independence between sectorized and nonsectorized disks, you should *reserve* areas whose lengths are some multiple of 256 bytes for I/O areas. However, your BLKSIZE specification need not be a multiple of 256 bytes to maintain device independence.

Keyword Parameter IOAREA1:

IOAREA1=symbol
Specifies the location, half-word aligned, of the I/O area; required for files defined by the DTFSD, DTFDA, and DTFNI macros and for file partitions defined by the DPCA macro, where symbol (label) is the address of the I/O area. Length of I/O area is specified by BLKSIZE keyword parameter (15.6.3).

15.6.10. Specifying a Secondary Input/Output Buffer (IOAREA2)

You may improve your processing efficiency by specifying a secondary I/O area for standby processing; this allows you to overlap I/O operations with your record processing. You may optionally specify the IOAREA2 keyword parameter for DTFSD files and sequentially processed files and partitions defined by the DTFNI and DPCA macros; it is not supported by the DTFDA macro.

Keyword Parameter IOAREA2:

IOAREA2=symbol
Specifies the location of an optional secondary I/O area for files defined by the DTFSD macro or sequentially processed files and partitions defined by the DTFNI and DPCA macros, where symbol (label) is the address of the secondary I/O area. If specified, the area is subject to the same considerations as noted for the area defined by the IOAREA1 keyword parameter (15.6.10). The IOAREA2 keyword parameter is not supported for DTFDA files.

15.6.11. Specifying Index Register for Current Data Pointer (IOREG)

When you need an index register to reference current data for your I/O processing, you specify the general register to be used for this purpose with the IOREG keyword parameter. General registers 2 through 12 are always available, but if you have specified the SAVAREA keyword parameter, general register 13 is also available (15.6.25).

The IOREG keyword parameter is not supported for files defined by the DTFDA declarative macro. You should specify it for the following input or output files:

- DTFNI files and partitions processed randomly (using the READ or WRITE imperative macro), when you have specified two I/O buffers (15.6.11).

- DTFNI files and partitions processed sequentially (using the GET or PUT macro) and files defined by the DTFSD macro when you have:
  - specified two I/O areas but no work area; or
  - specified no work area but have blocked records.

Table 15—9 summarizes the use of an index register with the GET macro (15.7.12).

When you do use a work area for sequential processing instead of an I/O area, you specify its address as an operand of each GET or PUT imperative macro you issue, and you indicate to data management that you are using a work area by specifying the WORKA keyword parameter in your DTF. You then do *not* specify the IOREG keyword parameter. (See 15.6.34.)

For input files, data management loads the index register with the address of the next available record or block.

For output files, data management loads the index register with the address of the next available I/O area.

When a file is opened, data management loads the index register with the current buffer address and, for a multipartitioned file defined by the DTFNI macro, sets partition 1 active. If you specify an IOREG keyword parameter for several partitions, the index register for partition 1 is loaded when the file is opened; only when you issue a SETP imperative macro to gain access to a *subsequent* partition does data management load the index register for that partition (15.7.4).

Keyword Parameter IOREG:

IOREG=(r)
Specifies the general register to be used as an index register to reference current data for I/O operations, where *r* is the number of the general register and must be enclosed in parentheses. Registers 2 through 12 are available, and register 13 is also available when you specify the SAVAREA keyword parameter. The IOREG keyword parameter may not be specified for DTFDA files, nor when you specify a work area with the WORKA keyword parameter; it is required for DTFNI files and partitions processed randomly with two I/O areas and for DTFSD files and DTFNI files and partitions processed sequentially when records are blocked or when you use two I/O areas.

15.6.12. Specifying Address of Argument for Key Search (KEYARG)

When you want to search your file for a block with a specific key (if, for example, you are issuing the block-level READ,KEY or WRITE,KEY imperative macro), you must provide data management with the search argument in a field in your program that you specify with the KEYARG keyword parameter. Data management uses the search argument to locate a block with an identical key.

You should also remember to specify the length of the key with the KEYLEN keyword parameter, described next (15.6.13). The READ,KEY and WRITE,KEY imperative macros are described in 15.7.14.2 and 15.7.11.5.

Keyword Parameter KEYARG:

KEYARG=**symbol**

Specifies that a search is to be made for a block having a key identical to a search argument you provide to data management, where symbol (label) is the field in your program containing this argument. The KEYARG keyword parameter is not supported for DTFSD files; it is required for DTFDA files and randomly processed DTFNI files (and partitions defined by the DPCA macro) when READ,KEY or WRITE,KEY imperative macros will be issued. When you specify the KEYARG keyword parameter, you must also specify the length of the key, using the KEYLEN keyword parameter.

15.6.13. Specifying the Length of Block Keys (KEYLEN)

When you are referencing blocks by key, generating new blocks with keys, or reading both the key and data areas of your blocks, you must specify the length of the keys in your file with the KEYLEN keyword parameter.

All keys in a DTFDA file or a single-partitioned DTFNI file must have the same length; however, each partition of a multipartitioned DTFNI file may have its own characteristic key length (which must remain the same for all blocks in the partition); you define this length separately with a KEYLEN keyword parameter in the DPCA declarative macro for each partition that does differ in length of keys from the basic file. The minimum key length for any file or partition is 3 bytes; the maximum is 255 bytes. No byte of any block's key may contain the hexadecimal value FF.

It is important to specify the actual length of the key as it exists in all blocks of your disk file; otherwise, data management either truncates or pads out the key and sets the *wrong length found* flag (bit 5, byte 1) in *filenameC*. (See Appendix B.)

Another important point to remember has to do with sequentially processed DTFNI files: these also may have a key associated with each block of data. (See, for example, Figure 15—1.)

If yours do, you must remember to place the key at the beginning of the data block, before you issue a PUT imperative macro. Your PUT macro will then cause both the key and the data portion of the block to be written out to disk, and your subsequent GET macros will retrieve both the key and data. Data management will, as always, block and deblock your records automatically when you are processing DTFNI files sequentially (that is, via the GET and PUT macros). DTFSD files do not contain keyed blocks; DTFDA files may not be specified as having blocked records.

Keyword Parameter KEYLEN:

KEYLEN=n

Specifies the length of keys in DTFDA and DTFNI files and in file partitions defined by the DPCA macro, where *n* is the number of bytes in the keys. All keys in the same partition or the same nonpartitioned DTFNI or DTFDA file must have the same length; the key length may range from 3 bytes minimum to 255 bytes maximum. No byte of any key may contain the hexadecimal value FF.

The KEYLEN keyword parameter is not supported for DTFSD files.

15.6.14. Specifying Address of Your Label Processing Routine (LABADDR)

As you know from 14.2.4, OS/3 data management gives you the option of having your own user header and trailer labels (UHL and UTL) on your nonindexed disk files. When you need to process your labels, you use the LABADDR keyword parameter to specify the address of your label processing routine.

The LABADDR keyword parameter is supported for DTFSD, DTFDA, and DTFNI files; because user header and trailer labels are supported at the file level and not below, you do not specify the LABADDR keyword with the DPCA declarative macro (by which you define a file partition).

When you choose to use your own headers and trailer labels, it is well to remember that they are written by data management on the first track of *each* volume of a DTFSD file (because only one volume is mounted at a time) and on the first track of the *first* volume only of a DTFDA or DTFNI file (because all volumes of these files are online for processing).

In examining the DTF of a direct access file that contains user labels, do not be confused by the relative block addresses calculated by data management and stored in the DTF. These will appear to be inflated because data management adds to the relative disk address in question the number of data blocks that could have been contained by the user label track if it were used for data. (This is more fully explained in 15.6.24.)

Another important point to remember is that your label processing routine may comprise only BAL instructions and the LBRET imperative macro, described in 15.7.3. You may not issue any other file processing macro in your label routine; there is, for example, no legal way to issue a macro to subsequently list your labels for inspection, via a printer file. They do show up, however, in a disk print of your file taken with the system utility (SU) symbiont if your specification to this utility includes head 00 of the volume, and the last one processed may also be seen in your I/O area in a program dump under the right circumstances.

A third point to keep in mind is that, whenever your LABADDR routine will be processing UTL when you issue the CLOSE imperative macro to the file, you must always specify the TRLBL keyword parameter in the DTF (15.6.28).

Your standard UHLs and UTLs have a simple 80-byte format (14.2.4); this means that your I/O areas must always be at least 80 bytes long when you specify a LABADDR routine to process them.

Keyword Parameter LABADDR:

LABADDR=symbol
Specifies the address of your routine for processing optional standard user header and trailer labels, where symbol (label) is the address. This keyword is supported for DTFSD, DTFDA, and DTFNI files. UHL and UTL are not supported at the partition level; therefore, the LABADDR keyword is not specified with the DPCA declarative macro. Your label processing routine may issue the LBRET imperative macro (15.7.3), but no other. If you will be processing UTL on closing the file, you must also specify the TRLBL keyword parameter in the DTF (15.6.28).

15.6.15. Suppressing a File Lock (LOCK)

For a description of the LOCK keyword parameter, see 11.4.11.

15.6.16. Specifying an Optional Sequential File (OPTION)

When you have a sequentially processed file that your program can sometimes do without — that you anticipate you will not invariably want to process every time you run — you may specify this fact with the OPTION keyword parameter. This keyword may be used for files defined by the DTFSD macro and for sequentially processed DTFNI files. Because all volumes of DTFDA or randomly processed DTFNI files are always online for processing, the OPTION keyword parameter is not used for these.

When you use the OPTION keyword parameter, and data management detects that you have not allocated the file to a device (that is, you have not specified a job control DVC-LFD device assignment set for it), the first GET imperative macro transfers control to your end-of-file routine. This action maintains the continuity of your program, but it is up to you to close the optional file when you receive control (you specify the address of your end-of-file routine, which itself is optional, via the EOFADDR keyword parameter (15.6.4)).

On the other hand, if you have *not* specified the OPTION keyword parameter for a file that you neglect to allocate to a device with job control statements, your EOFADDR routine does *not* receive control. Instead, data management either transfers control to your error routine (if you have coded one and supplied its address via the ERROR keyword parameter) or returns control to you inline. In any case, you may neither create records for the file nor obtain records from it even if it exists on disk. For output files, the PUT mechanism is disabled.

You may not randomly process a file described as optional with this keyword parameter, which is reserved for sequentially processed files. If you issue a direct access file processing macro (READ or WRITE) to a file described by the OPTION keyword parameter, data management will transfer control either to your error routine or to your program inline, at the instruction next after the imperative macro, setting the *invalid macro* flag (byte 0, bit 6) in *filenameC*.

Keyword Parameter OPTION:

OPTION=YES
Specifies that the sequentially processed file defined by this DTFSD or DTFNI macro is an optional file: one that you anticipate will not invariably be present for every program execution. When specified for a file not allocated to a device by job control DVC-LFD device assignment set, transfers control to your EOFADDR routine on first issue of the GET macro. For sequential output files, the PUT mechanism is disabled. When specified for sequential files, issue of a direct access imperative macro (READ or WRITE) causes transfer of control to your error routine or to you inline. The OPTION keyword is not used with DTFDA files.

15.6.17. Specifying Address of Partitions for DTFNI Files (PCA)

You must use the PCA keyword parameter in a DTFNI declarative macro to provide data management with the address of each partition of the file that has a separate identify (that is, each partition that is defined with a DPCA declarative macro — see 15.5.4). You need not use the PCA keyword parameter when the DTFNI file is not partitioned (15.5.3).

Each DTFNI file may comprise seven partitions; descriptive information on the first partition is contained in the DTFNI declarative macro. The second partition through the seventh are described in separate DPCA declarative macros, which are limited to presenting those partition-level aspects in which the partition differs from the basic file. The labels of these DPCA macros are the partition names specified by the corresponding PCA keyword parameters of the DTFNI macro.

Although you may subsequently access DTFNI file partitions in any order, when you are defining an *output* file you must specify the partitions in unbroken sequence in the DTFNI macro: you must specify the first partiton before the second, the second before the third, and so on. When you are defining an *input* file, you specify only those partitions you actually require for processing. (You will subsequently access these separately by issuing a SETP imperative macro, 15.7.4.)

You should not confuse the PCA *keyword parameter* described here with the PCA *declarative macroinstruction* of the OS/3 supervisor. The PCA macro of the supervisor has a role analogous to that of the OS/3 data management DPCA declarative macro; see the supervisor user guide, UP-8075 (current version).

Keyword Parameter PCA:

PCA1=symbol,...,PCA7=symbol
Specifies the address of each partition of a multipartitioned DTFNI file, where *symbol* (label) is the address. *PCA2=symbol,...,PCA7=symbol define the labels of the corresponding DPCA macros. Partition 1 is contained in the table defined by the DTFNI macro, in which PCA1=symbol* is required only for data management use in assigning the label to the partition table defined within the DTF. The PCA keyword is not used for nonpartitioned DTFNI files, nor for files defined by the DTFSD or DTFDA macros. Partitions must be specified in unbroken sequence. The symbolic address is used as the label of the corresponding DPCA declarative macro.

15.6.18. Specifying Issue of a READ,ID Macro (READID)

When you intend to read a direct access file, locating each block by its relative disk address or ID, you will issue the READ,ID form of the READ imperative macro (15.7.14.1). Beforehand, however, you must inform data management that you will be doing so, by specifying the READID keyword parameter in the DTFDA or DTFNI declarative macro. (The READID keyword is not used with files defined by the DTFSD macro.)

Keyword Parameter READID:

READID=YES
Specifies to data management that you will issue a READ,ID imperative macro to the DTFDA or DTFNI file defined by this declarative macro.

The READID keyword is not used for DTFSD files.

15.6.19. Specifying Issue of a READ,KEY Macro (READKEY)

When you intend to read a direct access file, locating each block by a search on key, you will issue the READ,KEY form of the READ imperative macro. Beforehand, you must inform data management that you will be doing so, by specifying the READKEY keyword parameter in the DTFDA or DTFNI declarative macro. The action of the READ,KEY imperative is explained in 15.7.14.2; you will need also to specify the following keyword parameters:

SEEKADR (15.6.23)
KEYARG (15.6.12)
KEYLEN (15.6.13)

Keyword Parameter READKEY:

READKEY=YES
Specifies to data management that you will issue a READ,KEY imperative macro to the DTFDA or DTFNI file defined by this declarative macro (15.7.14.2). This keyword is not used for DTFSD files. The SEEKADR, KEYARG, and KEYLEN keyword parameters are also required.

15.6.20. Specifying Format of Records in Disk Files (RECFORM)

The optional keyword parameter RECFORM is your means of specifying to data management the format of the records in your disk files. It is unnecessary to specify this keyword parameter if your records are fixed-length and unblocked: data management assumes that this is the desired format when you omit the RECFORM keyword from your DTF.

OS/3 data management supports four record formats for nonindexed disk files; Table 15—6 shows which formats are used with which file type. Note that undefined records are not supported.

Table 15—6. Record Formats for Nonindexed Disk Files

Record Format	Declarative Macro				Specification for RECFORM Keyword Parameter
	DTFSD	DTFDA	DTFNI	DPCA	
Fixed-length, unblocked	X	X	X	X	FIXUNB
Fixed-length, blocked	X	—	X	X	FIXBLK
Variable-length, blocked	X	—	X	X	VARBLK
Variable-length, unblocked	X	X	X	X	VARUNB

X = supported
— = unsupported
▒ = assumed if not specified

Keyword Parameter RECFORM:

Optional for all files; specifies format of records. There are four specifications:

RECFORM=FIXUNB
Specifies that records are fixed-length and unblocked. Data management assumes this format if you omit the RECFORM keyword parameter from the DTF.

RECFORM=FIXBLK
Specifies that records are fixed-length and blocked. Not supported for DTFDA files.

A maximum of 255 records per block is allowed. The number of records per block is saved in the format label in a 1-byte field. Thus, if it exceeds 255, the value placed in the 1-byte field will be the actual number of records per block minus modulo 256. For example, if there are 300 records per block, the effective value placed in the format label will be 44.

RECFORM=VARBLK
Specifies that records are variable-length and blocked. Not supported for DTFDA files.

RECFORM=VARUNB
Specifies that records are variable-length and unblocked. Supported for DTFSD, DTFDA, DTFNI, and DPCA declarative macros.

15.6.21. Specifying Size of Records in Blocked Disk Files (RECSIZE)

When you have specified blocked records in your file, fixed-length, you may specify the number of bytes in a logical record to data management with the optional RECSIZE keyword parameter. This keyword may not be specified for files defined by the DTFDA declarative macro because OS/3 DAM does not provide for blocking or deblocking records.

It is not necessary to specify the RECSIZE keyword parameter when your records are fixed-length, unblocked: data management assumes that the record size equals the length of the I/O area that you specify by your BLKSIZE keyword parameter (15.6.3) when your records are fixed-length, unblocked. Nor do you specify the RECSIZE keyword for files containing variable-length records, blocked or unblocked: here, data management expects to find the size of the record in the first two bytes of the RDW at the head of each variable-length record. A look at Figure 15—1 (15.6.3) will make these points clear.

See the description of the BLKSIZE keyword parameter (15.6.3) for information regarding the maximum blocking factor for FIXBLK files.

Keyword Parameter RECSIZE:

RECSIZE=n

Specifies the length of fixed-length logical records in blocked files, where *n* is the number of bytes in the record. Optional for files defined by DTFSD and DTFNI macro and for partitions defined by DPCA declarative macro. Not used for DTFDA files.

Data management assumes that record size equals block size (BLKSIZE keyword parameter) for fixed-length, unblocked records and expects to find record size in RDW of variable-length records.

15.6.22. Specifying the Form for Relative Addressing (RELATIVE)

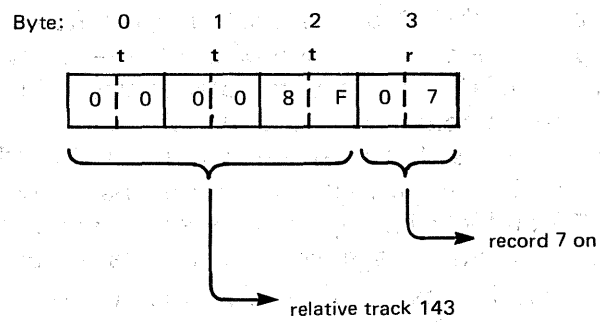
The nonindexed disk file processor system gives you the choice of two forms for specifying the *relative disk address* (ID) of an individual block or record in a direct access file: *relative record or relative track* addressing. You specify the form that is to be used for data management with the RELATIVE keyword parameter of the DTFDA or DTFNI declarative macro. When you specify the relative disk address of a record or block to data management, or when data management returns an ID to you in the course of processing, it is placed in this form on the appropriate 4-byte field in your program. The *absolute* disk identification address* is not used in the nonindexed processor system.

By relative record address, you should understand a hexadecimal number, right-justified in its 4-byte field, that represents the sequential position of the block or record, relative to the beginning of the direct access file or partition. The number of the first record of a file or partition is 1.

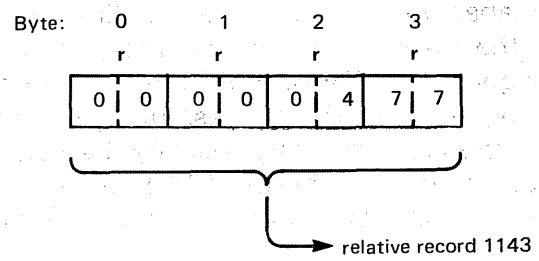
**The absolute disk identification address, an ID used in other data management systems, is a 5-byte address containing the cylinder number, the head number, and the record number that define the physical location of a record on a disk. In OS/3 data management, all disk record addressing in direct access files is relative and uses a 4-byte address in either of the formats described here. OS/3 ISAM uses a 5-byte file-relative address (or record pointer); refer to Sections 10 and 11.*

By relative track address, understand a 4-byte hexadecimal number, the first three bytes of which represent the track number on which the record or block is written, relative to the first track of records in the direct access file or partition. Track number begins with 001. The fourth byte of the relative track address represents the sequential number of the block or record on this relative track; block or record numbering begins with 1.

For example, to specify record 7 on relative track 143 in a direct access file, if you have specified RELATIVE=T in your DTF, this is what you would move into the 4-byte field in your program:



Assume, that this same file is laid out with eight blocks to a track, but that you have specified RELATIVE=R in the DTF. The same record would be the 1143rd in the file, and you would specify its relative disk address this way in the 4-byte field:



Any relative disk addresses that you generate in your program for use with the data management imperative macros designed for random processing must be presented to data management in the appropriate one of these two forms, and you must inform data management as to which form to expect by specifying the RELATIVE keyword parameter.

You may, at this point, be asking yourself when a relative disk address is the ID of a *block,* and when it is the ID of a *record*; you may also want to know how to decide between *relative track* or *relative record* addressing. These questions are taken up in the next few paragraphs, but a glance back at Figure 14—4 will suggest part of the answer: that a record has an ID only in the same sense that a record has a key. When it is in unblocked record format (fixed or variable in length), a record actually exists on disk in a block as shown in the figure, and the ID of the block and its key can both be *considered* to represent the only record in the block.

For example, when you are referring to your direct access records by key, using the block-level READ,KEY imperative macro for retrieving data from your file, you specify to data management the key of the block that contains the record or records you are after, placing it in the KEYARG field of your program (15.6.13). You must also specify a relative disk address to indicate to data management where in the file to start its search for the desired block, by moving it into the SEEKADR field of your program (15.6.24). This then, is the ID of the first block to be tested for a key that matches the content of your KEYARG field. You should consider it the ID of a record only if this record is in the unblocked format and the only record in the block — which it must be in a DTFDA file.

Moreover, if you want to save the ID of the block that the READ,KEY macro retrieves for you (to use it, for example, in later processing of the file — perhaps in another program), then you must specify the IDLOC keyword parameter in your DTF, thus providing the label of a field to which data management makes a return: the relative disk address of the keyed block it has just retrieved. This is discussed further under the IDLOC keyword parameter (15.6.7), where Table 15—5 summarizes the ID returns made after execution of the various forms of the READ and WRITE imperative macros. Once the READ,KEY macro has read the block you want into main storage, you may access the desired record by your own deblocking code or by successive issues of the GET macro. For further details, refer to the READ,KEY macro description, 15.7.14.2.

When you are retrieving data from your file with the READ,ID imperative macro, you must specify the ID of the *record* you want retrieved by placing this, in the form you have specified with the RELATIVE keyword parameter, in the field defined by your SEEKADR keyword. Although data management reads in the entire block, including the key if the block has one, it points to the specified record by loading its displacement within the block in a field of your DTF designated as *filenameD*. The ID returned by data management to the IDLOC field of your program after the successful execution of the READ,ID macro is the relative disk address of the *block* that is physically the next in your file. Again, this ID is in the form you have specified with the RELATIVE keyword parameter; further details are documented under the READ,ID macro description, 15.7.14.1.

The advantage of specifying the *relative track* form of record addressing in DTFDA files (RELATIVE=T) is that you can treat each track of your data as if it were a partition or a subfile (only DTFNI files may actually comprise partitions or subfiles (15.4, 15.5.3)). On the other hand, such use requires that you keep tight control over your data, knowing what goes where in each case, without the help of the DTFNI and DPCA file tables or the special imperative macros (NOTE, POINT, POINTS, SETP, and SETS) that cannot be issued to DTFDA files.

An advantage of specifying *relative record* addressing (RELATIVE=R) is that you can stand off further removed from your file, without as much concern for its layout on disk, remaining independent of all actual disk locations.

The RELATIVE keyword is not specified for the DPCA declarative macro; when you are randomly processing a partition of a DTFNI file, data management uses the specification of the RELATIVE keyword you made in the DTFNI macro defining the file to which the partition belongs.

Keyword Parameter RELATIVE:

For DTFDA files and randomly processed files defined by the DTFNI macro, specifies the method of relative addressing. Not used for DTFSD files. For direct access file partitions, data management uses the specification in the DTFNI macro; the RELATIVE keyword is therefore not specifiable in the DPCA macro.

RELATIVE=R

Specifies *relative record* addressing, in the form:

rrrr

where:

rrrr

Is a hexadecimal number, right-justified in a 4-byte field, that represents the number of the block or record to be accessed, relative to the first block or record in the file or partition. The number of the first block or record in a file or partition is 1.

RELATIVE=T

Specifies *relative track* addressing, in the form:

tttr

where:

ttt

Is a 3-byte hexadecimal number, relative to the first track in the file or partition, of the block on which the record or block occurs. Track numbering begins with 001.

r

Is the sequential number of the record or block on this relative track; record numbering begins with 1.

15.6.23. Specifying a Save Area for Contents of General Registers (SAVAREA)

Before you issue an imperative macro for processing any OS/3 data management file, you should generally first load general register 13 with the address of a 72-byte labeled save area — always aligned to a full-word boundary — in which data management will expect to save the contents of your registers. One purpose of the SAVAREA keyword parameter is to make things easier for you if you are converting a program, written for use under some other data management system, in which you have already used register 13 for some other purpose.

When you are converting such a program to run under OS/3 data management, you need not recode it to revise your use of register 13. You need only add to it a 72-byte labeled save area, aligned as described, and specify its address with the SAVAREA keyword parameter in the DTF for each file your program will process. (Although you will need to specify this keyword in each DTF, you need only *one* register save area for your program.)

In writing a new program using OS/3 data management, you must make one choice or the other: load register 13 with the same area address before issuing macros, or specify the SAVAREA keyword. One advantage of using the SAVAREA keyword is that register 13 then becomes available (along with register 2 through 12) for use in your program: as the IOREG or VARBLD register, for example.

When it does not encounter the SAVAREA keyword in your DTF, data management will always assume that, before issuing an imperative macro to the file, you have preloaded register 13 with the address of a 72-byte save area, aligned on a full-word boundary.

Keyword Parameter SAVAREA:

SAVAREA=symbol
Specifies the address of a 72-byte labeled save area for the contents of general registers, full-word aligned, where *symbol* (label) is the address. If used, must be specified in the DTF for each file your program will process; however, only one such save area is required per program. Not supported by DPCA declarative macro, as SAVAREA is a file-level parameter.

15.6.24. Specifying Relative Disk Address for Random Processing (SEEKADR)

The SEEKADR keyword parameter is mandatory for random processing; you must specify it for DTFDA files and for randomly processed files defined by the DTFNI declarative macro. Its purpose is to specify the 4-byte field in your program into which you load the relative disk address you will use for directly accessing a block or updating it, for initiating a key search, and for controlling the movement of the disk head. There are therefore three basic ways you must use the SEEKADR keyword parameter: with the READ,ID and WRITE,ID macros; with the READ,KEY and WRITE,KEY macros; and with the CNTRL and WRITE,RZERO/WRITE,AFTER macros.

When you are issuing READ,ID and WRITE,ID macros to directly access blocks in a randomly processed file to retrieve and update them, you must load into the SEEKADR field of your program the relative disk address (ID) of the current block.

On the other hand, when you are referencing your keyed blocks by key and therefore using the READ,KEY and WRITE,KEY macros, the address you must load into the SEEKADR field is the relative disk address at which data management is to start the key search.

The third required use of the SEEKADR keyword parameter is related to your uses of the WRITE,RZERO macro and the CNTRL macro. When you use the WRITE,RZERO and WRITE,AFTER macros to select and initialize a disk track, you must load, into the SEEKADR field of your program, the relative disk address to which data management is to reposition your output file for subsequent processing, initialized so as to write the first block on the new track with the WRITE,AFTER macro (15.7.11.2). Similarly, when you use the CNTRL macro to control disk head movement (either to return to the current track or to seek another track, 15.7.15), you must place the relative track address you want in the field specified by the SEEKADR keyword parameter.

In certain circumstances (for example, when you want to use the ID returned by the WAITF macro that follows each READ or WRITE macro issued to automatically update the SEEKADR field for you), you may specify that the IDLOC and SEEKADR fields are physically one and the same area in main storage. Refer to 15.6.7, where the means and rationale for doing so are discussed in some detail.

Data management does not require a special alignment of the IDLOC or SEEKADR field (as it does for the I/O buffer and certain other areas). However, you may well need to align the SEEKADR field on a specific boundary if you are performing certain operations on it. For example, if you are controlling movement through your file by incrementing the relative track number (having specified RELATIVE=T), using, say an *add immediate* (AI) instruction, you should align the 4-byte SEEKADR field on an *odd* boundary so that the increment is added to byte 2 by this half-word-oriented instruction.

Consider the following example, which presupposes relative track addressing in a direct access file named DAMFLE. You are stepping through the file, updating blocks on every third track. This is the content of your IDLOC field at a certain point, representing record 7 on relative track 145:

Byte:	0	1	2	3
	t	t	t	r
	0 0	0 0	9 1	0 7

You have aligned your SEEKADR field (the label of which is SKADR) and defined it as a 4-byte constant with the following pair of BAL statements in your program, and you are moving the IDLOC contents to this field before each incrementation.

```
   LABEL     ΔOPERATIONΔ        OPERAND                Δ
   1         10        16
1. ALIGNODD  DC        CL3'XXX'
2. SKADR     DC        CL4'0000'
```

1. Assuming that the alignment of previously defined storage areas leaves the location counter at an even boundary, this statement forces odd alignment. No label is required, of course, except to document the point.

2. The label SKADR (which you have equated to the SEEKADR keyword parameter in the DTF defining the direct access file DAMFLE) is defined as a 4-byte character (not full-word) constant.

At the appropriate point in your executable code, you issue the following pair of instructions, to increment relative track number by 2 and to write record 7 on relative track 147.

```
LABEL     ΔOPERATIONΔ     OPERAND                    Δ
1         10        16
INCREM    AI        SKADR+1,2
          WRITE     DAMFLE,ID
```

The half word addressed by the AI instruction comprises bytes 1 and 2 of the SEEKADR field, SKADR. Byte 2 contains the least significant byte of the relative track number, *ttt*.

Other operations may require other alignment — but whatever you do to the contents of the SEEKADR field, remember that they must be in fixed-point binary format before you issue an imperative macro that uses the seek address:

```
WRITE,RZERO
WRITE,ID
READ,ID
READ,KEY
CNTRL
```

Remember also that the relative disk address must never have a negative or zero value.

Before leaving the subject of relative track addressing, you should note that, when your DTFDA file contains optional user labels, data management reserves the entire first track on the first volume of your file for them. However, you must not include this user label track in your relative track count. The first of your data blocks is written on the first track *after* the label track, and this, then, is relative track 001.

The same caution applies also to the DTFNI file containing user labels, but these files may also contain subfiles in each partition. When this is the case, data management reserves the first track after the user label track (or the first track of the file, in the absence of user labels) to maintain its subfile tables. Relative track 001, on which the first of your data lies, may then be actually the third track of the extent; it is still the first data track of your *file*.

There are two reasons for keeping these points in mind. One is that, when you are using the system utility (SU) symbiont to obtain a print of a specific part of your disk file, and are specifying certain heads and cylinders to be listed, you must include the user label and subfile table tracks in your reckoning. There is no way to specify a *relative* track to the SU symbiont (which is documented in the system service program (SSP) user guide, UP-8062 (current version)).

Another reason to remember the user label and subfile table tracks in your calculations is that the length of these tracks (in terms of the number of your data blocks they could contain if used for data) is included by data management in its calculation of the current relative block and certain other relative addresses that it maintains in the DTF file tables. You will eventually learn to read these fields in the DTF you see in a program dump, for they are often useful in debugging, and you will need to be prepared for an otherwise mystifying discrepancy. For example, if each track in your data file can hold 10 blocks — and you have both user labels and subfile tables on disk — each relative block address contained in the DTF will appear to have been inflated, being 20 blocks higher than you expected.

The *form* in which you load the relative disc address into the SEEKADR field is governed by your specification of the RELATIVE keyword parameter; see 15.6.22, where relative track and relative record addressing are discussed in detail. In no case may the ID be negative or zero.

Keyword Parameter SEEKADR:

SEEKADR=symbol
Specifies the location in your program into which you load the relative disk address for use in processing direct access files with the READ,ID; READ,KEY; WRITE,ID; WRITE,RZERO; and CNTRL imperative macros. Required for DTFDA files and randomly processed files defined by the DTFNI macro. Form in which address is loaded is governed by your specification of the RELATIVE keyword parameter. Relative disk address may not be negative or zero.

15.6.25. Assigning Initial Disc Space to a File Partition (SIZE)

When you are defining each partition of a DTFNI file, you may specify the percentage of the total file allocation that data management is to initially assign to the partitions, by using the SIZE keyword parameter.

You indicate the initial disk space you require for each partition by specifying the SIZE keyword parameter in the DTFNI or DPCA declarative macro.

It is not necessary to specify the SIZE keyword parameter in the DTFNI or DPCA macros if you want only 1% of the total file allocation to be assigned initially to the first partition and to those partitions that you do not specify other percentages for.

As you know, you specify the *total* file space allocation to OS/3 job control when you initially allocate the file, using the fourth and fifth positional parameters on the EXT job control statement of your device assignment set. For subsequent automatic dynamic extension of a *partition*, you will use the UOS keyword parameter (15.6.30).

Keyword Parameter SIZE:

SIZE=n
Specifies the percentage of total file allocation to be initially assigned by data management to the partition being defined by this DTFNI or DPCA declarative macro. Not used with nonpartitioned DTFNI files, nor with DTFSD or DTFDA files.

If omitted from DTFNI macro, data management assumes that SIZE=1 had been specified and assigns 1 percent. If omitted from DPCA macro, data management makes a 1 percent allocation.

15.6.26. Extending Key Search to Multiple Track (SRCHM)

When you issue a READ,KEY imperative macro to a DTFDA or randomly processed DTFNI file, the search continues until the first block headed by a key that matches the content of your KEYARG field is found, or the end of the current track is reached, whichever occurs first. To search *beyond* the end of the current track to the end of the cylinder, you may specify the SRCHM keyword parameter in the DTF.

Keyword Parameter SRCHM:

SRCHM=YES
Specifies that a search key issued to a DTFDA or randomly processed DTFNI file (READ,KEY) is to be extended beyond the current track to the end of the cylinder.

15.6.27. Specifying Support of Subfiles in a Partition (SUBFILE)

You may further subdivide each of the seven optional partitions of a DTFNI file into as many as 71 subfiles. These subfiles, which you will access serially for processing within the partition, via the SETS imperative macro (15.7.5), maintain the same characteristics as the partition. (You select the appropriate partition with the SETP macro (15.7.4).)

Data management reserves one track of the first volume of the file on which to maintain subfile tables when subfiles are to be supported; to indicate to data management that it is to do so, you must specify the SUBFILE keyword parameter in the declarative macro that defines each separate partition containing subfiles: the DTFNI macro for the first partition, and the appropriate DPCA macros for the succeeding partitions.

If you file does not contain optional user labels, data management writes the subfile tables on the first track; when you do have UHL/UTL, data management writes these on the first track, and the subfile tables go on the second track. Refer to 15.6.24 for a discussion of the effect the presence of these tracks have on positioning yourself in your file. Refer to 15.7.5 for discussion of the contents of the subfile.

Keyword Parameter SUBFILE:

SUBFILE=YES
Specifies that data management is to support subfiles in the partition defined by this DTFNI or DPCA declarative macro. A maximum of 71 subfiles may be established in each partiton; you access these serially by issuing a SETS imperative macro to the file, having previously selected the correct partition with a SETP imperative macro.

15.6.28. Specifying Processing of User Trailer Labels (TRLBL)

As you recall, you may have optional standard user trailer labels (UTLs) in your nonindexed disk files. These are processed by your label routine (LABADDR keyword parameter, 15.6.15) and written by data management on the user label track when you issue the CLOSE imperative macro to the file. When you want to process your UTL on closing the file, you must specify the TRLBL keyword parameter beforehand in the DTF for the file; naturally, you must also provide data management with the address of your label processing routine by specifying the LABADDR keyword parameter. If you omit the TRLBL keyword parameter, your LABADDR routine does not receive control at file close, and your UTLs, consequently, are never processed.

Keyword Parameter TRLBL:

TRLBL=YES
Specifies that you will process UTL when you issue the CLOSE imperative macro to the DTFSD, DTFDA, or DTFNI file defined by this DTF. You must also specify the LABADDR keyword parameter.

15.6.29. Defining the Type of File (TYPEFLE)

One of the most significant keyword parameters, used for DTFSD, DTFDA, and DTFNI files, is the TYPEFLE keyword parameter. It is important to you because with it you specify not only what type of optional header/trailer *label* processing you will perform on the file, but also what basic processing you will be performing on the data itself. It is also important to note that, having defined a file for one mode of processing, you are generally restricted to that mode until you explicitly provide for a change in one of the ways pointed out in the following discussion.

This is the format of the TYPEFLE keyword parameter:

```
TYPEFLE= {INPUT  }
         {OUTPUT }
         {INOUT  }
```

If you omit specifying this keyword, data management assumes that TYPEFLE=INPUT has been specified.

The effect of the TYPEFLE keyword on header/trailer label processing is simply summarized: TYPEFLE=INPUT specifies that data management will read header/trailer labels for the file; TYPEFLE=OUTPUT specifies that it will write header/trailer labels. In both cases, these actions include reading, checking, and writing your optional UHL and UTL, which you generate and update with your LABADDR label processing routine. If you specify TYPEFLE=INPUT and the LABADDR and TRLBL keyword parameters, data management assumes UHL and UTL exist. You may not specify TYPEFLE=INOUT for DTFDA files, but this specification for DTFSD or DTFNI files is what you use for label updating.

The TYPEFLE keyword parameter controls only the mode of *label* processing to be performed on files defined by the DTFDA macro; for DTFSD files and sequentially processed DTFNI files, however, it also constrains your processing of the data records in the file. When you specify TYPEFLE=INPUT for one of these, you define a file that is to be *read only*. You may not issue an output function to it (that is, the PUT imperative macro) unless you have *also* specified UPDATE=YES in the DTF. (The UPDATE keyword parameter is described in 15.6.31.) Similarly, when you specify TYPEFLE=OUTPUT, you define a file that is to be *written*, and you may not issue an input function (the GET imperative macro) to the file. Specifying TYPEFLE=INOUT defines a file that you may use either as an input or an output file, issuing the GET or PUT macro as required.

After you have used a file as an output type, you must close the file and change its type to input before you reopen it. You alter the file type by issuing the SETF imperative macro to the file; this procedure is developed in 15.7.8. (The OPEN macro is described in 15.7.1; the CLOSE imperative in 15.7.2.)

Keyword Parameter TYPEFLE:

TYPEFLE=INPUT
Specifies a read-only file; used with DTFSD, DTFDA, and DTFNI macros. Data management will read and check standard labels for this file. You may not issue an output function to this file unless you have also specified UPDATE=YES in the DTF. Data management assumes you have specified TYPEFLE=INPUT when you omit the TYPEFLE keyword parameter.

TYPEFLE=OUTPUT
Specifies a file that is to be written; used with the DTFSD, DTFDA, and DTFNI macros. Data management will write standard labels for this file. For DTFDA files, this keyword controls only the mode of *label* processing to be employed; for DTFSD files and sequentially processed DTFNI files, it also controls the mode of record processing you may use within the file. You may not issue an input function to this file unless you close it, reset its file processing direction to *input* with the SETF imperative macro, and reopen the file.

TYPEFLE=INOUT
Specifies a file that you may use for either input or output. Used with DTFSD and DTFNI files; not used with DTFDA files.

15.6.30. Specifying Dynamic Extension of a File Partition (UOS)

You use the SIZE keyword parameter (15.6.25) to assign *initial* disc space to a partition of a DTFNI file, but when you anticipate a need to *extend* a partition dynamically, you use the UOS keyword parameter. This keyword is specified only with the DPCA macro or with a DTFNI macro which defines the first (or only) partition of a nonindexed file.

With the UOS keyword, you specify to data management what percentage of your secondary allocation of disk space to the *file* it is to suballocate each time the *partition* requires additional space. From your acquaintance with OS/3 job control, recall that you specify the number of cylinders or blocks that is to constitute the secondary allocation of disk space to a file with the third positional parameter on the EXT job control statement of your file's device assignment set. (To review this statement in detail, you should refer to the job control user guide, UP-8065 (current version).)

When you have specified the UOS keyword parameter for the partition (and have not specified a *zero* increment in your EXT job control statement), and the partition requires more space, data management will automatically suballocate to it the number of additional tracks that is equivalent to the amount of storage specified by the UOS keyword. This amount is designated the *unit of store*; the same amount is used each time the partition needs extension. Data management keeps a record of suballocation information developed for all partitions in the format 2 label associated with the file (Appendix D).

Data management learns of the need to extend a file partiton each time one of your output imperative macros (PUT or WRITE) references a block that lies beyond the current maximum relative block address data management maintains in the PCA table for the partition. Data management acts automatically to extend the partition, but there are several points you should keep in mind.

One point is that you must have specified the UOS keyword parameter in the first place; if you omit it, no extension can be made. A second point is that file partitions will not be extended beyond the volumes on which the file resides. Furthermore, if, after extending your partition by one unit of store, data management finds that the requested block still lies beyond the new maximum relative block address, it sets the *invalid ID* flag (byte 0, bit 1) in *filenameC* and transfers control to your error routine (or to you inline if you have no error routine). Data management takes the same action if there is no space available on the disk to extend the partition or if you have not specified the UOS keyword. Finally, a unit of store specification greater than 100% is not valid.

Keyword Parameter UOS:

UOS=n

Specifies, as the unit of store, the percentage of secondary disk storage allocation for the file that data management is to suballocate to the partition being defined each time it requires more space. The value of n, which is the specified percentage, may not exceed 100. Secondary storage allocation is specified in the EXT job control statement in the device assignment set for the file.

Used with DPCA declarative macro or with DTFNI macro defining first (or only) partition of a nonindexed file. Not used with DTFDA or DTFSD macro.

If omitted, or if a zero secondary storage allocation is specified in EXT job control statement, no extension will be made.

15.6.31. Specifying Update Processing Mode for Sequential Files (UPDATE)

The UPDATE keyword parameters is used with the DTFSD and DTFNI macros.

You will recall from the discussion of the TYPEFLE keyword parameter that when you have specified TYPEFLE=INPUT, you define a read-only file and may not issue an output imperative macro to it unless you have also specified UPDATE=YES (15.6.29).

When, therefore, you have an input file defined by a DTFSD macro, or an input file defined by the DTFNI macro that is to be processed sequentially and you want to update your data records, you specify UPDATE=YES to make this possible. Only then may you retrieve your records with the GET imperative macro and, modifying them if you need to, rewrite them to disc with the PUT macro. (The UPDATE=YES specification is also accepted in your DTF for sequentially processed files you have defined as TYPEFLE=INOUT, although you do not need to use it for this 2-way type of file.) Another point worth remembering is that the UPDATE keyword parameter, unlike the TYPEFLE keyword, has nothing to do with your mode of label processing; its use affects only your mode of data record processing in sequential input files.

Keyword Parameter UPDATE:

UPDATE=YES
Used only with DTFSD macro or DTFNI macro defining sequentially processed files and only when you have specified TYPEFLE=INPUT or TYPEFLE=INOUT for these files. When used, specifies that sequential output function (PUT macro) may be issued to update data records in file. Unrelated to label processing.

If omitted from DTF for a sequentially processed input file (TYPEFLE=INPUT), you may not issue an output function to the file.

15.6.32. Specifying Register for Residual Space, Variable Records (VARBLD)

When you are outputting variable-length, blocked records to a sequentially processed disk file, and you are building these in an I/O area without a work area, you must specify the VARBLD keyword parameter.

Data management uses the general register you specify with this keyword to inform you of the number of bytes of residual space remaining in the I/O area after the execution of each PUT macroinstruction you issue. Before you issue your next PUT macro, you must test whether there is enough space left to accommodate the next record. If there is not, you must issue a TRUNC macro to write out the current block. (These procedures are detailed in 15.7.9.4 and 15.7.10.)

The VARBLD keyword parameter is not supported by the DTFDA macro because the blocked record formats may not be specified for DTFDA files; you may specify the keyword for sequentially processed output files or partitions defined by the DTFSD, DTFNI, or DPCA declarative macros.

Keyword Parameter VARBLD:

VARBLD=(r)
Specifies a general register into which data management loads the number of bytes remaining in the I/O area after each execution of a PUT macroinstruction to a sequentially processed output file containing blocked, variable-length records, where *r* is the number of the register and must be enclosed in parentheses. Supported for DTFSD, DTFNI, and DPCA macros; not supported for DTFDA macro. Value of *r* may range from 2 through 12, but register 13 may also be used if SAVAREA keyword has also been specified.

When you are building variable-length blocked records in an I/O area for output to a sequentially processed file, without a work area, you must access the VARBLD register to test whether enough space remains in the area for the next record before issuing your next PUT macro. You issue a TRUNC macro when it does not. See 15.7.9.4 and 15.7.10.

15.6.33. Specifying Parity Check Verification of Output (VERIFY)

When you want data management to make a parity check of your data records or blocks after it has written each of them to disk, you must specify the optional VERIFY keyword parameter. Verification necessarily increases execution time for the output commands involved (PUT or WRITE macroinstruction). If a parity check is conducted and reveals an error, data management normally sets the *output parity check* error flag (byte 2, bit 2) in *filenameC* and transfers control to your error routine, if you have specified one, or to you inline (Appendix B).

You may specify the VERIFY keyword parameter with the DTFSD, DTFDA, and DTFNI macros; it is not supported by the DPCA macro.

Keyword Parameter VERIFY:

VERIFY=YES
Specifies that data management is to conduct a parity check of output blocks or records after writing them to disk. Parity check verification necessarily increases execution time for PUT or WRITE macro. Optional for DTFSD, DTFDA, and DTFNI files; not supported for DPCA macro.

If omitted, no verification is performed; however, data management may detect an output parity check error by other means. You may direct data management to take certain actions with the ERROPT keyword parameter (15.6.5).

15.6.34. Specifying Sequential Processing in a Work Area (WORKA)

When you are going to process input or output records sequentially in a work area rather than in the I/O area, you indicate this to data management by specifying the WORKA keyword parameter in the DTF. This keyword is supported for the DTFSD declarative macro and for sequentially processed files or partitions defined by the DTFSD declarative macro and for sequentially processed files or partitions defined by the DTFNI or DPCA macro; it is not specified for the DTFDA macro.

You specify the address of the work area in the second positional parameter of each PUT or GET macro you issue to the file (15.7.9 and 15.7.12). When you use a work area and therefore specify the WORKA keyword, you must not specify the IOREG keyword parameter in your DTF (15.6.11).

Keyword Parameter WORKA:

WORKA=YES
Specifies to data management that you will be processing input or output records sequentially in a work area and not in the I/O area. Supported for DTFSD macro and sequentially processed files or partitions defined by DTFNI or DPCA macros; not supported for DTFDA files.

Do not specify IOREG keyword parameter when you specify the WORKA keyword.

Address of work area is specified with each issue of GET and PUT macro.

15.6.35. Specifying Issue of WRITE,ID Macro (WRITEID)

When you are processing a DTFDA file or randomly processing a DTFNI file and will issue the WRITE,ID form of the WRITE imperative macro to locate an output block by means of its relative disk address or ID, you must notify data management by specifying the WRITEID keyword parameter in your DTF. (This use of the WRITE macro is detailed in 15.7.11.4.)

Keyword Parameter WRITEID:

WRITEID=YES
Specifies that you will issue a WRITE,ID macro to the randomly processed file defined by this DTFDA or DTFNI declarative macro to locate an output block by its relative disk address (ID). Not supported for DTFSD files. See 15.7.11.4 for the WRITE,ID macro.

15.6.36. Specifying Issue of WRITE,KEY Macro (WRITEKEY)

When you are processing a DTFDA file or randomly processing a DTFNI file and will issue the WRITE,KEY form of the WRITE imperative macro to rewrite or update a block just read by the READ,KEY macro, you must notify data management by specifying the WRITEKEY keyword parameter in your DTF. (These uses of the WRITE and READ macros are detailed in 15.7.11.5 and 15.7.14.2.)

Keyword Parameter WRITEKEY:

WRITEKEY=YES
Specifies that you will issue a WRITE,KEY imperative macro to the randomly processed file defined by this DTFDA or DTFNI declarative macro to rewrite a block just retrieved by the READ,KEY macro. Not supported for DTFSD files. See 15.7.11.5 for the WRITE,KEY macro; 15.7.14.2 for the READ,KEY macro.

15.6.37. Nonstandard Forms of the Keyword Parameters

In order to minimize any recoding you may need to revise programs previously prepared to run under other data management systems, OS/3 data management accepts certain variant spellings for the keyword parameters described in this section. The nonstandard forms of these keywords, listed alphabetically, are as follows:

Nonstandard Spelling	OS/3 Standard Form	Nonstandard Spelling	OS/3 Standard Form
AFTR	AFTER	RDID	READID
BKSZ	BLKSIZE	RDKY	READKEY
EOFA	EOFADDR	REL	RELATIVE
ERRO	ERROPT	SKAD	SEEKADR
IOA1	IOAREA1	SRCM	SRCHM
IOA2	IOAREA2	TYPF	TYPEFLE
IORG	IOREG	UPDT	UPDATE
KARG	KEYARG	VBLD	VARBLD
KLEN	KEYLEN	VRFY	VERIFY
LBAD	LABADDR	WRID	WRITEID
RCFM	RECFORM	WRKY	WRITEKEY
RCSZ	RECSIZE	WORK	WORKA

Table 15—7 summarizes nonindexed file declarative macro keyword parameters.

Table 15–7. Summary of All Declarative Macro Keyword Parameters Used With the Nonindexed File Processor System (Part 1 of 2)

Keyword Parameter	Keyword Specification	Declarative Macros				Used to Specify or define
		DTFSD	DTFDA	DTFNI	DPCA	
ACCESS	EXC	X	X	X	–	This DTF: read/update/add use Other jobs: no access
	EXCR	X	X	X	–	This DTF: read/update/add use Other jobs: read use
	SRD	X	X	X	–	This DTF: read use Other jobs: read/update/add use
	SRDO	X	X	X	–	This DTF: read use Other jobs: read use
AFTER	YES	–	X	X	–	Issue of WRITE, AFTER or WRITE, RZERO macro
BLKSIZE	n	R	R	R	R	Length of I/O buffer
EOFADDR	symbol	R	–	X	X	Address of end-of-file/partition routine
ERROPT	IGNORE					Availability of record in I/O area despite parity error
	SKIP					Bypass input record or ignore output record despite parity error
ERROR	symbol	X	X	X	–	Address of user error routine
IDLOC	symbol	–	X	X	–	Define field for next available record address
IOAREA1	symbol	R	R	R	R	Address of primary I/O buffer
IOAREA2	symbol	X	–	X	X	Address of secondary I/O buffer
IOREG	(r)	X	–	X	X	I/O buffer index register
KEYARG	symbol	–	X	X	X	Address of field containing key search argument
KEYLEN	n	–	X	X	X	Length of key
LACE	n	X	X	X	X	Lace factor for record interlace operations
LABADDR	symbol	X	X	X	–	Address of user header/trailer label processing routine
LOCK	NO	X	X	X	–	Specifies that file lock is not to be set on a lockable file at OPEN
OPTION	YES	X	–	X	–	Optional file for input
PCA(n)	symbol	–	–	X	–	Partition address, where n = 1–7
READID	YES	–	X	X	–	Issue of READ, ID macro
READKEY	YES	–	X	X	–	Issue of READ, KEY macro
RECFORM	FIXUNB VARUNB FIXBLK VARBLK	X X X X	X X – –	X X X X	X X X X	Format of data records
RECSIZE	n	X	–	X	X	Record size
RELATIVE	R T	– –	X X	X X	– –	Relative addressing method

Table 15—7. Summary of All Declarative Macro Keyword Parameters Used With the Nonindexed File Processor System (Part 2 of 2)

Keyword Parameter	Keyword Specification	Declarative Macros				Used to Specify or define
		DTFSD	DTFDA	DTFNI	DPCA	
SAVAREA	symbol	X	X	X	—	Address of register save area
SEEKADR	symbol	—	R	R	—	Address of seek address field
SIZE	n	—	—	X	X	Percent of total file allocation to be initially assigned to partition
SRCHM	YES	—	X	X	—	Multi-track search for key
SUBFILE	YES	—	—	X	X	Support of subfiles
TRLBL	YES	X	X	X	—	Support trailer labels
TYPEFLE	INPUT OUTPUT INOUT	X X X	X X —	X X X	— — —	Define file type and mode of label processing
UOS	n	—	—	X	X	Secondary allocation
UPDATE	YES	X	—	X	—	Update of sequential input file
VARBLD	(r)	X	—	X	X	Count of residual I/O buffer space for VARBLK file
VERIFY	YES	X	X	X	—	Read/check of output records to be performed
WORKA	YES	X	—	X	X	Sequential processing in work area
WRITEID	YES	—	X	X	—	Issue of WRITE, ID macro
WRITEKEY	YES	—	X	X	—	Issue of WRITE, KEY macro

LEGEND:

X = Optional
R = Required
— = Not supported
▒ = Assumed specification if keyword not specified.

15.7. IMPERATIVE MACROS FOR NONINDEXED DISK FILES

You inform the nonindexed file processor system of the distinct operations that data management is to perform on your files and partitions by including the appropriate file processing imperative macroinstructions in your program.

There are 18 of these imperative macros available to you for processing your nonindexed files and partitions; Table 15—8 summarizes their functions.

The paragraphs following the table describe the imperative macros in detail; the macro descriptions are arranged in four groups, according to the file processing functions involved:

- Initialization and Termination Macros:

  OPEN

  CLOSE

LBRET

SETP

SETS

POINTS

FEOV

SETF

- Creation, Addition, and Updating Macros:

  PUT

  TRUNC

  WRITE

- Retrieval Macros:

  GET

  RELSE

  READ

- Validation and Positioning Macros:

  CNTRL

  WAITF

  NOTE

  POINT

Before issuing an imperative macro to an OS/3 data management file, you must provide a 72-byte save area, full-word aligned, into which data management expects to place the contents of your registers. You may load general register 13 in your program, or use the SAVAREA DTF keyword parameter to specify the address of the register save area. (See 15.6.23.)

Table 15—8. Summary of Imperative Macroinstructions for Processing Nonindexed Disk Files

Macro	Secondary Operands*	May be issued to			Remarks
		DTFSD Files	DTFDA Files	DTFNI Files	
OPEN	–	X	X	X	File or partition initialization
CLOSE	–	X	X	X	File or partition termination
LBRET	None	X	X	X	Creating, retrieving, and updating standard user header and trailer labels
SETP	**partition-name**	–	–	X	Select partition for subsequent processing
SETS	**subfile-number**	–	–	X	Select subfile or terminate current subfile creation
POINTS	–	–	–	X	Initialize to first block of file or partition
REOV	–	X	–	X	Terminate processing of current volume
SETF	INPUT OUTPUT UPDATE	X	–	X	Set processing direction for type INOUT file
PUT	**workarea**	X	–	X	Record-level output, sequential mode
TRUNC	–	X	–	X	Terminate output processing of current block
WRITE	ID	–	X	X	Block-level output by relative address
	KEY	–	X	X	Block-level output by direct write
	RZERO	–	X	X	Track initialization
	AFTER	–	X	X	Write current block in next sequential position in file
	AFTER, EOF	–	X	X	Record end-of-dat ID in DTF and format 2 label
GET	**workarea**	X	–	X	Record-level retrieval, sequential mode
RELSE	–	X	–	X	Terminate input processing of current block
READ	ID	–	X	X	Block-level retrieval by relative address
	KEY	–	X	X	Block-level retrieval by key
CNTRL	SEEK	X	X	X	Seek to current track or to track indicated by SEEKADR
WAITF	–	–	X	X	Wait on I/O completion; required after READ or WRITE
NOTE	–	–	–	X	Access current relative block address
POINT	**address-field**	–	–	X	Position file to relative block address in **address-field**

*Except for the LBRET macro, **filename** is assumed for operand-1; operands listed are secondary.

OPEN

15.7.1. Opening a Disk File (OPEN)

You will use the OPEN imperative macro to open a disk file defined by the appropriate DTF macro instruction in order to initialize it (and its partitions, if any) before it may be accessed by the logical IOCS processor. The OPEN macro instruction calls the appropriate transient routines to perform the following functions:

- validating and completing the file or partition tables;
- validating or creating system standard labels; and
- reading or writing user standard header labels.

If you define a DTFNI file to have more than one partition (by specifying two or more PCA keyword parameters and by coding the necessary DPCA declarative macros), you initialize all partitions by issuing an OPEN macro for the file.

This is the format of the OPEN macro:

LABEL	Δ OPERATION Δ	OPERAND
[name]	OPEN	filename-1[,...,filename-n] (1) 1

Positional Parameter 1:

filename
Is the label of one or more corresponding DTF declarative macroinstructions in your program. You may have as many as 16 files opened.

(1) or **1**
Indicates that you have preloaded register 1 with the address of the DTF filetable. (You use (1) or 1 as the operand only when you have a single DTF.)

CLOSE

15.7.2. Closing a Disk File (CLOSE)

After you have completed your processing of a disk file, you must issue the CLOSE macro to check that all your I/O orders have actually been completed and to read or write the system standard and optional standard trailer labels: in other words, to *terminate* the file. Once you have terminated a file with the CLOSE macro (which calls a transient routine to perform all of these functions), you cannot access it again unless you issue another OPEN macro. An important point to note is that you do not terminate file *partitions* separately; once you have done with all the partitions of a file you intend to process, you terminate operations by issuing one CLOSE macro for the *file* that contains them. (For this reason, the partition name never appears as an operand of the CLOSE macro.)

The format of the CLOSE macro is:

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

The three basic ways to code the CLOSE macro involve the use of symbolic addresses in the operand.

Positional Parameter 1:

filename
Is the label of the DTF declarative macro in your program; there may be 1 or as many as 16 files named.

(1) or **1**
Specifies that you have preloaded register 1 with the address of the DTF file table. You may use (1) or 1 when you have only one file to terminate.

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

LBRET

15.7.3. Processing Optional User Labels (LBRET)

You will use the imperative macro LBRET in your label processing routine (whose symbolic address is specified to data management via the LABADDR keyword parameter of your DTF) to create, retrieve, or retrieve and update optional user header or trailer labels. Note that LBRET is the only data management macro your label processing routine may issue. The maximum number of each type of label you may process is eight.

When your LABADDR routine receives control, data management will have loaded general register 1 with the address of the I/O buffer for use in processing input and output UHLs and UTLs. You must always use register 1, even though you may have specified only one buffer with the IOAREA1 keyword; it is not possible to use a work area for processing user labels.

Your LABADDR routine is accessed when the file is opened and again when it is closed; register 0 contains the EBCDIC alphabetic character 0 in its least significant byte when the file is opened and the character F when it is closed. Your LABADDR routine should be coded to access register 0 and to process your header labels when the register contains 0 and your trailer labels when it contains F.

Another point to remember is that user header/trailer labels, if you have them at all, are maintained at the file level only; data management does not maintain them at the partition level. (For this reason, there is no LABADDR keyword parameter in the DPCA declarative macro.)

The format of the LBRET macro is as follows:

LABEL	Δ OPERATION Δ	OPERAND
[name]	LBRET	{1 / 2 / 3}

where:

1

Does not write or read a label; returns control to your program at the next instruction after the OPEN or CLOSE macroinstruction.

2

Writes a label to an output file or reads a label from an input file; then returns control inline to your program at the next instruction after the LBRET 2 macroinstruction.

3

Writes back to disk the label just read and returns control to your program inline at the next instruction after the LBRET 3 macroinstruction.

TYPEFLE=INOUT must be specified in your DTF declarative macroinstruction, or the file processing direction reset for update processing with the SETF macroinstruction.

15.7.3.1. Creating Optional User Labels

Your LABADDR label-creating routine delivers your standard user header or trailer labels to data management one at a time, up to the maximum of eight. Data management will write each label to the disk file. If you have specified LBRET 2, it will return control to your label routine after each label has been written, until the eighth label has been written to disk. Then data management will transfer control to you inline. If you are creating user labels, control returns to you at the instruction next after the OPEN macro call by which you initially opened the file for processing. If you are creating user trailer labels, of course, data management returns control to you inline at the instruction next after the CLOSE macro call by which you are terminating your processing of the file. (Remember that reading or writing optional user trailer labels is one of the functions performed by the CLOSE macro.) Remember also where labels are written:

- User header labels are written by the LBRET macro on the first track of each volume of a DTFSD file, and on the first track of the first volume of a DTFDA or DTFNI file. They are 80 bytes long; their simple format is shown in 14.2.4.

- User trailer labels are written on the first track of each volume of a DTFSD file and on the first track of the first volume of a DTFDA or DTFNI file, following your user header labels. Their format and content are similar to those of the user header label and are also shown in 14.2.4.

When you have fewer than eight user labels of either type to create, you issue LBRET 1 when you have created the last one. After it has written the last label to your disk file, data management will return control to your program at the instruction next inline after your OPEN macro call.

15.7.3.2. Retrieving or Updating User Labels

If you need to retrieve your user labels for updating or other label processing, you will specify in the DTF the type of label processing you intend to perform (TYPEFLE=INPUT or TYPEFLE=INOUT), specify TRLBL=YES if you are going to process user trailer labels, specify the address of your label processing routine (LABADDR), and open the file with the OPEN imperative macro. (You do not use the UPDATE keyword parameter of the DTF; this is *not* related to label processing but affects data I/O.)

When the file is opened, data management will deliver your user header and trailer labels, one at a time, to your LABADDR routine either until all existing user labels have been passed to you, or until you specify that you need no more, by issuing LBRET 1.

Your label routine processes each label delivered by data management and then returns control to data management by issuing the LBRET macro with 1, 2, or 3 for the positional parameter.

If you want to terminate label processing short of the maximum (eight standard user labels of each type), you issue LBRET 1 when you need no more (this implies, of course, that you keep track). Data management then transfers control to you inline, to the instruction next after your OPEN macro call.

When you are processing all your labels (but not updating them), you issue LBRET 2. Data management will retrieve the next label and pass it to your LABADDR routine. It will continue to do so until there are no more to process; then, after you have processed the last label, data management automatically transfers control to the next instruction after your OPEN macro call.

When you are updating your user header and trailer labels, you issue LBRET 3. Here, data management will update the label just read, writing the new label to disk in the place of the old.

SETP

15.7.4. Accessing a Selected File Partition (SETP)

When you open a multipartitioned DTFNI file for processing, the only one of its partitions that you may then access is PCA1, the partition defined in the DTF itself. All partitons of the file are initialized when you issue an OPEN macro for the file, but only partition 1 set active; you cannot access any partition other than this first one with the OPEN macro alone. To select another partition of the opened file, you need the SETP macro.

The SETP macro acts to select the new partition, but it is important to remember that it positions you for processing this partition at its *current* position; that is, at the next accessible block after the point at which you were last processing. If you want to begin at some *other* point, you will need to issue additonal macros, for example, SETS (15.7.5), POINTS (15.7.6), or POINT (15.7.18).

Once you have accessed a partition with the SETP macro, all subsequent processing continues on the selected partition until you issue another SETP macro. All other imperative macros with which you may process within a partition (POINT, POINTS, SETS, and NOTE) depend on you to select the proper partition before calling them.

This is the format of the SETP macro; notice that it is the *only* imperative macro that may have a partition name in the operand.

LABEL	Δ OPERATION Δ	OPERAND
[name]	SETP	{filename / (1) / 1}, {partition name / (0) / 0}

Positional Parameter 1:

filename
Is the label of the DTFNI declarative macro that describes the already-opened file of which partition-name is part.

(1) or **1**
Indicates that you have preloaded register 1 with the address of the DTFNI file table.

Positional Parameter 2:

partition-name
Is the label of the partition control appendage (PCA1—7) that denotes the partition you want to access. (This is, of course, the same thing as the label of the corresponding DPCA declarative macro for PCA2—7 and the label assigned to the partition defined by PCA1 within the DTFNI macro.)

(0) or **0**
Indicates that you have preloaded register 0 with the address of the partition table defined by the DTFNI keyword (PCA1—7) that describes the partition you want to access.

Each DTFNI file table maintains the address of the partition that is currently active; when you issue a SETP macro, data management modifies this current partition address to indicate which partition you have selected to be active for subsequent file accesses. (When you close the file, you have no further access to the file until you initialize it again with a subsequent OPEN macro, which once more sets PCA1 active.)

If you have specified an index register (via the IOREG keyword parameter of your DTF), each SETP macro you issue will cause the index register to be loaded for the partition you are accessing. (During the opening of a multipartitioned file, only the index register for PCA1 will have been loaded.)

SETS

15.7.5. Processing Subfiles within a Partition (SETS)

Within each partition of a DTFNI file, you may establish as many as 71 subfiles. Subfiles must be created sequentially, but you may access them at random for data retrieval. If you intend to use subfiles in the first partition (PCA1) of a DTFNI file, you specify the SUBFILE keyword parameter in the DTF; data management reserves one track of the first volume of the file for maintenance of a subfile table. Similarly, if you want subfiles supported for any of the subsequent partitions, you specify the SUBFILE keyword parameter in the DPCA declarative macros describing these partitions.

Once you have selected the partition to be subdivided, the SETS imperative macro is the one that provides you with the ability to create and subsequently retrieve the data in the partition subfiles. This is its format:

LABEL	Δ OPERATION Δ	OPERAND
[name]	SETS	{filename / (1) / 1}, {subfile-no. / (0) / 0}

Positional Parameter 1:

filename
Is the label of the DTFNI macroinstruction describing the file to which the subdivided partition belongs.

(1) or **1**
Indicates that register 1 has been preloaded with the address of the DTF file table.

Positional Parameter 2:

subfile-no
Is the decimal integer number of the subfile (1 through 71) to be referenced.

(0) or **0**
Indicates that register 0 has been preloaded with the subfile number.

The reason that the *partition* name does not appear in the operand is that the SETS macro expects that you have already selected the partition you want, by issuing an appropriate SETP macro before calling it. If you have not done so, or if the SETP macro you issue is for the wrong file or partition, data management sets the *invalid subfile* bit (byte 3, bit 3) in the error flag field of the DTFNI file table, referenced as *filenameC* (Appendix B). You should check this bit whenever you issue a SETS macroinstruction.

What the SETS macro does is to maintain a partition-relative subfile table, on the track of the first volume of the file that you reserved by specifying SUBFILE keyword parameter in the DTFNI or DPCA macro. (This is either the first track on the volume or, when optional user labels are present, the first track after the user label track.) Data management uses this table to keep track of the start of each subfile: the address of the record that starts it. No record is kept by data management of the *end* of a subfile; unless you keep track yourself of the record with which it ends, it is possible to process through a subfile to the end of the partition or logical EOF.

The subfile table is not available for you to access, but you may examine it in a disk print taken with the system utility (SU) symbiont. To know its contents may help you visualize what the SETS macro is doing for you when you create or retrieve subfile records. There is one 6-byte entry in the table for each subfile, consisting of the relative block address of the record at the start of the subfile and, if the records are in blocked format, its displacement into the block. When you are creating a subfile, you issue a SETS macro to insert the address of the next available block or record as an entry in the subfile table. (Remember that you are creating subfiles sequentially, although you may retrieve the subfiles at random.)

During retrieval, the SETS macro you issue moves the table entry for the subfile you have selected to the current relative address field of the DTFNI or DPCA file table; the file may then be processed between the limits of the current relative address of the start of the subfile and the logical end of the file or partition. This selective positioning of a file for subfile processing is a useful ability to keep in mind.

For creation of subfiles (output), the SETS macro must be issued following the output of the last record to each subfile. SETS for output indicates termination of the subfile.

For retrieval of subfiles (input), the SETS macro should be issued prior to the first GET of a subfile record. SETS for input initializes processing to the start of a subfile.

POINTS

15.7.6. Initializing Position of a File or Partition (POINTS)

When you are processing within a file partition and want to get back to the start of it (that is, to reset the current relative block address from its present value to the partition-relative address of the first block of the same partition), you will use the POINTS macro that is designed to do just this.

When you want to *change* partitions, you must first issue a SETP macro (15.7.4) to select the new partition, and then issue the POINTS macro to get to the beginning of that: POINTS initializes the relative block address of the *current* partition.

This is the format of the POINTS imperative macro:

LABEL	ΔOPERATIONΔ	OPERAND
[name]	POINTS	{ filename (1) 1 }

Positional Parameter 1:

filename
Is the label of the corresponding DTFNI macroinstruction in your program.

(1) or **1**
Indicates that register 1 has been preloaded with the address of the DTFNI file table.

	LABEL (1)	ΔOPERATIONΔ (10)	(16)	OPERAND	Δ
1.		POINTS		PART03	
2.		LA		(1),PART04	
		POINTS		(1)	

FEOV

15.7.7. Forcing End-of-Volume Procedures (FEOV)

When you are processing a DTFSD file for input or output, and want to discontinue short of the actual end of this volume and to begin processing the next, you issue the FEOV imperative macro.

This macro initiates end-of-volume (EOV) procedures on the current volume, which is closed just as if the actual EOV had been reached. Volume swapping is performed, and your next GET or PUT macroinstruction continues processing on the next sequential volume. The FEOV macro may be used only for DTFSD files; these have only one volume mounted at a time. It may not be used for sequentially processed files described by the declarative macro because all volumes are always online, and there is no "current volume" in the sense used here.

When you are processing sequential blocked input files and need to skip over the records remaining in a block in order to resume with the next block of the same volume, a different macro, RELSE, is available (15.7.13).

This is the format of the FEOV macro:

LABEL	ΔOPERATIONΔ	OPERAND
[name]	FEOV	{ filename (1) 1 }

Positional Parameter 1:

filename
Is the label of the corresponding DTFSD macroinstruction in the program.

(1) or **1**
Indicates that you have preloaded register 1 with the address of the DTFSD file table.

Example:

1 LABEL	10 ΔOPERATIONΔ	16 OPERAND	Δ
	FEOV	INFILE	

Treats the file described by the DTFSD macroinstruction, whose label is INFILE, as if logical end-of-volume address had been accessed.

SETF

15.7.8. Setting File Processing Mode (SETF)

The SETF macroinstruction enables you to set the processing direction (change the type of file) for DTFSD files, or sequentially processed DTFNI files, described by the keyword parameter TYPEFLE=INOUT. You should not issue the SETF macro during your file processing; it should, instead, be issued between your terminating file processing (with the CLOSE macro) and your opening it again, or before the OPEN issued to the file at the start of your program.

This is the format of the SETF macro:

LABEL	ΔOPERATIONΔ	OPERAND
[name]	SETF	{filename | (1) | 1}, {INPUT | OUTPUT | UPDATE}

Positional Parameter 1:

filename
Is the label of the DTFSD or DTFNI macroinstruction that defines the INOUT file.

(1) or **1**
Indicates that register 1 has been preloaded with the address of the DTFSD or DTFNI file table describing the INOUT file.

Positional Parameter 2:

INPUT
Indicates that the INOUT file is to be set for input processing, without updating.

OUTPUT
Indicates that the INOUT file is to be set for output processing.

UPDATE
Indicates that the INOUT file is to be set for input processing, with updating.

15.7.9. Output of Sequential Disk Files (PUT)

You will use the PUT imperative macro to create, extend, or update disk files processed sequentially. These are either files defined by the DTFSD declarative macroinstruction (15.5.1) or those files and partitions defined by the DTFNI and DPCA declarative macros (15.5.3 and 15.5.4).

Essentially, the PUT macro handles record-level output for sequential files in either of two ways. It delivers an output record to data management in the current output area, if you are processing in IOAREA1 or IOAREA2. But, if you are building your output records in a work area (specifying WORKA=YES), the PUT macro delivers these to data management for writing to disk directly from there. Either way, the record is no longer available to you, once delivered.

If your output records are unblocked, they are delivered singly to disk; if your records are to be blocked, data management handles blocking automatically for you from the work area. You must take care of blocking for variable-length blocks constructed in an I/O area, however, and you may optionally write out short blocks of fixed-length records. Both of these actions are easily accomplished: see 15.7.9.4 and the TRUNC imperative macro (15.7.10).

When your output records are blocked, or when you have specified a standby I/O area (IOAREA2) but no work area, you must supply an index register via the IOREG keyword parameter of your DTF or DPCA macro so that data management has a place to put the starting address of the current record position in the output area (15.6.11). You do *not* do this when building records in a work area, because you specify its address every time you issue the PUT macro. And, if you are processing unblocked records in a single I/O area (IOAREA1), you may reference these directly by means of the name you have assigned to the area and do not need an index register.

An important point to remember is that, after data management writes the current output data to disk from the output or work area, it does *not* clear these areas. You must be careful either to clear the area yourself, or always to supply records — padded with blanks if necessary — which completely fill out the work area or I/O area. Otherwise, spurious characters left over from previous records may appear in your output data. When you are processing in a work area, it is freed for subsequent processing (but not cleared) each time data management moves an output record from there into the I/O area for you.

Sequentially processed DTFNI files with keys may also be easily output with the PUT macro; see 15.7.9.5.

This is the format of the PUT macro:

LABEL	Δ OPERATION Δ	OPERAND
[name]	PUT	{ filename / (1) / 1 } [, { workarea / (0) / 0 }]

Positional Parameter 1:

filename
Is the label of the corresponding DTFSD or DTFNI macroinstruction that defines the output file.

(1) or **1**
Indicates that you have preloaded register 1 with the address of the DTF file table.

Positional Parameter 2:

workarea
Is the label of the work area from which the output record may be obtained.

(0) or **0**
Indicates that register 0 has been preloaded with the address of the work area.

If omitted, indicates that you have chosen to reference the current record either by means or a register (IOREG keyword parameter) or by directly accessing the data relative to the name you have assigned to IOAREA1. You may use the latter method only for unblocked records processed in a single I/O area.

NOTE:

When the work area is used, the keyword parameter WORKA=YES must be present in the DTF statement.

15.7.9.1. Creating a Sequential Disk File

The PUT macro gives you the ability to create a new disk file and then to process it sequentially: this amounts to using the disk file as a work file. You use the same DTF file table to describe the file when you create it and when you process it; such a file may be defined by a DTFSD or DTFNI declarative macro (15.5.1 or 15.5.3). The procedure is as follows:

1. Define the file, specifying the DTF keyword parameter TYPEFLE=INOUT among the other parameters you need.

2. Open the file for output, using the OPEN macroinstruction (15.7.1); then create the file, using the PUT macro to write your records to disk.

3. Close the file, using the CLOSE macroinstruction (15.7.2).

4. Issuing the SETF macroinstruction (15.7.8), reset the file processing direction to INPUT or UPDATE.

5. Reopen the file.

6. Retrieve your records sequentially, via the GET macroinstruction (15.7.12), or optionally, update them with pairs of GET and PUT macros (15.7.9). It is important to remember that when you update with the PUT macro, you may never change the record length.

7. Close the file.

The following coding example shows how you might do this for a file with the logical name INVNTRY. Once created as a TYPEFLE=INOUT file, the file is closed; after you reset the file processing direction to UPDATE, you reopen and update it by issuing paired GET and PUT macros to retrieve records and write them to disk.

Example:

```
   LABEL      ΔOPERATIONΔ      OPERAND                    Δ
   1          10          16
1. INVNTRY    DTFSD       .
              .
              .
2.            TYPEFLE=INOUT,
3.            .
              .
              .
4.            OPEN        INVNTRY
              .
              .
              PUT         INVNTRY
              .
              .
              .
              CLOSE       INVNTRY
5.            SETF        INVNTRY,UPDATE
              OPEN        INVNTRY
```

	LABEL (1)	OPERATION (10)	OPERAND (16)
6.		GET	INVNTRY
		.	
		.	
		.	
		PUT	INVNTRY
7.		CLOSE	INVNTRY

1. Other DTF keyword parameters, including required keywords not pertinent to the example, are not shown.

2. Must be INOUT file

3. Other parameters are not shown.

4. Creating file, using disk as work file

5. Now reset file processing direction to update and reopen file

6. Issuing paired GET/PUT macros to retrieve records, update them, and rewrite to disk.

7. Terminate file; cannot be accessed until reopened.

15.7.9.2. Updating and Extending an Existing Disk File Processed Sequentially

The *logical* end of a disk file — the relative block number of the last data block — plus 1 is recorded by data management in the DTF and in the format 2 label of the file as its end-of-data (EOD) address. (See D.3.2.) In a DTFSD file, this address is also called the logical end-of-file (EOF) address. No EOF sentinel or other flag is recorded in the data area of the file to mark this point, and there is no data record in the block at the EOD address.

When you are sequentially processing a nonindexed disk file in an update mode (that is, you have specified UPDATE=YES in the DTFSD or DTFNI declarative macro), you may extend the file beyond its current EOF record by the following procedure, which you will include in your end-of-file routine. (When you issue a GET macro and an end-of-file condition is detected, control returns to you at the address specified by your EOFADDR parameter in the DTF. You may not extend a file beyond its current volume by this means, but see 15.7.9.3.)

1. Issuing a GET macro, you place the record to be added to the file in the I/O area (IOAREA 1 or 2) or in the work area you have specified.

2. Issue a PUT macro, which causes the new record to be added.

3. Issue another GET macro and follow it by a PUT macro; this will output the next record to be added to the file.

4. Terminate the file by issuing a CLOSE macro when you have completed all your additions to it (15.7.2).

An important point to remember is that you must have anticipated the eventual need to extend this file and provided for extension by the appropriate job control statements issued at the time you originally created it. Extension is done automatically for you by data management through the disk space management routines of the OS/3 supervisor; you never need to call the supervisor EXTEND macro in your program.

Another point to remember is that you must not issue two PUT macros in succession in the update mode; this will be flagged as an *invalid macro sequence* (byte 0, bit 6 of *filenameC* (Appendix B)).

15.7.9.3. Extending an Existing DTFSD Output File

Extending a sequential file within your EOFADDR routine is discussed in 15.7.9.2. You have another means available for extending an existing sequential file: processing it in the output mode (that is, by specifying TYPEFLE=OUTPUT in the DTFSD declarative macro, or resetting the direction of file processing to OUTPUT with the SETF imperative macro (15.7.8)). This procedure also requires that you issue appropriate job control statements, which are discussed in what follows.

First, the current last volume of the file is mounted; for a disk file described by the DTFSD declarative macro, this is always a specific volume because only one volume is mounted at a time. (All volumes of a DTFNI file, processed sequentially or not, are always online.)

When you issue an OPEN macro for the file, if you have specified the appropriate job control statements, data management will position the file to its current logical end-of-file (EOF) address.

You then issue the PUT macros necessary to add records beyond the current logical EOF.

If you need them, allocate additional disk volumes to the file, and you may continue to extend it beyond its current last volume to subsequent volumes.

The OS/3 job control statements you need to specify to extend a sequential disk file by this method are discussed in detail in the job control user guide, UP-8065 (current version). Briefly, what you need in your device assignment set is an LFD statement that contains the logical name of your file (the name by which your program references it) and indicates by an EXTEND (third positional parameter) that the extend mode of processing is to take place: the sequential file will be extended by appending the new records to the present end of the file.

It is important for you to remember that all LFD names within a job step must be unique; if more than one file is given in the same name, only the last one to be specified is available for any operation — including extension by this method.

15.7.9.4. Output of Blocked Records, Sequential Disk Files

As you are forming variable-length records for output to a sequentially processed disk file via the PUT macro, you will be determining the length of each record and placing this information in the first two bytes of the 4-byte record descriptor word (RDW), at the head of each record (14.3.2). This information is contained within the record and is always in the I/O area whenever the record is.

When you are outputting variable-length *blocked* records from an I/O area to disk and are not using a workarea, you must test whether the next record will fit into the remaining space in the I/O area before you issue the PUT macroinstruction for it. Data management informs you of the amount of space remaining in the I/O area after each variable-length record is moved in by a PUT macro; it does so by placing the number of bytes of residual space into the general register you have specified (via the VARBLD keyword parameter of the DTFSD or DTFNI macro or the DPCA macro (11.6.34). If you find that the next record will fit, add it to the current block with the PUT macro. If you find that it will *not*, you instead issue a TRUNC macro to write out the current block to disk and free the entire I/O area for building the next (15.7.10). Data management will calculate the block size and will enter it into the first two bytes of the 4-byte block descriptor word (BDW), as explained in 14.3.2, before writing the block to disc.

On the other hand, when you are forming your variable-length blocked records in a *work area*, each PUT macro you issue causes data management to test whether the record it moves will fit into the I/O area. If it will, it is added to the block currently being built; if it will not fit, data management first writes out the current block and then starts a new block with the current record.

15.7.9.5. Output of Sequential DTFNI Files With Keys

Files defined by the DTFNI declarative macro and processed sequentially may have a key associated with each *block* of data. Before you issue a PUT macro to output such a block to disk, you must place this key at the head of the block (just as you would before issuing a WRITE macro with an ID positional parameter for direct access method output of blocks with keys). When you issue the PUT macro, data management will write the key and data portion of the block to disc. Subsequent sequential retrieval of blocks having keys (via the GET macro) will, similarly, cause transfer of both the key and data. Remember that data management will perform the normal blocking and deblocking for DTFNI files processed sequentially.

Another point to remember is that, when you are creating or updating nonindexed files with keys, using the PUT macro, you must specify the key length via the KEYLEN keyword parameter of the DTFNI or DPCA macroinstruction (15.6.13).

15.7.9.6. Optional Sequential Output Files

When you have a program that you anticipate will not invariably be called on to process a particular sequential output disk file each time it is executed, you should designate this file as optional by specifying OPTION=YES in the DTFSD or DTFNI macro (15.6.16). On the occasions when you do not require the file to be output from your program, you need merely omit the device assignment set of job control statements that usually allocate the file to a disk. When your program executes the OPEN macro for the optional file, the OPEN transient marks the file as optional, and it disables the PUT macro mechanism so that no I/O is performed.

You should not forget to specify the OPTION keyword parameter for an optional file: if you do forget, and the file has not been allocated by job control when your program is executed, it is impossible to process the file. Data management will transfer control to the address of your error routine.

TRUNC

15.7.10. Output of Short Variable Blocks to Sequential Disk Files (TRUNC)

The TRUNC imperative macro is used with sequentially processed, variable-length, output files, defined by the DTFSD or DTFNI macros, to enable you to write short blocks of output data to disk. Its use is *mandatory* with variable-length, blocked records that you build in the output I/O area. Its function is to notify data management that the block currently being built is to terminate and is to be written out to disk. Data management calculates the block size and inserts it in the first two bytes of the block descriptor word (BDW) before it writes the block to disk.

When you have formed a variable-length record to be added to the block building in the I/O area, you must determine whether there is room for it in the remaining space in the I/O area before you issue the PUT macro. You compare the current record length, which you have placed in the first two bytes of its record descriptor word (RDW), with the number of bytes remaining in the I/O area. (Data management informs you of the space left in the I/O area by placing the number of bytes of residual space in the general register that you designated by specifying the VARBLD keyword parameter in the DTFSD, DTFNI, or DPCA macro (15.6.34); it updates this number each time a variable-length record is added to the current block by a PUT macro.)

If the current record will fit in the space remaining, you will issue a PUT macro to add it to the current block. But if it will not, you issue a TRUNC macro to write the current block to disk and to free the I/O area for building the next block. A subsequent PUT macro will then start off the next block with the current record; the TRUNC macro resets the IOREG index register to point to the new current area address of the next available output I/O area.

This is the format of the TRUNC macro:

LABEL	ΔOPERATIONΔ	OPERAND
[name]	TRUNC	{ filename (1) 1 }

Positional Parameter 1:

filename
Is the label of the corresponding DTFSD or DTFNI macroinstruction in the program.

(1) or **1**
Indicates that register 1 has been preloaded with the address of the DTF file table.

Example:

LABEL (1)	OPERATION (10)	OPERAND (16)
	TRUNC	OUTPUT

Sends to the output device the short block of records accumulated by PUT macroinstructions since the last TRUNC was issued or since a full block of records was sent automatically to the output device for the file described in the DTF macroinstruction whose label is OUTPUT.

WRITE

15.7.11. Random Output of Records to Disk (WRITE)

The WRITE imperative macro is a block-level output processing macro that, in its various forms and uses, provides you with the following capabilities with randomly processed disk files defined as output files by the DTFDA or DTFNI declarative macros, or as input/output files by the DTFNI macro:

- creating a newly allocated file;
- updating a previously created file by rewriting blocks to their original locations;
- extending a file by generating data blocks in space newly allocated to it;
- overwriting unwanted data in an expired or newly allocated file;
- recording the logical end of a file or partition; and
- moving the disk access arm to a new track and ensuring that the new track is initialized.

Three forms of the WRITE macro output a block from main storage to disk; the main storage address from which your data is written is contained in the location specified by the IOAREA1 keyword parameter of your DTF. Its input counterpart for disk files that may be processed randomly is the READ imperative macro, which is also a block-level processing macro (15.7.14).

Because these two operate a block-level, you must control any record blocking and deblocking that may be necessary, as OS/3 data management handles this function automatically for you only for *sequentially* processed DTFSD and DTFNI files.

For disk files you define with the DTFDA macro, of course, this is no problem, as only *unblocked* record format (fixed- or variable-length) may be specified for these files. For randomly processed DTFNI files, however, in which you may have blocked records, you will need to control their blocking and deblocking with the PUT and GET macros, used in conjunction with READ and the WRITE,KEY or WRITE,ID form of the WRITE command (15.7.11.4 and 15.7.11.5).

In all uses, you must issue a WAITF imperative macro (15.7.16) after each READ or WRITE macro you issue, to ensure that the intended data transfer has taken place, before issuing another imperative macro. Remember also that none of your transferred records will be check-read for parity unless you specify the VERIFY=YES keyword parameter in your DTF macro; check reading necessarily increases the execution time for each WRITE operation (15.6.33).

Another DTF keyword parameter involved is IDLOC, which enables you to have the relative disk address (ID) of your block returned to a specified field. The form of the returned ID is governed by your specification of another DTF keyword parameter, RELATIVE, which you might also review (15.6.22). The uses of the IDLOC keyword parameter are explained in 15.6.7 and further discussed in what follows.

WRITE, AFTER

15.7.11.1. Creating a Random Disk File by Sequential Load (WRITE,AFTER)

This is one form of the WRITE command you may use to create a file:

LABEL	Δ OPERATION Δ	OPERAND
[name]	WRITE	{ filename / (1) / 1 } ,AFTER

The second positional parameter, AFTER, specifies that the current block in main storage is to be written as the next sequential block on the current track and that the remainder of the track is to be cleared. If this current block, when written, occupies the last position on the track, data management sets the *last block on track accessed* bit (byte 0, bit 0) in *filenameC* after completion of the WAITF macro. (See Appendix B for the details on *filenameC*.) You should check this bit after each issue of the WAITF macro, and be prepared to move to another track if it is set, using, for example, the WRITE,RZERO form of the WRITE command (15.11.2) or the CNTRL imperative macro (15.7.15).

You should review the AFTER keyword parameter of the DTFDA and DTFNI declarative macros; this parameter must be specified when you use the WRITE,AFTER form of the WRITE command and precludes your use of the WRITE,ID function (15.7.11.4). Data management does not automatically preformat the file at OPEN when AFTER=YES is specified.

In the first positional parameter, *filename* represents the label of the DTFDA or DTFNI declarative macro; *(1)* or *1* indicates that you have preloaded general register 1 with the address of the DTF file table. The first positional parameter is specified in the same way for all forms of the WRITE command and will not be discussed further.

If you want to store the relative disk address or ID of the next block in the file or partition you are processing, you would specify the IDLOC keyword parameter in the DTF (15.6.7). The form in which this ID is returned to you is governed by your specification of another keyword parameter, RELATIVE, which you should review (15.6.22).

You have two basic ways of using the WRITE,AFTER imperative macro to create a new direct access disk file: a simple sequential load process that proceeds from the file start through all the tracks in succession, and another use, in conjunction with the WRITE,RZERO macro (15.7.11.12), which enables you to select the tracks to be filled in whatever order you choose. Because the WRITE,AFTER macro (having written a block to a file on a variable-sector 8411, 8414, 8424, 8425, 8430, or 8433 disk) then overwrites the remainder of the current track with binary 0's, you may also use it (with or without the WRITE,RZERO macro) to expunge unwanted data from an existng variable-sector disk file. You cannot do this if the file resides on the fixed-sector 8416 disk in OS/3.

When you open a newly allocated direct access file the first time for output, the OPEN transient positions you automatically at the head of relative track 001. If you are going to use the WRITE,AFTER macro, you have specified the AFTER keyword, and no preformatting has been done (if the file is on a variable-sector disk). If you issue successive WRITE,AFTER imperatives, each followed by a WAITF macro (15.7.16), you effect a sequential loading of your file, in the simple order in which you present your blocks to data management. You do not use the SEEKADR field's contents to control this macro, for the WRITE,AFTER macro used this way automatically shifts from track to track and cylinder to cylinder sequentially. However, you must arrange to move the contents of the IDLOC field (to which the following WAITF macro returns the ID of the next block in physical sequence in the file) into the SEEKADR field in order for this method to work. (One easy way for bringing this about is to define the IDLOC and the SEEKADR fields as the same area in main storage, as described in 15.6.7.) The file-filling sequence continues until you reach logical end of volume and an error condition in *filenameC* indicates that you have exhausted your file space, unless you have terminated loading sooner (as, for example, when reaching logical end of file (EOF) in your input file). You do not, in this method, need to test for setting of the *last block on track accessed* flag in *filenameC* because of the automatic movement to the head of the next track that data management performs.

You do need to keep your finger on track fill, however, when you use the WRITE,AFTER macro in the second method mentioned, for this controls your issue of the WRITE,RZERO macro to select the next track to be filled.

You should note a few more points about this method. The first is the *last block on track accessed* flag is set by OS/3 after you have issued the WRITE,AFTER macro that writes the block in the last position of the current track; in some other data management systems, this flag is not set until you issue a macro to write the next block, which will not fit on the track.

Another point is that the setting of this flag must be tested for in your program *inline* because accessing the last block is not a condition that causes control to transfer to your error routine: if you test *filenameC* for this flag only in your error routine, you will miss it. A third point is that, although the WRITE,AFTER macro does not require a seek address to guide it, the WRITE,RZERO macro does; you must, therefore, arrange to increment your SEEKADR fields's contents to the new relative track address you want before you issue it. You will probably have specified *relative track addressing* (RELATIVE=T) when you use this method for creating a file with WRITE,RZERO and WRITE,AFTER, but you may also use *relative record addressing* (RELATIVE=R), although this is harder to control.

One final point is most important if your file resides on an 8416 disk. Because of the fixed-sector format of this disk in OS/3, the action of the WRITE,AFTER macro is significantly different from the foregoing description in that the macro, having written a block, does *not* set all fields to binary 0 in the remainder of the blocks on the track. On the 8416 disk, moreover, there is no record 0 at the head of the track from which data management can be informed of the amount of unused space. For these reasons, you must always fill each 8416 disk track completely when you use the WRITE,AFTER macro. If you write only the one block at the head of the track, for example, and then pass on to another track, the residual data in the 39 remaining blocks is still there and may produce unpredictable results when your program encounters it in later retrieval operations.

WRITE,RZERO

15.7.11.2. Selecting and Initializing a New Track (WRITE,RZERO)

The form of the WRITE command to use for moving the disk access arm to a new track, and ensuring that the new track is initialized is:

LABEL	Δ OPERATION Δ	OPERAND
[name]	WRITE	{ filename / (1) / 1 } ,RZERO

Here, the second positional parameter, RZERO, specifies that data management will position the file for subsequent processing at the relative track address you have preloaded into the field specified by the SEEKADR keyword parameter of your DTF (15.6.24). This means that you have selected a new track, or that the track specified is to be treated as a new track, and that writing will begin at the first record on this track. Note that this form of the WRITE command does not actually output a record: it merely repositions the file so that you may write the subsequent records beginning with the first record of a new track. For writing the next record, you must follow this form with the WRITE,AFTER form of the WRITE command, (15.7.11.1).

Because the WRITE,AFTER macro clears the rest of the current track, this WRITE,RZERO/WRITE,AFTER combination is one you may use to create a new file or to overwrite (or erase data from) an expired or newly allocated file on a variable-sector disk.

If you want to store the relative disk address, or identifier (ID), of the next block in the file or partition you are processing with the WRITE,RZERO/WRITE,AFTER macros, you would specify the IDLOC keyword parameter in the DTF (15.6.7). The form in which this ID is returned to you is governed by your specification of another keyword parameter, RELATIVE, which you should also review (15.6.22). No ID is returned by the WAITF macro that follows the WRITE,RZERO macro; the contents of the IDLOC field are unchanged.

Use of the WRITE,RZERO macroinstruction requires that you specify AFTER=YES in your DTF; because data management does not preformat the file on a variable-sector disk when this keyword is specified, you may not issue the WRITE,ID macro to it.

WRITE,AFTER,EOF

15.7.11.3. Recording the Logical End-of-File (WRITE,AFTER,EOF)

This form of the WRITE macro may be used to record the logical end of data or end-of-file address in the DTF. Data management also records it in the disk format 2 label:

LABEL	Δ OPERATION Δ	OPERAND
[name]	WRITE	{ filename / (1) / 1 } , AFTER, EOF

The logical end-of-file (EOF) or end of partition is the relative disk address of the block one *beyond* the block (containing data) that is written the farthest or deepest into the file or partition. It is the address one block beyond the highest *used* relative disk address, and there is no data that belongs to your file there. Data management uses this address, which it also knows as the end-of-data ID (EODID), to prevent your reading extraneous data outside of the limits of the current partition or file space on disk, and it records the EODID in the disk format 2 label. (See D.3.2.)

If you issue a READ,ID macro that references a block whose relative disk address exceeds the EODID, data management sets the *invalid ID* flag (byte 0, bit 1) in *filenameC*, issues error message DM24, and branches to your error routine. (See Appendix B.) You may, on the other hand, *write* at or beyond the current logical end of file, if you have provided for file extension.

You should not need to issue the WRITE,AFTER,EOF macro in a new program written in OS/3, because data management automatically keeps track of the progress your program is making as it fills the file, and automatically records the EODID on file close. However, if you have a program coded for some other system where it was necessary for you to issue the macro, you need not remove your WRITE,AFTER,EOF macro call from it.

The WRITE,AFTER,EOF macro does not output a block, nor does it make an ID return to the IDLOC field. It is however, necessary for you to place in the SEEKADR field the relative disk address of the block that data management is to use in calculating and recording the EODID. Usually, this is the address of the block just written.

Note, however, that the *partition* name is never used in the first positional parameter. If you need to know where the end of a *subfile* occurs within a partition, you must maintain your own end-of-subfile data record; data management does not provide this service to you. On the contrary, it is possible to process through the end of a subfile to the logical EOF or end of partition.

Remember to specify the keyword parameter AFTER in the DTF when you intend to issue the WRITE,AFTER,EOF form of the WRITE macro (15.6.2); otherwise it will be flagged as an *invalid macro* (byte 0, bit 6 of *filenameC).*

WRITE, ID

15.7.11.4. Creating or Updating Blocks by Relative Disk Address (WRITE,ID)

You may use the following form of the WRITE imperative macro for creating a direct access file or for updating an existing one on disk:

LABEL	Δ OPERATION Δ	OPERAND
[name]	WRITE	{ filename / (1) / 1 } , ID

The second positional parameter, *ID*, specifies that a search is to be made in the output file for a block whose relative disk address (ID) matches the content of the SEEKADR field in your program (15.6.13). The ID you present to data management in this field may not be negative or zero; it must be in the form you specify with the RELATIVE keyword parameter (15.6.22). If you issue the WRITE,ID macro to process a file, you must specify the WRITEID keyword parameter in the DTFDA or DTFNI declarative macro defining the file (15.6.35); the AFTER keyword must not be specified (15.6.2). If the blocks in your file are keyed, you must specify the length of these keys with the KEYLEN keyword parameter (15.6.13).

Before you issue the WRITE,ID macro, you must ensure that the correct relative disk address is contained, as a fixed-point binary number in the form you have specified with the RELATIVE keyword, in the 4-byte SEEKADR field in your program. If you define the IDLOC and the SEEKADR fields to be the same physical area in main storage, data management is, in effect, providing a new relative disk address to your WRITE,ID macro automatically. (Refer to 15.6.7.)

In addition to providing your WRITE,ID macro with the correct ID, you must have the new block already formed in the I/O area before you issue the macro. If your file contains keys, moreover, you must place the key of each block in the I/O area ahead of the record proper (or ahead of a string of blocked records), allowing for the block descriptor and record descriptor words (BDW and RDW) as appropriate. In executing the WRITE,ID macro, data management locates the relative disk address you have specified (if it can) and writes the entire block at the location, including its key if yours is a keyed file.

Following your issue of the WRITE,ID macro — or of any form of the WRITE macro — you must issue a WAITF macro before issuing any other to the file. This is necessary to ensure that the intended I/O is completed. Among other things, the WAITF macro returns the relative disk address of the *next* block in physical sequence in the file to your IDLOC field, if you have specified this keyword, in the form you have specified with the RELATIVE keyword. If the relative disk address specified is not located, data management set the *record not found* flag (byte 1, bit 3) in *filenameC*, which you should test in your error routine when you are updating a file. If the block written by the WRITE,ID macro occupies the last possible position on the current track, data management sets the *last block on*

track accessed flag (byte 0, bit 0) in *filenameC*, but, because this is not an error condition, control does not pass to your error routine. If you are controlling movement through your file by incrementing the relative track number when you reach the end of track, you must test for the setting of this flag *inline*.

There are two ways to use the WRITE,ID macro to create a new direct access file: a simple file-filling operation that uses this macro alone, and an operation writing on specific tracks that you select and move to, via the CNTRL macro, followed by the WRITE,ID macro.

When you open a new DTFDA or DTFNI file for the first time for output on a variable-sector disk,* unless you have specified AFTER=YES in the DTF (15.6.2), the OPEN transients preformat every track in the file, writing the count fields throughout at intervals corresponding to your specified block size, and then position you at the head of relative track 001, ready to write the first block. By providing your WRITE,ID macro with the relative disk address for the first block, and thereafter incrementing the contents of your SEEKADR field, you may fill your file at will, in any sequence of *relative track* or *relative record* addresses. If you simply increment the relative record number in the SEEKADR field, for example (having specified RELATIVE=R), before each WRITE,ID macro issued, you achieve a file load in the simple sequential order in which you present your blocks. In this circumstance, you have no need to test for the *last block on track accessed* flag because data management automatically shifts you to the head of the next relative track. When you close the file, data management records the end-of-data ID in the DTF file table and in the disk format 2 label; you do not need to issue the WRITE,AFTER,EOF imperative macro for this.† On the other hand, if you need to structure the data in your direct access file in such a way that you must skip relative track 001 or certain other tracks during its initial loading, the first imperative macro to issue after opening the file might be the CNTRL macro (15.7.15), guided by a relative track address you have placed in the SEEKADR field to position you to the head of the first track you wanted to receive your data. After execution of the CNTRL macro, simply issue successive WRITE,ID macros, incrementing the record number in the SEEKADR field until end of track; then increment the relative track number before issuing the next CNTRL macro. (You cannot pair the WRITE,ID macro with the WRITE,RZERO macro for this mode of processing, because this macro also requires you to specify AFTER=YES in your DTF, and this is incompatible with using the WRITE,ID macro.)

Updating an existing direct access file with the WRITE,ID macro is conceptually different from creating a new one, although the action of the macro is the same. One thing to keep in mind, however, is that the new block is written in the place of the old one on disk, and data management requires that it be the same length. New blocks written to an existing file must have the same length as existing blocks. The BLKSIZE specification is checked at file open time against the block size recorded in the disk format 1 label; if these are unequal, data management issues error message DM17 (INVALID BLOCK SIZE SPECIFIED) and branches to your error routine. (Refer to Appendix B.)

**The variable-sector disks used with OS/3 data management are the SPERRY UNIVAC 8411, 8414, 8424, 8425, 8430, and 8433 Disk Subsystems. The SPERRY UNIVAC 8415, 8416, and 8418 Disk Subsystems are fixed-sector disks and are already preformatted at OPEN time.*

†In fact, because this macro requires that you specify the AFTER keyword in your DTF, and you cannot issue the WRITE,ID macro if you do, you cannot use WRITE,AFTER,EOF.

If you know the relative disk addresses of all the blocks that require updating, and you have the update information formatted to replace the entire block (including its key, if this is a keyed file), then you need not read your existing blocks into main storage. You need merely issue successive WRITE,ID macros to update the file, providing the relative disk addresses in the order of the update information in your input file.

On the other hand, if you must read the old blocks to determine whether or where to update them, you either would issue the READ,ID macro, process each incoming block, and then issue the WRITE,ID macro to update each block that requires it, or you would issue the READ,KEY macro followed by the WRITE,KEY macro. Remember the difference between the IDs returned after execution of the two different READ macros, and plan your use of the IDLOC field contents accordingly (Table 15—5). To equate the IDLOC and SEEKADR fields in update mode is not good practice.

Another point to keep in mind about updating a keyed file is that you must not only specify the length of keys with the KEYLEN keyword parameter in your DTF, but you must also provide a key for each updated block in your I/O area. Both the key and the data fields of the block on disk are updated when you specify this keyword.

Data management does not provide you a direct means of changing only the key of a block on disk. You may do so with the WRITE,ID macro only by presenting data management with a block in main storage that contains a new key field and a data portion that is identical to that already on disk. Of course, this can be done by reading in the whole block in question and updating only the key field before copying it all back to disk with the WRITE,ID macro. (You must not attempt this with the READ,KEY/WRITE,KEY macros, because with this pair you should not change the key of a block before returning it to the disk file.)

A massive update of a direct access file with the WRITE,ID macro may be made more efficient if the update information is presented in the physical order of the blocks that it is intended for on disk; this is especially true if the file has been created by using record interlace and a LACE factor tailored to the updating program (15.6.8). Having a keyed file precludes this possibility, because you cannot create a keyed file without specifying the KEYLEN parameter, and you cannot use record interlace if you do. In deciding whether to lace a direct access file that does not contain keys, you need, of course, to consider the additional costs of sorting your input file before using it in your update program.

WRITE,KEY

15.7.11.5. Rewriting Randomly Retrieved Blocks to Disk (WRITE,KEY)

When you want to effect a direct-access rewrite, or updating, of a block that you have just retrieved with the READ,KEY form of the READ command (15.7.14.2), you will use this form of the WRITE macro:

LABEL	Δ OPERATION Δ	OPERAND
[name]	WRITE	{ filename / (1) / 1 } , KEY

Here, the second positional parameter, KEY, specifies that the block just read by a READ,KEY macro is to be rewritten to its original location on disk. Both the key and the data field will be updated. Remember to specify WRITEKEY=YES in your DTF (15.6.35). You should also remember that if the new block is longer than the old, the new block will be truncated; if the new block is shorter, data management will pad out the original field with binary zeros. In either case, the *wrong length error* flag (byte 1, bit 5) will be set in *filenameC* (Appendix B). You should check this bit after issuing the WAITF macro that must follow each WRITE,KEY macro.

Another point to remember is that, because each block to be rewritten by the WRITE,KEY form must first be retrieved by the READ,KEY macro, consecutive issues of the WRITE,KEY form constitute a sequence error and will be flagged as an *invalid macro sequence* (byte 0, bit 6, of *filenameC*).

Because the WRITE,KEY macro does not conduct a search but relies on the search made by the preceding READ,KEY macro, it is not possible to add new records to a file with the WRITE,KEY macro alone. It is guided neither by the contents of the SEEKADR field nor by the contents of the KEYARG field. The only way the WRITE,KEY macro could be used to create a new file would be to originally create one whose keyed records contain blank or zero-filled data portions and then use the READ,KEY/WRITE,KEY combination to overwrite each null data portion with actual data. But this would be an inefficient and unnecessary way to go about creating a file.

If you want to store the ID of the block updated by the WRITE,KEY macro, you should specify the IDLOC keyword parameter in your DTF (15.6.7). The *form* in which this relative disk address is returned to you is governed by your specification of another DTF keyword parameter, RELATIVE, which is described in 15.6.22.

GET

15.7.12. Retrieving Records From Sequentially Processed Disk Files (GET)

You use the record-level GET imperative macro to obtain records of all types, one at a time, from DTFSD files or from sequentially processed DTFNI files, opened for input processing. Data management reads the records, automatically deblocks them if they are blocked, and delivers each unblocked or deblocked record to you. If you are processing in an I/O area, it places the records there, one at a time; if you are processing in a work area, it delivers the records there from the I/O area.

When you use a work area, you must remember to specify WORKA=YES in the DTF declarative macro (15.6.35). Because you specify the address or label of the work area to be used each time you issue the GET macro, you may use more than one work area, and a different one each time.

When your input records are blocked, and you have no work area, or if you provide two I/O areas, you must also provide an index register (using the IOREG keyword parameter of your DTF (15.6.11)), into which data management places the starting address (in the I/O area) of the next available record. An index register should not be used to keep track of the current *work* area, because you specify this with each GET macro; if you elect to use two or more work areas, it is important to use them consistently.

This is the format of the GET macro:

LABEL	ΔOPERATIONΔ	OPERAND
[name]	GET	{ filename | (1) | 1 } [, { workarea | (0) | 0 }]

Positional Parameter 1:

filename
Specifies the label of the DTFSD or sequentially processed DTFNI declarative macroinstruction.

(1) or 1
Indicates that you have preloaded register 1 with the address of the DTF file table.

Positional Parameter 2:

workarea
Is the label of an area into which data management moves the current record for you to process. (A different work area may be used for each GET macro.)

(0) or **0**
Indicates that you have preloaded register 0 with the address of a work area.

Omit Positional Parameter 2 when you do not use a work area.

If you specify only one I/O area in your DTF, and your input records are not blocked, you may access the data directly in the I/O area (IOAREA1). Otherwise, you must specify an index register with the IOREG keyword parameter or use a work area. Table 15—9 shows the uses of an index register and work areas, and the actions taken by data management under the various combinations possible.

Table 15—9. Use of IOREG Keyword Parameter for Processing Blocked or Unblocked Input Disk Files Sequentially with GET Macro

Number of I/O Areas		Number of Work Areas		Record Format		IOREG Specification Required	Data Management Action
1	2	0	>0	Blocked	Unblocked		
						YES	DMS uses IOREG to point to address of current record within the block contained in IOAREA1.
						NO	DMS delivers each unblocked record directly to user in IOAREA1.
						NO	DMS deblocks in IOAREA1 and delivers each deblocked record to workarea specified in positional parameter 2 of GET macro.
						NO	DMS delivers unblocked record from IOAREA1 to workarea specified in GET macro.
						YES	DMS deblocks in one IOAREA and delivers deblocked records to other IOAREA for user to process.
						YES	DMS delivers unblocked record to IOAREA specified by IOREG. Alternate areas are available for overlap processing.
						YES	DMS deblocks in one IOAREA and delivers deblocked records to workarea specified by GET macro. Other areas are available for overlap processing.
						YES	DMS delivers unblocked record to workarea specified by GET macro, from IOAREA specified by IOREG. Alternates are available for overlap processing.

When you are retrieving blocked input records with the GET macro, you may come to a point in the current block where you want to skip over the remaining records to process the first record of the succeeding block; the RELSE imperative macro is designed for this; see 15.7.13.

You have a different imperative macro, FEOV, available if you want to discontinue processing the current *volume* of an input DTFSD file in order to begin processing the next. This macro may not be used with sequentially processed DTFNI files, however; see 15.7.7.

RELSE

15.7.13. Skipping Records in Sequentially Processed Input Blocks (RELSE)

When you are processing blocked input records with the GET imperative macro from DTFSD files or sequentially processed DTFNI files, you may reach a point at which you want to skip over the remaining records in the current block and to commence processing with the first record of the succeeding block. To effect the skip, you issue the RELSE imperative macro; the next GET macro you issue makes the first record of the next block available to you (15.7.12).

The format of the RELSE macro is simply:

LABEL	Δ OPERATION Δ	OPERAND
[name]	RELSE	{ filename (1) 1 }

Positional Parameter 1:

filename
Is the label in your program of the DTFSD or DTFNI declarative macro defining the file you are processing sequentially.

(1) or **1**
Indicates that you have preloaded general register 1 with the address of the DTFSD or DTFNI file table.

When you want to terminate processing the current *volume* of a sequentially processed input file defined by the DTFSD declarative macro in order to begin with the next volume, you will use the FEOV imperative macro (15.7.7).

15.7.14. Random Retrieval from Direct Access Files (READ)

The READ imperative macro is a block-level input processing macro that, in its two forms and various uses, provides you with the following capabilities for randomly processing disk files defined as input files by the DTFDA and DTFNI declarative macros, or as input/output files by the DTFNI macro:

- retrieving a block or record by means of its relative disk address (ID), which you specify;
- retrieving a block by searching for its key, to be matched with a key you specify (the search begins at an address that you also specify); and
- updating a previously created file.

The READ macro causes a block to be retrieved from disk and to be read into main storage at an address specified by the IOAREA1 keyword parameter of your DTFDA or DTFNI declarative macro, when you specify only a single I/O area (15.6.10). On the other hand, when you are processing DTFNI files and have specified a second I/O area (IOAREA2; see 15.6.11), the main storage address is contained in the index register you must specify with the IOREG keyword parameter (15.6.11). (The DTFDA declarative macro, as you know, does not support either the IOAREA2 or the IOREG keyword parameter.)

Because the READ macro operates at block level, and OS/3 data management handles blocking and deblocking automatically only for *sequentially* processed files, you must control any deblocking necessary when you read blocked input files defined by the DTFNI macro. This deblocking you will take care of via the GET imperative macro (15.7.12), after the READ macro has placed a block into main storage from disk. As you know, you may not specify either of the blocked record formats for a DTFDA file. If your "unblocked" block in a DTFDA file actually contains more than one logical record, therefore, you must tend to the deblocking yourself. The GET macro is not supported for DTFDA files, however, and is flagged as invalid if issued to such a file. The best solution to this quandary is probably to avoid it by defining such a file with the DTFNI declarative macro in the first place.

Another important point to remember is that, after each READ macro you issue, you must issue a WAITF imperative macro before you issue any other imperative (15.7.16). This is necessary, to ensure that the intended transfer of the block from disk to main storage has indeed taken place.

If you want data management to store the relative disk address (ID) of the block retrieved by the READ macro, or of the next block — the two forms of the macro make different returns — you would specify the IDLOC keyword parameter of the DTF declarative macro (15.6.7). The form in which the ID is returned to you is governed by your specificaton of another DTF keyword parameter, RELATIVE, which you might also review (15.6.24). The uses of the IDLOC keyword parameter are further developed in what follows.

OS/3 data management supports the READ macro only for DTFNI files or for DTFDA files.

READ,ID

15.7.14.1. Random Retrieval of Records by Relative Disk Address (READ,ID)

In order to use the READ,ID form of the READ macro to retrieve a specific record by its relative disk address, or ID, you have a number of keyword parameters in the DTFDA or DTFNI declarative macro to consider. First of all, recall the uses of the IOAREA1 and IOREG keyword parameters (the latter is used with the DTFNI macro only) to specify the main storage address into which the block that contains the record will be read (15.6.10 and 15.6.11). You will also need to use the SEEKADR keyword parameter (15.6.26) to specify the 4-byte field in your program into which you will place the relative disk address of the record you want retrieved (which must always be greater than zero), and you must give notice to data management that you will be using the READ,ID form of the macro, by specifying the READ=YES keyword parameter in the DTF (15.6.18). Both the key of the block (if a key exists) and all data the block contains will always be read in. The form in which you provide the relative ID of the desired record must be the same as you specify with the RELATIVE keyword parameter.

Further, if you need data management to return to you the ID of the *next* block in the file or partition after you issue the READ,ID form of the macro, you would specify the *location* to which the ID is to be returned by the IDLOC keyword parameter (15.6.7). Finally, recall that the *form* in which the ID is returned has been specified by means of the RELATIVE keyword parameter (15.6.22).

When the logical record you want retrieved by ID from a DTFNI file lies within a block of records retrieved, data management reads the entire block into your I/O area. After your execution of the mandatory WAITF macro, data management returns the displacement of the desired record into the block (measured in hexadecimal bytes) to the DTF file table, right-justified in a 2-byte field designated as *filenameD*. You may address this field by concatenating the character "D" to your 7-character file name. To retrieve subsequent records contained in the same block, you may issue successive GET macros in the 2-parameter form to specify the work area into which data management is to move the current record (15.7.12). An important point here is that you must know the structure of your blocks: there is no end-of-block indication provided by data management to prevent you from going past the last record in a block you are processing in this way. A GET macro issued after you have processed the last record in a block causes the entire *next block* to be read in from the DTFNI file, and the first record from it to be moved to the specified work area.

The format of the READ,ID form of the READ macro is:

LABEL	Δ OPERATION Δ	OPERAND
[name]	READ	{ filename | (1) | 1 }, ID

Positional Parameter 1:

filename
Is the label in your program of the DTFDA or DTFNI declarative macro that defines the randomly processed file from which you are retrieving records.

(1) or **1**
Indicates that you have preloaded register 1 with the address of the DTF file table.

Positional Parameter 2:

ID
Specifies that a search is to be made for a record with an ID matching the relative disk address you have presented to data management, via the field specified in the SEEKADR keyword parameter of the DTF macro, in the form you have specified with the RELATIVE keyword parameter.

15.7.14.2. Direct Retrieval and Updating of Input Blocks by Key (READ,KEY)

You will use the READ,KEY form of the READ macro when you want to retrieve blocks by a search on key from input files defined by the DTFDA macro (15.5.2) or randomly processed files defined by the DTFNI macro (15.5.3). It is this form of the READ macro that is also used, in combination with the WRITE,KEY form of the WRITE macro, for updating a randomly processed disk file (15.7.11.5).

You have a number of points to consider when you are using the READ,KEY macro; these involve several keyword parameters in the DTF. First of all, in order that data management may set up the DTF file table properly for this macro, you must specify the READKEY keyword parameter. To guide the search, you must provide both a search argument and a starting point. With the KEYARG keyword, you specify the key that is to be matched by the key of the block you want retrieved (15.6.12); with the relative disk address that you place in the 4-byte field, whose label you specify with the SEEKADR keyword (15.6.24), you supply the starting point of the search. As to its end point, if you want the search to continue past the end of the current track to the end of the current cylinder, you specify the SRCHM keyword parameter; otherwise, the search ends at the end of the track (15.6.26).

Remember that the relative disk address you supply to data management must be in the form (relative track or relative record) that you selected when you specified the RELATIVE keyword parameter (15.6.22): the same form is used by data management when it makes the return of the relative disk address of the retrieved block to the 4-byte field whose label you specified with the IDLOC keyword (15.6.7).

If the search is successful, data management moves the entire block to your I/O buffer, including the key. Thus, the length of IOAREA1 (and of IOAREA2, if this is a DTFNI file and you have specified double-buffering) must accommodate the length of the key, as well as the length of your data record. If this is a DTFNI file containing variable-length blocked records, you have not only the KEYLEN specification and the block descriptor word (BDW) to keep in mind, but also the record descriptor words (RDWs) that precede each logical record. In a DTFDA file, you may not specify either of the blocked record formats, but a variable-length record in this type of file is also preceded by a BDW and an RDW, exactly as the variable, unblocked record is in the DTFNI file. A glance back at Figure 14—4 in the preceding section will help you visualize what is in your I/O area when the READ,KEY macro completes a successful search. Note that both the BDW and each RDW are four bytes long.

The data portion of a fixed, unblocked record from a DTFDA or DTFNI file begins at a displacement into your I/O area that is equal to your KEYLEN specification; the same is true of the first of your fixed-length logical records in the blocked format in DTFNI files; the succeeding records begin at intervals equal to your RECSIZE specification (15.6.21). When you have variable, unblocked records in either file, the first (or only) data portion is found at a displacement that is eight bytes longer than your KEYLEN specification; the displacement of each succeeding variable record in a blocked DTFNI file may be calculated by adding in the contents of the first two bytes of the RDW for the record preceding it. Figure 14—4 shows these relationships, which you must use in accessing your logical records from a block retrieved by the READ,KEY macro after a successful search.

If the search is unsuccessful, data management first sets the *record not found* flag (byte 1, bit 3) in *filenameC* and either the *end of track* (byte 1, bit 6) or both this flag and the *end of cylinder* flag (byte 1, bit 7), depending on whether or not you have specified SRCHM=YES in the DTF. Data Management then branches to your error routine (or returns control to you inline if you have no error routine specified). (Refer to Appendix B). Your error routine should check for these bits and provide action that is appropriate for your application; otherwise, if you accept error returns inline, you should test for these flags after each issue of the READ,KEY macro.

The format of the READ,KEY form of the READ macro is:

LABEL	Δ OPERATION Δ	OPERAND
[name]	READ	{ filename / (1) / 1 } , KEY

Here, the second positional parameter, KEY, specifies that data management will search for a block that has a key matching the one you have placed in the location in your program specified by the KEYARG keyword parameter. The search begins at the relative disk address you have placed in the location defined by the keyword parameter SEEKADR and continues to the end of the *one* track, unless you have specified the SRCHM keyword parameter.

The I/O area into which data is read will contain both the key and the data portions of the block read; data management moves the key to the I/O buffer.

CNTRL

15.7.15. Controlling Disk Head Movement to a Track (CNTRL)

The CNTRL imperative macro gives you a means by which you may overlap disk head movement with the processing of your records. While it may be used when you are processing files defined by the DTFDA, DTFSD, or DTFNI macros, the CNTRL macro is perhaps most useful for achieving some increase in throughput when you are sequentially processing files on a *shareable* disk volume. (A shared volume is one that may be accessed by more than one user program in a multiprogramming environment at your installation. It is described in the device assignment set by a VOL job control statement which has "S" as its second positional parameter.)

When you issue the CNTRL macro to a DTFSD file, data management issues a seek command that positions the disk head to the track you are currently processing. When you are processing a DTFDA or DTFNI file, on the other hand, you may issue the CNTRL macro to reposition the disk head to a *new* relative track address, which you place in the location specified by the SEEKADR keyword parameter of your DTF (15.6.26).

When your program and another are sharing the same disk volume, issuing CNTRL may increase your throughput by positioning the disk head to the specified track while you are processing your records. Any CNTRL macro you issue to a blocked file is ignored, and data management will wait until the block is finished. (Of course, this feature protects you from interrupting your own block processing prematurely, as well.)

When there is no problem of seek contention among programs (as when you are processing a nonshared volume), using the CNTRL macro may still increase your throughput and save you overall processing time. If you move the new current track address into the location specified by SEEKADR after successful completion of a READ or WRITE operation, for example, and then issue the CNTRL macro before you execute your other record processing instructions, the disk head is repositioned during the time you are processing, and data management can more promptly execute the next input or output imperative macro you issue.

It should be clear from the foregoing discussion what the position of the WAITF macro must be that you will always issue after a READ or WRITE macro to randomly processed files (15.7.16): it must follow this macro and *precede* the CNTRL macro. To prevent you from repositioning the disk head before you learned of an incomplete input/output operation, data management will set the *WAITF required* flag (byte 0, bit 7 of *filenameC*) after determining that you have failed to issue the WAITF macro before executing the CNTRL imperative macro after a READ or WRITE macro. (The WAITF macro is not required after the CNTRL macro; the CNTRL macro makes no return to the IDLOC field of your program.)

This is the format of the CNTRL macro used for processing all nonindexed disk files:

LABEL	Δ OPERATION Δ	OPERAND
[name]	CNTRL	{ filename / (1) / 1 } , SEEK

The second positional parameter, SEEK, is *always* required; it specifies that data management will issue a seek command to the disk head, positioning it for DTFSD files to the current track or, for DTFDA or DTFNI files, to the relative track address contained in the 4-byte location whose label you have specified with the SEEKADR keyword parameter. (The address at this location should be given in the form *Ott*, where *tt* is the relative track number and 0 is binary 0; it must be left-justified in the SEEKADR field, and you must have specified RELATIVE=T in the DTF for the file.) In the first positional parameter, *filename* is the label in your program for the DTFSD, DTFDA, or DTFNI declarative macro defining the file; *(1)* or *1* indicates that you have preloaded register 1 with the address of this DTF file table.

WAITF

15.7.16. Waiting on Completion of I/O to Random Disk Files (WAITF)

When you are randomly processing input or output disk files defined by the DTFDA or DTFNI declarative macros, using the WAITF macro is a compulsory safety measure to ensure that the data transfer initiated by a READ or WRITE macro is completed before you issue another imperative. You must issue the WAITF macro following each READ macro (15.7.14) or WRITE macro (15.7.11) you execute, before you may issue a CNTRL macro or another READ or WRITE macro for the same file or partition. The WAITF macro is not required following the CNTRL macro.

When data management detects that you have omitted this mandatory macro, it sets the *WAITF required* error flag (byte 0, bit 7 of *filenameC*), and transfers control to your error routine, if you have specified one, or to you inline.

The WAITF macro itself acts to check the completion of the data transfer you intended, and to set the appropriate status or error bits in the status code fields of *filenameC*. These bits you must always check after the execution of the WAITF macro — it is pointless to anticipate the execution of the WAITF macro by checking *filenameC* immediately after you issue the READ or WRITE macro, although perfectly legitimate to check it when you are processing sequentially with the PUT macro and GET macro. (You do *not* use the WAITF macro with these sequential input/output processing imperatives.)

The WAITF macro is not involved in the parity checking of output records; this is performed as a separate operation by data management only when you specify the optional VERIFY keyword parameter in your DTF (15.6.32).

The format of the WAITF macro is simply:

LABEL	Δ OPERATION Δ	OPERAND
[name]	WAITF	{ filename (1) 1 }

As elsewhere, *filename* is the label in your program of the DTFDA or randomly processed DTFNI declarative macro; *(1)* or *1* indicates that you have previously loaded general register 1 with the address of the corresponding DTF file table.

NOTE

15.7.17. Accessing the Current Relative Block Address (NOTE)

The NOTE macro, which you may use only with DTFNI files, is one of the extended capabilities OS/3 data management provides for processing nonindexed disk files. You may use it, whether you are processing DTFNI files randomly or sequentially, to access the relative address of the current block or record. The NOTE macro is used in conjunction with the POINT macro (15.7.18), where a coding example is given.

When you are processing sequentially, after issuing a GET or PUT macro, you use the NOTE macro to access the relative address of the current block and the displacement of the current record within this block. Because the addresses it returns are *partition-relative*, the NOTE macro relies upon your having selected the partition previously by issuing the SETP macro (15.7.4) and, if you are working within a subfile of this partition, having positioned yourself to this with the SETS macro (15.7.5).

For sequentially processed DTFNI files, the NOTE macro returns the address to a 6-byte field of the DTF file table in discontinuous binary in the form:

0bbbdd

where:

0bbb
Is the number of the current block, relative to the first block in the file, *0* being binary 0.

dd
Is the displacement, measured in hexadecimal bytes, of the current record within that block.

For randomly processed DTFNI files, you issue the NOTE macro following a READ or WRITE macro. The 6-byte address is in discontinuous binary, and the form of the address is governed by what you have specified for the RELATIVE keyword parameter in your DTF (15.6.22). If you have specified RELATIVE=R, the form is:

rrrdd

where:

rrr
Is the *relative record address* of the current block.

dd
Is binary 0.

If you have specified RELATIVE=T, on the other hand, the address returned is in the form:

0ttrdd

where:

0ttr
Is the *relative track address* of the current block; that is, *r* is the number of the block on the relative track denoted by *0tt*, *0* being binary 0.

dd
Again, is binary 0.

You need to use the SETP and SETS macros for randomly processed files also, as mentioned in the previous paragraph, to pre-position yourself to the correct subfile of the correct partition.

The NOTE macro returns the address you have specified to a 6-byte field in the DTFNI or DPCA file table that you address by concatenating the character "B" to your 7-character filename or partition name. It is important to remember that you must address this field as *filenameB* whether you are processing the first partition (PCA1) of a partitioned DTFNI file or are processing a nonpartitioned file. It is only when you are working in the other partitions (PCA2 through PCA7), that you address this field as *partitionnameB*.

This is the form of the NOTE macro:

LABEL	Δ OPERATION Δ	OPERAND
[name]	NOTE	{ filename (1) 1 }

Positional Parameter 1:

filename
Is the label in your program of the corresponding DTFNI macroinstruction. The partition name is never used.

(1) or **1**
Indicates that you have preloaded register 1 with the address of the DTFNI file table.

Remember that all returned addresses are *partition-relative*; you must issue the appropriate SETP/SETS macros before issuing the NOTE macro (15.7.4 and 15.7.5).

POINT

15.7.18. Positioning a File or Partition to a Relative Block Address (POINT)

The POINT imperative macro is your means of randomly positioning a sequentially processed DTFNI file or partition to a relative block address. This address may be obtained, for example, via the NOTE macro just described (15.7.17). Like NOTE, the POINT macro is a useful extension of capabilities (OS/3 data management provides for processing your DTFNI files; it is not supported for files you define by the DTFDA or DTFSD macros. Another similarity to the NOTE macro is that the POINT macro also relies on you to have selected your partition via the SETP macro (15.7.4) before you issue it. A third similarity is that the form of the relative block address used by the POINT macro is the same form the NOTE macro uses.

When you issue the POINT macro for a sequentially processed DTFNI file, data management modifies the current partition-relative block address and block displacement that it maintains in the DTF file table (or in the DPCA partition table, if you are processing in a partition); you are subsequently processing from the new address.

An important point to remember is that the POINT macro is effective only so long as you continue to use the GET and PUT macros in your subsequent processing of the file or partition. If you subsequently issue a READ or WRITE macro (which would be quite legitimate, for these are DTFNI files), these macros cause the current partition-relative block address to be modified by the relative track or record address you will have supplied to data management in the area specified by the SEEKADR keyword parameter of your DTF (15.6.24). There is no error indication returned to you when this occurs; you must rely upon yourself to keep this point in mind.

This is the format of the POINT macro:

LABEL	Δ OPERATION Δ	OPERAND
[name]	POINT	{filename / (1) / 1} , {address-field / (0) / 0}

Positional Parameter 1:

filename
Is the label in your program of the corresponding DTFNI macroinstruction.

(1) or **1**
Indicates that you have preloaded register 1 with the address of the DTFNI file table.

Positonal Parameter 2:

address-field
Is the label of a 6-byte field (0bbbdd) in your program containing the relative block address and displacement of the record within the block. The relative block address portion (0bbb) is right-justified in the first four bytes, and the record displacement (dd) is right-justified in the second two.

(0) or **0**
Indicates that you have preloaded register 0 with the address of the 6-byte address field.

Remember that, when you are processing within a partition, or a subfile of a partition, you must have preselected these by issuing the appropriate SETP and SETS macros (15.7.4 and 15.7.5). For this reason, the partition name is never used as an operand of the POINT macro.

The coding example that follows shows the use of a NOTE macro to access the current relative address of a sequentially processed output file, FILE1, whenever a record containing a desired value, 'VALU', in its first four positions is encountered. This address is then used by a POINT macro in a selective input or updating of FILE1, in what amounts to *random* processing in a *sequential* file. FILE2 is used as an index to FILE1 for this processing.

Example:

	LABEL (1)	OPERATION (10)	OPERAND (16)
1.	STEP1	OPEN	FILE1,FILE2
		.	
		.	
		.	
	OUT	PUT	FILE1,WORKA1
2.		CLC	WORKA1(4),DESIRE
		BNE	OUT
3.		NOTE	FILE1
		LA	0,FILE1B
4.		PUT	FILE2,(0)
		B	OUT
		.	
		.	
		.	
		CLOSE	FILE1,FILE2
		.	
		.	
		.	

Example (cont):

```
   LABEL     ΔOPERATIONΔ          OPERAND                    Δ
   1         10        16
5. STEP2     OPEN      FILE1,FILE2
                 .
                 .
                 .
6. IN        GET       FILE2,WORKA2
7.           POINT     FILE1,WORKA2
8.           GET       FILE1,WORKA1
                 .
                 .
                 .
             PUT       FILE1,WORKA1
             B         IN
                 .
                 .
                 .
9. DESIRE    DC        CL'VALU'
```

1. FILE1 and FILE2, both DTFNI files, are opened for sequential output processing.

2. Record output to FILE1 is tested for desired value, 'VALU'.

3. NOTE macro is issued when 'VALU' found; relative address is returned to *filenameB*.

4. Relative address found by NOTE is output to FILE2, which will serve as an index in STEP2.

5. FILE1 is reopened for updating; FILE2 for input processing (assume file processing directions have been reset).

6. Record retrieved from FILE2 is the address of a record in FILE1 containing 'VALU'.

7. POINT macro is issued to position FILE1 to address obtained in 6.

8. Sequential processing continues in FILE1 from position set in 7.

9. Definition of the desired value, normally located outside of executable code.

15.8. ERROR AND EXCEPTION HANDLING

15.8.1. FilenameC

When certain errors or exceptions to file processing performance are detected by OS/3 data management, it will make appropriate entries in specific fields of the DTF file table, which your program may address in order to learn of these conditions and take the proper course of action on regaining control. One such field is *filenameC* (in the nonindexed file processor system a 4-byte field), which you may access by concatenating the character "C" to your 7-character logical filename.

A point to remember when you are processing the partitions of a DTFNI file is that, just as your ERROR routine is a file-relative specification and belongs in your DTFNI declarative macro (not in the DPCA declarative macro defining the partition), you do not have a field for error flags in the partition control appendage established in the DPCA macro. That is, there is no "partition-nameC"; error and status flags set during processing of partitions are set in the field of the DTFNI file table and are accessed, therefore, as *filenameC*.

Most of the error and status flags have already been discussed in preceding paragraphs; refer to Table B—3 for the meanings of the bits in *filenameC* of the DTFSD, DTFDA, and DTFNI filetables that are set to binary 1 by OS/3 data management for certain error and exception conditions.

Not all of the status flags represent conditions causing transfer of control to your error routine. Some of these must be tested for *inline* in your program if you want to act upon them.

16. System Resource Control

16.1. DEVICE ALLOCATION AND FILE ASSIGNMENT

In OS/3, the supervisor and job control have the essential responsibility for reserving main storage and allocating peripheral devices for jobs that appear in the control stream and make their needs known through job control statements following the JOB statement. Of these statements, DVC, VOL, EXT, LBL, and LFD are essential for providing information relating to your files. With proper use of these statements you may reserve peripheral devices and identify and assign to your program the files you have on them or will place on them.

16.1.1. Use of Job Control Statements

Every file that you intend to reference in your program must be represented in the job control stream by a set of control statements, called the *device assignment set*, which contains at least a DVC statement followed by an LFD statement. Between these basic two, you will need to insert as many as six other statements for your magnetic tape and disk files: the VOL, EXT, LBL, LCB, VFB, and DD statements. The device assignment set specifies the relationships between your files or volumes and the peripheral devices; there is one set for each file. Following is a short summary of the functions of these statements; all are described in the job control user guide, UP-8065 (current version):

- The DVC statement assigns peripheral devices to your job.
- The VOL statement specifies the magnetic tape or disk file volumes to be accessed by the job.
- The EXT statement establishes new disk files or extends existing files on disk.
- The LCB statement specifies and loads to the printer a unique load code buffer that overrides the LCB set at SYSGEN time.
- The VFB statement specifies and loads a unique vertical format buffer that overrides the VFB set at SYSGEN time.
- The DD statement modifies fields within the DTF file table at run time and avoids recompiling DTF's when changes in DTF specifications are required.

- The LBL statement supplies label information for magnetic tape or disk files.
- The LFD statement identifies the file control block for each file used in your job.

Card and paper tape files use only the DVC and LFD statements and optionally the DD statement. The printer always uses the DVC and LFD statements and optionally the LCB, VFB, and DD statements.

16.1.2. Sample Device Assignment Set

Following is an example of a set of control statements that you would use for reserving space for a sequential disk file that is to be created:

```
LABEL      ΔOPERATIONΔ          OPERAND                    Δ
1          10        16
// DVC 50
// VOL 124365
// EXT BQ,C,5,CYL,10
// LBL 'INVENTORY MASTER FILE'
// LFD INVNTRY,1
```

This device assignment set will cause modification of the volume table of contents (VTOC) of disk pack 124365 to show that a sequential file called 'INVENTORY MASTER FILE' exists, and that the space reserved for it occupies a specific position on the pack. OS/3 job control will also note that your program will address this file as 'INVNTRY' (its logical name), and that space for one extent entry in the prologue area will suffice. To start with, the file will be assigned 10 cylinders of contiguous space. When this space is exhausted, you desire automatic extension, five cylinders at a time.

When space for only one extent in the prologue area is specified, no dynamic extension can take place.

For a program that will subsequently operate on this file, you would use the same set of job control statements, except that you would omit the EXT statement. If you want to recreate a file (for example, if an error occurred during your first attempt to create it), you may use the INIT positional parameter of the LFD statement. This parameter has the same effect as scratching the file and reallocating it to the same area as it previously occupied on disk.

16.1.3. Job Control Deallocation Statement (SCR)

The OS/3 job control language also provides an SCR statement that you may use to deallocate (scratch) a file from a disk pack. When you use this statement, you must use an LFD name that is the same as the one you have used for the file in a preceding valid device assignment set (DVC-LFD). The SCR statement will deallocate the file and its extents before the program named in the next EXEC statement in the control stream is executed; therefore, you may use it to free disk space needed for a subsequent job or job

step. (The disk space management facilities of the OS/3 supervisor also include an imperative macro, SCRTCH, that provides similar functions and is used for dynamic deallocation (16.3).)

16.1.4. Using the File Lock Feature

The OS/3 file lock feature allows you to control the sharability of a file while you are using it. Sharability control only applies to lockable files. To use the file lock feature, proceed as follows:

- Step 1

  At system generation, you specify the FILELOCK parameter to indicate which files are lockable.

- Step 2

  In your file definition (within your BAL program) and in the device assignment set for the file (regardless of the program type), you specify your read/write requirements and indicate whether other jobs or tasks within a job can share the file.

In the following paragraphs we will describe how to indicate which files can be locked and how to set the various degrees of sharability.

16.1.4.1. Indicating Which Files are Lockable

You indicate which files are lockable by using the FILELOCK parameter in the SUPGEN section of the parameter processor at system generation time.

If you choose FILELOCK=NO or omit the parameter, only the system files prefixed with Y are lockable. No user files can be locked.

If you choose FILELOCK=YES, all system files (prefixed with Y) and all files prefixed with $LOK01-$LOK99 are lockable.

If you choose FILELOCK=SHARE, all files are lockable.

16.1.4.2. Setting File Locks for Data Files in BAL Programs

After you specify which files are lockable, you specify the degree of sharability for each of these files.

If you choose FILELOCK=YES at system generation time, you can lock any file whose filename you prefixed with $LOK01-$LOK99 in the // LBL job control statement in the device assignment set. If you do nothing more, any prefixed file will be exclusively locked when it is opened during the execution of your program. You have exclusive use of the file. You can read, update, and add to the file. No other user can open the file until you close it.

If you choose FILELOCK=SHARE at system generation time, you can lock all of your files. If you do nothing more, each file will be exclusively locked when it is opened during the execution of your program. You have exclusive use of the file. You can read, update, and add to the file. No other user can access the file until you close it.

In both cases (FILELOCK=YES or FILELOCK=SHARE specified at system generation), you can override this lock by specifying one of the options of the ACCESS parameter or by specifying the LOCK=NO parameter in the DTF macroinstruction for a file. (See 11.4.1 and 11.4.11.)

You can also override this lock at program execution time in two ways. The first way is to include a // DD job control statement that specifies one of the ACCESS parameter options in your device assignment set. The second is to prefix the filename with an asterisk (*) in the // LFD job control statement in the device assignment set. This will cause a read-only lock to be applied to the file; that is, you can only read from the file and all other users can only read from the file.

NOTE:

To set file locks on SAT files, see the supervisor user guide, UP-8075 (current version).

16.1.4.3. Setting File Locks for Data Files in Non-BAL Programs

After you specify which files are lockable, you specify the degree of sharability for each of these files.

If you choose FILELOCK=YES at system generation time, you can lock any file whose filename is prefixed with $LOK01-$LOK99 in the // LBL job control statement in the device assignment set. If you do nothing more, any prefixed file will be exclusively locked when it is opened in your program. You have exclusive use of the file. You can read, update, or add to the file. No other user can access the file until you close it.

If you choose FILELOCK=SHARE at system generation time, you can lock all of your files. If you do nothing more, each file will be exclusively locked when it is opened in your program. You have exclusive use of the file. You can read, update, or add to the file. No other user can access the file until you close it.

In both cases (FILELOCK=YES or FILELOCK=SHARE specified at system generation) you can override this lock at program execution time. There are two ways to do this. The first way is to include a // DD job control statement that specifies one of the ACCESS parameter options in the device assignment set. The second is to prefix the filename with an asterisk (*) in the // LFD job control statement in the device assignment set. This will cause a read-only lock to be applied to the file; that is, you can only read from the file and all other users can only read from the file.

16.1.4.4. File Lock Feature Summary

Table 16-1 summarizes the data management file lock feature. Remember, before using this feature, you indicated which files are lockable at system generation time through the FILELOCK keyword parameter. Once again, the available options are:

- FILELOCK=NO indicates that only the system files prefixed with Y are lockable. No user files can be locked.
- FILELOCK=YES indicates that all system files (prefixed with Y) and all files prefixed with $LOK01-$LOK99 are lockable.
- FILELOCK=SHARE indicates that all files are lockable.

Table 16—1. File Lock Summary

LOCK Keyword	Action	ACCESS Keyword	Action
LOCK=NO ① not specified	This DTF: read use/ update use/add use Other jobs: no access	ACCESS=EXC ①	This DTF: read use/ update use/add use Other jobs: no access
LOCK=NO ②	This DTF: read use Other jobs: read use	ACCESS=EXCR	This DTF: read use/ update use/add use Other jobs: read use (ACCESS=SRD specified for other jobs)
		ACCESS=SRDO ②	This DTF: read use Other jobs: read use (ACCESS=SRD or SRDO specified for other jobs)
		ACCESS=SRD	This DTF: read use Other jobs: read use/update use/ add use (ACCESS=EXCR, SRD, or SRDO specified for other jobs)

NOTES:

① LOCK=NO not specified and ACCESS=EXC are functionally equivalent.

② LOCK=NO and ACCESS=SRDO are functionally equivalent.

Remember, files are locked based upon the physical file name or $LOKnn prefix for the name that you specified in the // LBL job control statement of the device assignment set. The volume serial number is not used. Since this is the case, you should use a unique physical file name or $LOKnn prefix to differentiate between unrelated files or file sets on different volumes.

If you specify the same physical file name or $LOKnn prefix for unrelated files, you risk having files locked out unnecessarily when the ACCESS options are not compatible. For example, assume that files A and B are unrelated and are on different volumes. Also assume that these files have the same physical file name on the // LBL job control statement in their respective device assignment sets. If file A is lockable and has been opened for exclusive use with JOB1, no other job can open file B because its physical file name has already been locked to JOB1 even though file A is an unrelated/different file. Using unique file names prevents this from happening.

RENAME

16.2. RENAMING A DISK FILE (RENAME)

Neither OS/3 job control nor data management provides a means for renaming an existing disk file. If you need to change the name of one of your files (as recorded in the 44-byte *file ID* field of the format 1 label on disk), you must do it dynamically within your program, using the RENAME imperative macroinstruction provided by the disk space management facilities of the OS/3 supervisor. Note that you should not issue the RENAME macro to a file that is currrently open.

Function:

The disk space management macro RENAME allows you to rename any disk file but a system file. By specifying the new 44-byte file identifier you want used, the 7-byte logical file name, and the volume sequence number of the volume on which the file resides, you cause the new file identifier to replace the old file ID in the format 1 label of the VTOC. (The 7-byte logical file name is the same as that appearing as the first operand of an LFD job control statement for the file and in the label field of the corresponding data management DTF declarative macro.)

Format:

LABEL	ΔOPERATION Δ	OPERAND
[name]	RENAME	{param-list | (1)} [, {error-addr | (r)}] [,vol-seq-no]

Positional Parameter 1:

param-list
Specifies the address of a parameter list containing a 7-byte filename (as listed on the LFD job control statement and in the label field of the corresponding DTF macro) and a 44-byte new file identifier. The parameter list is a 52-byte character string, the first eight bytes of which contain the *LFD-name* of the file, left-justified; the last 44 bytes contain the new *file ID*, also left-justified.

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

Positional Parameter 2:

error-addr
Symbolic address to which control is transferred if an error is encountered.

(r)
Indicates that a register (2 through 12) has been preloaded with the error address.

Positional Parameter 3:

vol-seq-no
Specifies the volume of a multivolume file to be renamed.

If omitted, the value 1 is assumed.

Examples:

```
LABEL     ΔOPERATIONΔ        OPERAND                    Δ
1         10        16
          RENAME    (1),(12)
          RENAME    PLIST,,2
```

SCRTCH

16.3. DYNAMIC DEALLOCATION OF A DISK FILE (SCRTCH)

Function:

The disk space management macro SCRTCH enables you to deallocate any disk file but a system file, making the space available for future use. After validating your request, the SCRTCH macro removes the definition of the space or extents from the format 1 or format 3 labels and updates the format 5 label in the VTOC. The extent of the newly available freed space is inserted in the correct position in the format 5 record, where available extents are described in ascending sequence of relative track addresses. If you deallocate your file, the SCRTCH macro deletes all format 1, 2, and 3 labels from the VTOC and replaces them with format 0 labels. Note that you should not issue the SCRTCH macro to a file that is currently open.

Three basic deallocation (scratch) functions are available to you:

- deallocation of files by prefix;
- deallocation of files by expiration date; and
- complete deallocation of a file by file ID.

To deallocate files by prefix, you place a 4-byte prefix in bytes 76—79 of the file control block (FCB) of the file in main storage and specify the PREFIX parameter; the SCRTCH macro searches the VTOC for files with file ID fields beginning with these four characters and deallocates each one matched. The prefix may not include the characters Y, so that it is not possible to you to scratch system files by mistake. But by this macro you may deallocate all your temporary work files with one call on OS/3 disk space management.

To deallocate expired files, you specify the ALL parameter and include the expiration date in the 3-byte *expiration date* field of the FCB; the SCRTCH macro compares this with the expiration date in each format 1 label in the VTOC and deallocates all files with earlier expiration dates.

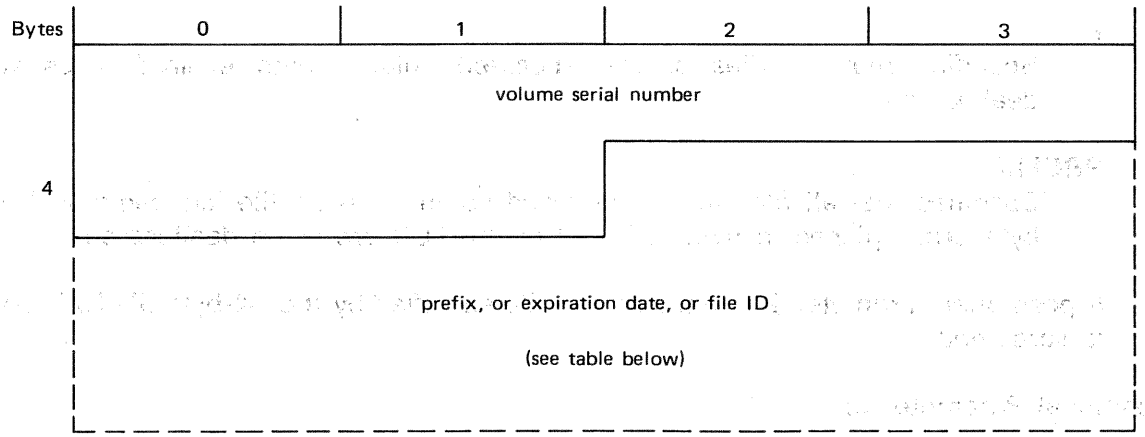
To deallocate an entire file by the 44-byte file ID that is contained in the FCB, you either omit the second positional parameter altogether, or specify (0) and preload bits 0 through 7 of general register 0 with the hexadecimal code 00.

Format:

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

Positional Parameter 1:

FCB-name
Specifies the symbolic address of the FCB in main storage.



Bytes	Contents
0—5	Volume serial number (VSN) of disk pack on which files to be deallocated reside
6—n	*One* of the following:

- 4-byte prefix (may not contain Y); requires specification of PREFIX in positional parameter 2
- 3-byte expiration date, in discontinuous binary in the form YDD (year, day, day), where Y ranges from 0 to 99 and DD ranges from 1 to 366; requires specification of ALL in positional parameter 2
- 44-byte file identification name; requires omission of positional parameter 2

(1)
Indicates that you have preloaded general register 1 with the symbolic address of the FCB in main storage.

Positional Parameter 2:

(0)
Indicates that you have preloaded register 0 with a hexadecimal function code specifying the scratch operation desired, as follows:

Function Code	Interpretation
00	Scratch entire file.
82	Scratch all files of the volume whose expiration date is exceeded by the content of the 3-byte *expiration-date* field of the FCB.
83	Scratch all files that have the 4-byte prefix contained in bytes 76—79 of the FCB.

ALL
Specifies that all files of the specified volume with expired dates will be deallocated.

PREFIX
Specifies that all files of the specified volume whose file IDs begin with the 4-byte prefix placed in bytes 76—79 of the FCB are to be deallocated.

If positional parameter 2 is omitted, the file specified by the 44-byte file ID in the FCB is scratched.

Positional Parameter 3:

error-addr
Specifies the symbolic address of your error routine, to which control will be transferred if an error is encountered.

(r)
Indicates that you have preloaded the specified register with the address of your error routine. Register 0 and 1 cannot be specified.

Examples:

	LABEL (1)	OPERATION (10)	OPERAND (16)
1.		SCRTCH	MYFLE,ALL,ERRXT
2.		SCRTCH	TRIFLE,,(10)

1. This macro deallocates all files whose expiration dates are exceeded by the contents of the 3-byte *expiration date* field of the FCB whose symbolic address is MYFLE. Your error routines' symbolic address is ERRXT.

2. This macro scratches the file whose 44-byte file ID is contained in the FCB whose symbolic address is TRIFLE. You have preloaded general register 10 with the address of your error routine.

16.4. DISK SPACE MANAGEMENT AND THE VTOC

We have discussed two disk space management routines that update the VTOC of a disk pack for you. All together, there are five OS/3 disk space management routines providing vital services for maintaining a correct VTOC on every disk pack. Two of these transients (ALLOC and EXTEND) you will not use directly because they are invoked automatically for you, when you need them, by OS/3 data management or job control; they are therefore not documented here but in the supervisor user guide, UP-8075 (current version).

Because the disk space management routines provide efficient, completely automatic space accounting and maintenance, they relieve you of the burden of keeping precise track of the contents of your disk files. An understanding of the structure of the VTOC and the format of its label records is not essential to you as a data management user; however, Appendix D describes the VTOC in full detail for your information, and the disk space management macro OBTAIN is available for the rare occasions you will have to examine it from a program.

OBTAIN

16.4.1. Retrieving VTOC Information (OBTAIN)

Function:

The disk space management macro OBTAIN returns to you either the disk address or the contents of a specified VTOC format label record for a specified disk volume. You must specify a return buffer of a size appropriate for the information you want retrieved.

Format:

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

Positional Parameter 1:

param-list
Is the address of a 12-byte parameter list containing the following:

Bytes 0—7 7-byte logical filename (as listed in the LFD Job control statement)

Byte 8 Hexadecimal code of the retrieval function to be performed on the VTOC of the disk volume whose volume sequence number is specified by positional parameter 3

Code	Function
00	Return VOL1 disk address
01	Return format 1 disk address
02	Return format 2 disk address
03	Return format 3 disk address
04	Return format 4 disk address
05	Return format 5 disk address
06	Return format 6 disk address
80	Return contents of VOL1 label

Code	Function
81	Return contents of format 1 label
82	Return contents of format 2 label
83	Return contents of format 3 label
84	Return contents of format 4 label
85	Return contents of format 5 label
86	Return contents of format 6 label
87	Return contents of the label record that is located at the disk address that has been preloaded into the first work of the return buffer.

Bytes 9—11 Return buffer address. Specifies the address in main storage of a return buffer. The OBTAIN macro places the disk address or the contents of the specified format label in the return buffer specified by this field. All disk addresses returned by the OBTAIN macro are stored in bytes 0 through 3 of this buffer, in the form 0 ccc hh rr, in discontinuous binary. If you specify function code 87, you must store the disk address of the format label you want retrieved in the same location in this buffer, in the same form and format.

The size of your return buffer must be four bytes when you request the return of an address, 84 bytes when you request the contents of the VOL1 label, and 140 bytes when you request the contents of a disk format label record.

(1)

Indicates that you have preloaded register 1 with the address of the parameter list.

Positional Parameter 2:

error-addr

Symbolic address of your error routine, to which control is transferred when an error is encountered.

(r)

Indicates that you have preloaded the specified register with the address of your error routine. Registers 0 and 1 may not be specified.

Positional Parameter 3:

vol-seq-no
Specifies the volume sequence number of that volume of a multivolume file from which you want to retrieve VTOC information.

If omitted, a value of 1 is assumed.

Examples:

```
   LABEL      ΔOPERATIONΔ          OPERAND                    Δ
   1          10        16
1. ......     OBTAIN    (1),(4),2
2. ......     OBTAIN    RECOVER1,ERRRTN
```

1. You have preloaded register 1 with the address of your parameter list and register 4 with the address of your error routine; you are retrieving information from the VTOC of the second volume of a multivolume file.

2. This example is seeking information from the VTOC of the first volume of a multivolume file; the error routine and parameter list are specified by their symbolic addresses, ERRRTN and RECOVER1, respectively.

16.4.1.1. Retrieving Specific Format Label Items

Once you have retrieved the desired format label with the OBTAIN macro, you may address a specific field by its individual label; these labels are shown in Appendix D.

PART 5. PAPER TAPE FILES

17. Paper Tape Data Management

17.1. GENERAL

This section describes OS/3 paper tape data management, a system that provides the basic assembly language (BAL) programmer with access at the logical-record level to the SPERRY UNIVAC 0920 Paper Tape Subsystem. The operational characteristics of the latter are outlined in Table A—6; for further information, however, refer to the 0920 paper tape subsystem programmer reference, UP-7998 (current version).

After a brief discussion of the hardware and paper tape itself, this section describes the effects of various character and record types on the modes of processing paper tape files and gives an overview of programming with paper tape data management. Following this overview, the four file processing imperative functions OPEN, CLOSE, GET, and PUT are explained, with a description of the means available to you as a BAL programmer in OS/3 for including the paper tape data management processing modules with your own code.

The matter of defining a paper tape file to data management, using the keyword parameters of the DTFPT declarative macro to specify its characteristics and your requirements for processing it, is then presented in full detail, with a number of simple coding examples to clarify the use of shift code scan tables and translation tables. The discussion of file definition closes with a description of data management error processing for paper tape files and the use of scan and translation tables when processing paper tapes in ASCII (American National Standard Code for Information Interchange).

This section concludes with a few notes on the compatibility of OS/3 with the paper tape data management of other operating systems.

17.2. HARDWARE AND PAPER TAPE CONSIDERATIONS

The data management paper tape system in OS/3 is designed to be used with the 0920 paper tape subsystem, which can be configured as a paper tape reader, a paper tape punch, or a combined reader and punch.

It is important to note that, when the subsystem is being used as a combined reader and punch, there are two separate paper tape paths. Whereas reading and punching can take place at the same *time* in such a subsystem, they cannot take place in one pass on the same *tape*. Two different passes are necessary to read and punch on the same piece of paper tape, and in OS/3 the tape must be defined with a separate DTF for each such pass: once as an input file, once for output, using different file names. In addition, you require separate job control device assignment sets (of DVC-LFD statements).

As a reader, the 0920 paper tape subsystem can handle three different widths of tape: 11/16, 7/8, and 1 inch. The subsystem can punch two widths: 11/16 and 1 inch. As to the number of tracks, or levels of tape characters, the subsystem can read or punch 5-level tape on the 11/16-inch width and from 5 through 8 levels on the 1-inch width.

The subsystem runs in two reading and punching modes: binary and character. In the binary mode, which is used only with 8-level (1-inch) tapes, there is a fixed, direct correspondence between bits in main storage and holes punched on the tape, that cannot be altered by rewiring the program connector board (17.2.1).

In the character mode (also called standard, or nonbinary, mode) from 5- to 7-level tapes are read and punched; an even or odd parity signal may be punched on the tape and checked during reading. It is not transferred to main storage, but, if a parity error is detected (this is always done by the hardware), the most significant bit of the byte in main storage representing the character is set to 1.

In addition, you may set up the program connector board to allow detection of a stop character (often called the "wired stop character" because of the clip-on wires used), as well as a delete character.

17.2.1. The Program Connector Board

For control, the 0920 paper tape subsystem uses the *program connector*, a printed circuit patch card which you set up with clip-on wires for the specific tape you are reading or punching. You will have much need of the program connector for processing in character mode, but it is bypassed completely for processing in binary mode. A short summary to highlight the procedure for wiring the program connector follows; for full details, consult the 0920 paper tape subsystem programmer reference, UP-7998 (current version).

17.2.1.1. Wiring the Program Connector for the Tape Punch

There are eight punch actuator circuits, each connected to a pin on the program connector board. In addition, there is a pin on the board for each of the seven least significant bits of a byte in main storage, and two pins that generate odd or even parity signals, based on the content of all eight bits of the byte in main storage. You connect the actuator pins to the main storage bit pins by the clip-on wires; any actuator pin can be connected to any bit pin. You can connect either parity signal pin (the odd or the even) to one of the actuator pins, or you can ground the unused actuator pins together so that nothing is punched in their tracks.

17.2.1.2. Wiring the Program Connector for the Tape Reader

A pin on the program connector is connected to each of the eight photodiodes in the read station of the device; there are also pins for each of the seven least significant bits of a byte in main storage, and two pins connected to reader circuits for checking odd or even parity.

You connect seven of the photodiode pins to any of the seven bit pins; any of the bit pins can be grounded so that the associated bit in main storage is always zero. You may also connect one of the two parity-check circuit pins to any one of the photodiode pins, or the parity-check pins can be so connected that no parity checking is done.

The reader section of the program connector also allows you to specify one *delete character* and one *stop character* for detection by the hardware when it is reading in character mode; however, in the binary mode, it cannot detect these characters. Therefore, if you need to delete characters from your input records in binary mode, or if your application requires additional delete characters, remember that you must specify these as software deletes in your program, using the SCAN keyword parameter of the DTFPT declarative macro (17.5.3.1).

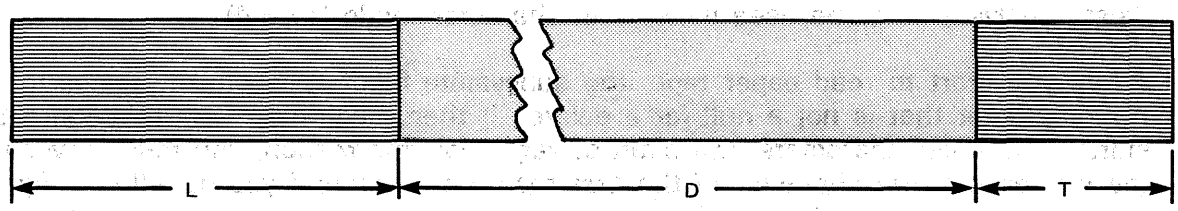
17.2.2. Paper Tape Leader

If the optional tape spooler is used with the 0920 paper tape subsystem, a tape leader at least three feet long must precede the first data character to be read on the tape. In the binary mode, the leader must consist only of *null characters;* in the character mode, you may use either null or delete characters. If the reader spooler is not used, the paper tape leader may be as short as two inches.

17.2.3. Paper Tape Trailer

When you are using the 0920 paper tape subsystem, you receive an indication of broken tape if there are data characters under the read photodiodes when the end-of-tape switch is activated. A false indication of broken tape, therefore, results when the end of an unbroken tape goes by while the last of the data characters are still being read. To prevent this false error indication from being given, you must follow the last data character on an input tape with a tape trailer at least 12 inches (120 characters) long; there must be nondata characters. In the binary mode, the trailer must consist only of null characters; in the character mode, you may use null or delete characters.

Figure 17—1 is a schematic diagram of a paper tape file with its leader and trailer.



LEGEND:

L Tape leader. Immediately precedes the first data character on paper tape and comprises only null characters in binary mode; may comprise null or delete characters in character (standard) mode. Must be at least three feet (360 characters) long if the optional tape spooler is used on 0920 paper tape subsystem; otherwise needs to be no more than two inches (20 characters) long.

T Tape trailer. Immediately follows the last data character on paper tape; must be at least 12 inches (120 characters) long to prevent false error indication of broken tape. Comprises only null characters in binary mode; may comprise null or delete characters in character mode.

D Data file. In character mode, the format of data records may be either fixed, unblocked or undefined (that is, records are of various lengths). The only record format used in binary is fixed, unblocked.

Figure 17—1. Tape Leader, Paper Tape Data File, and Tape Trailer

17.3. CHARACTER AND RECORD TYPES ON PAPER TAPE

Before looking into the way your data is organized on punched paper tape and appears to data management and to your program, consider the uses of several character types already mentioned, but not explainted so far: the null character, the delete character, and the end-of-record stop. These are part of a class of characters, selected by convention or arbitarily from among all the possible patterns that may be punched in one character's space on tape, different from the rest only in that they are not used to represent data. They are sometimes called *control characters*, but, on the other hand, they may represent the absence of data or serve as information separators or delimiters to format your data. They may serve merely to fill space on tape as a tape leader as trailer. One of them may be used to obliterate unwanted or erroneous data characters or other nondata characters. Characters from this class may indeed be used to control communications or devices (here, for example, the 0920 paper tape subsystem itself), and the term *control characters* is then appropriate.

An important realization is that each tape code you must assign to one of these characters reduces by one the subset of possibilities you may use to represent your data. When you are processing in character, or standard, mode, however, this limitation is not as serious as it may seem: data management's letter/figure shifting capability allows you to represent more than one set of charcters in the same set of hole patterns punched on tape.

17.3.1. Null, Delete, and Stop Characters

- Null Character

  The null character is represented on tape by the absence of any punches in the information levels (tracks); only the feed (or sprocket) hole is punched. To punch the tape code for the null character, you place the hexadecimal value 00 — a standard convention in both ASCII and EBCDIC processing — in your I/O area. You may use the null character (or the delete) in the *paper tape leader* and *trailer* when you are processing in character mode (MODE=STD), but you must use *only* the null for these purposes in binary mode (17.2.2, 17.2.3). Similarly, you must use only the null character to represent the optional *interrecord gap* in binary processing, although the delete character may be used instead, in character mode (17.3.4).

  When you start to read paper tape, the subsystem feeds tape until it comes to the first character that is not a null (or a delete); it then, beginning with this character, starts to transfer characters into main storage. For this reason, you must never use the null as the first character of the *first* record on a tape; if you do, all subsequent record lengths are off by one byte. In binary mode, once the transfer of data has begun, all subsequent null characters are read into main storage, as binary 0's (hexadecimal 00), and, whatever these represent in your input file, you must program to deal with them. In character mode, on the other hand (MODE=STD), nulls are never transferred into main storage.

- Delete Characters

You may represent a delete character on tape by any of the hole patterns available for you to punch, although the standard ASCII delete character is a punch in each of the seven data tracks (17.5.10). The practice of punching a hole in each track or data level to represent a delete character facilitates one of the main uses of the character: to cancel or obliterate unwanted data in a record. There are, however, two types of these: the "hardware" delete and the "software" delete.

In character mode only (MODE=STD), by wiring the program connector board, you can cause the hardware itself to recognize *one* of the incoming tape codes as a character to be deleted, which it does not transfer to main storage. It skips this character, without leaving a space in its place, and goes on to read in the next data character. This is called the "wired" or "hardware" delete; but such a character cannot be used in binary mode.

To prevent a character you want treated as a delete from being read into main storage in binary mode, you must so designate it in a scan table that you specify to data management via the SCAN keyword parameter of the DTFPT declarative macro that defines the file (17.5.3.1). You may specify more than one delete character this way — these are called "software" deletes, because the data management software takes care of them — in *both* binary and character modes.

A major use of the delete character, as suggested, is to obliterate unwanted characters from your paper tape file when these are identified in later processing, once the file is created. However (with one exception), you must take pains never to include the delete character in your output data when you are creating a paper tape file; it is punched into the tape by other means. Because of this use in cancelling out any tape code, the delete is often called a "rub-out" character.

The exception to the general rule of always excluding the delete character from appearance among the data characters in your output is its use as the record gap character when you are punching a tape (17.3.4).

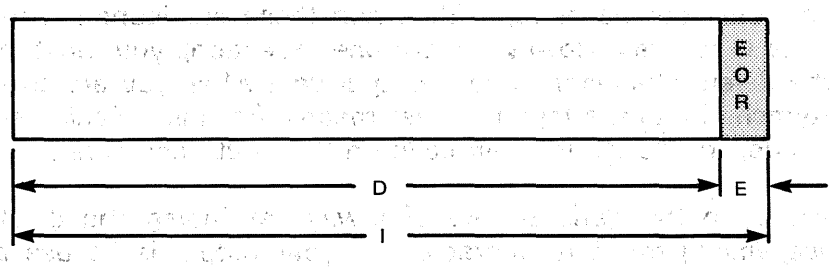
The other uses of the delete, to constitute the paper tape leader and trailer, or the interrecord gap (the record gap character) in standard mode, have already been referred to. In these uses, it is important to remember that the delete character must be the *hardware* delete, to be wired for input files via the program connector board, and not one of the software deletes designated in a SCAN table. You may place the hexadecimal code for the hardware delete in your I/O area as many times as you want it punched in the header or trailer, or you may form these file-delimiting character strings by splicing already punched strips of tape onto the ends of the file after it is punched without them.

- Stop Character

Undefined records, used only when you are processing in character mode (MODE=STD), require some means of delimitation, as they vary in length. For this, you use an end-of-record stop character at the low-order end of the record; it is placed there automatically by data management when you are punching an output file. It is read into your I/O area from an input file, and marks the farthest point into your I/O area you should attempt to reach in processing your data. You specify to data management what it is to use for this character on an output file by means of the EORCHAR keyword parameter of the DTFPT declarative macro defining the file (17.5.6); for an input file, you must specify the end-of-record stop to the 0920 paper tape subsystem itself by wiring the program connector board (17.2.1.2). When the subsystem encounters this character on reading an undefined record in character mode, it automatically stops tape motion. Remember that in binary mode the subsystem cannot be wired to recognize a stop character. The end-of-record stop character may not be used in OS/3 as the record gap character.

The use of an end-of-record stop character affects the length of your I/O buffers and work areas in character mode (MODE=STD); these effects are discussed in detail in 17.5.1.3, 17.5.1.4, and 17.5.1.6. Figure 17—2 shows the relationship of record length to the block size specification for an undefined record of the maximum size in a file. Note the position of the stop character in some of the subsequent figures, also.

UNDEFINED RECORD (USED IN CHARACTER (STANDARD) MODE ONLY)



LEGEND:

E End-of-record character, a "wired stop" character placed by data management as a delimiter at the end of each undefined (variable-length) record. Specified by the EORCHAR keyword of a DTFPT declarative macro defining an output paper tape file; set for input files with clip-on wires in the reader section of the program connector.

D Length of data record, which may not exceed 4094 bytes. For output files, you load this length into the register specified with the RECSIZE keyword of the DTFPT macro. For input files, data management loads the RECSIZE register with the length of each record it reads in.

I Assuming that D is the length of the longest data record in the paper tape file, then I as depicted here equals the BLKSIZE specification, which may not exceed 4095 bytes. The BLKSIZE specification is the size of the largest logical record to be processed and always includes 1 byte for the EORCHAR stop character that follows the data in an undefined record.

Figure 17—2. Undefined Paper Tape Record of Maximum Size for the File: Relationship of Record Length to BLKSIZE Specification

17.3.2. Letter and Figure Shift Characters

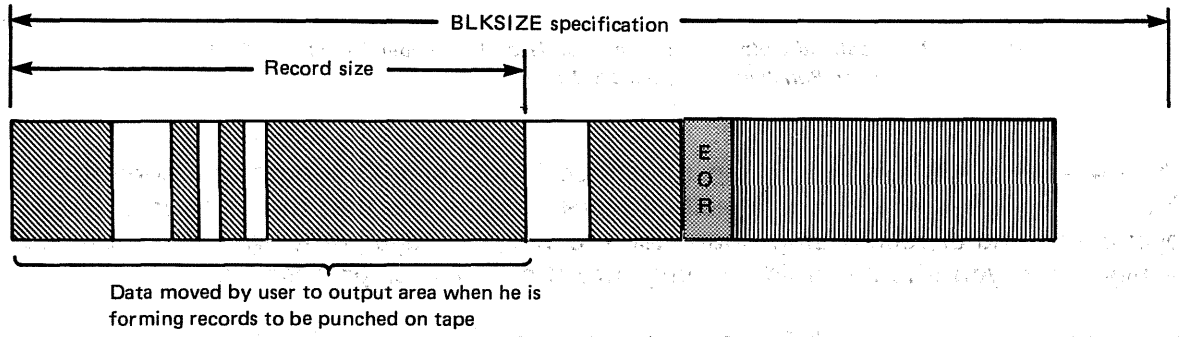
Only when you are processing in character mode (MODE=STD) may you use the letter/figure shifting capability that data management offers to permit you to represent more characters than you have unique hole patterns available in the number of data levels or tracks in your tape.

Setting aside a few of the hole patterns for the null character, one or more deletes, and (if you have undefined records in your file), the end-of-record stop, you can nearly double the number of characters that the remaining hole patterns represent if you select two more to be shift codes and then assign two meanings to each of the patterns that this leaves you: one meaning when it follows the one shift code on tape, one when it follows the other.

The conventions in OS/3 data management are that one of these shift codes is the *letter shift character,* all hole patterns following it on tape being translated as "letters", and that the other is the *figure shift character,* all tape codes that follow it being translated as "figures". Another convention is that, on opening an input file, data management expects that the first record of the file begins with a "figure" unless the first character of the record is the letter shift code. Data management inserts all shift codes automatically as it punches your output records into paper tape files, and it deletes them automatically, translating the intervening data as required, as it reads input records from tape.

To represent the letter shift and the figure shift codes, you may select any of the codes that you can punch into the tape at your disposal — except for the null, deletes, and end-of-record stop. Figure 17—3 shows the appearance of an undefined output record, comprising both "figures" and "letters", as it would appear in the I/O area after you had formed it there for punching by data management, and as the record would look after it was punched as the first record on tape. Notice that the letter shift code has been added automatically by data management as the first character of the punched record.

UNDEFINED OUTPUT RECORD, LESS THAN MAXIMUM SIZE FOR FILE, AS IT APPEARS IN I/O AREA



LEGEND:

"Letters": 8-bit configurations user has designated to be translated by the content of his LSCAN and TRANS tables

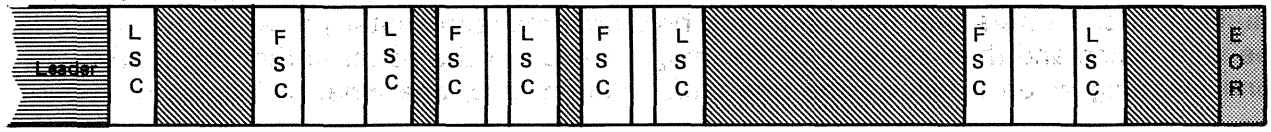
"Figures": 8-bit patterns user has designated to be translated by the content of his FSCAN and TRANS tables

E O R — End-of-record stop character, placed in I/O area by data management; specified by EORCHAR keyword

Residual data in I/O area, not processed

Figure 17—3. Undefined Output Record for Standard Mode Paper File in I/O Area and as Punched on Tape (Part 1 of 2)

AND AS IT APPEARS PUNCHED AS THE FIRST RECORD ON A PAPER TAPE FILE



LEGEND:

LSC — Letter shift code (LSC), punched by data management, the OS/3 convention is that the first character in the first record on tape is either a "figure" or the letter shift code.

Tape codes to be translated on input as representing "letters", because they follow LSC and precede FSC

Tape codes to be translated on input as representing "figures", because they follow FSC and precede LSC

FSC — Figure shift code (FSC), never the first character of the first record on tape

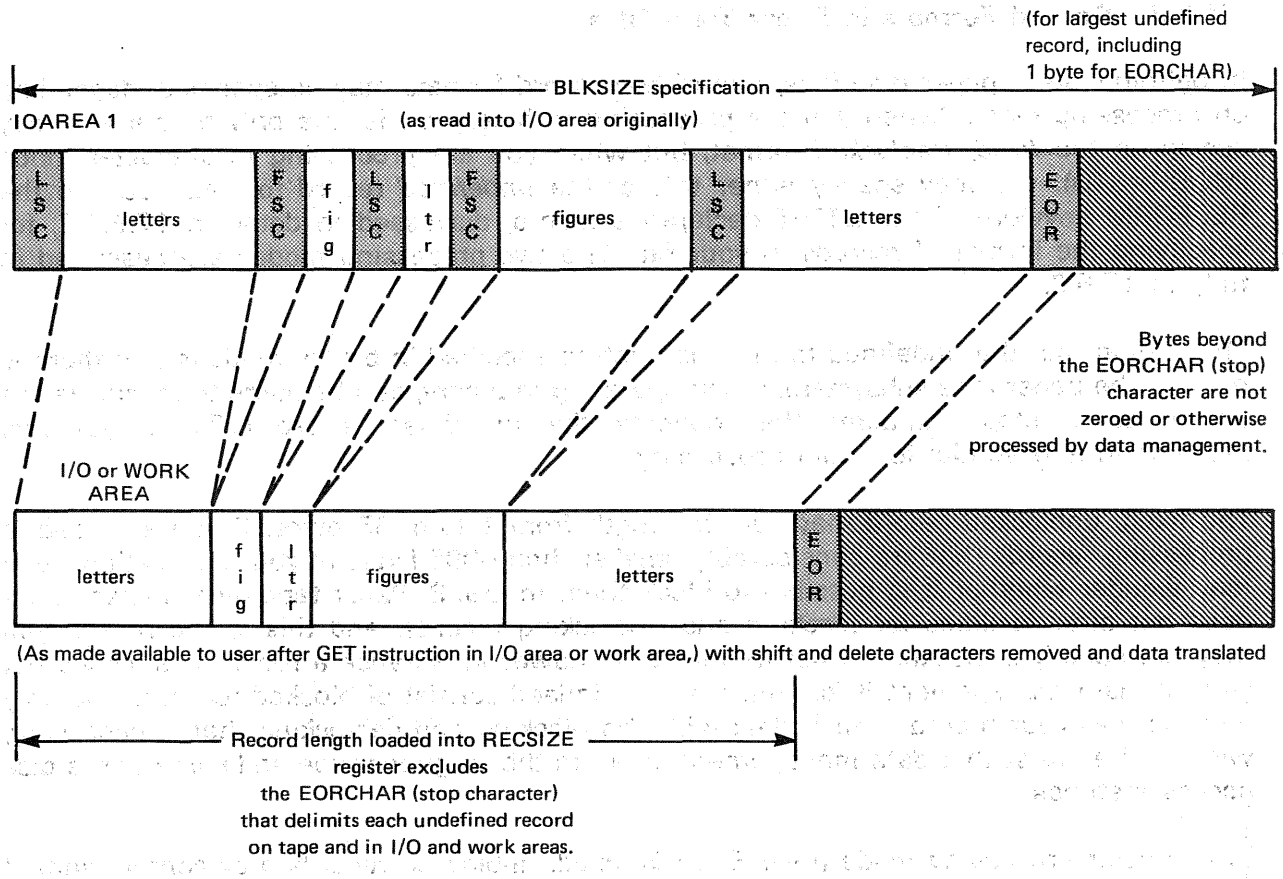
EOR — End-of-record delimiter, a character specified by the EORCHAR keyword of the DTFPT macro. On input, when this character is encountered, tape motion stops. The 0920 paper tape subsystem can be wired to recognize this character only in standard mode.

Paper tape leader

Figure 17—3. Undefined Output Record for Standard Mode Paper File in I/O Area and as Punched on Tape (Part 2 of 2)

Figure 17—4 shows the relations of an undefined input record's length and contents to the I/O and work areas (and to the specified block size); notice that the bytes in the data area beyond the end-of-record stop character are not zeroed or otherwise processed by data management; you should avoid running into them in your processing.

The means for designating the shift codes, for designating which characters are "figures" and which "letters", and for translating these are described in full elsewhere (17.5.3, 17.5.3.1, and 17.5.5, for example) under the DTFPT macro keyword parameters used for the purpose. See also Figure 17—10, in 17.5.1.5.



LEGEND:

EOR — EORCHAR (stop character)

LSC — Lettershift character

FSC — Figure shift character

Bytes beyond the EORCHAR (stop) character. These are not zeroed or otherwise processed by data management and should not be accessed by the user.

Figure 17—4. Relationships of Record Length, Work Area Length, and I/O Area Length to BLKSIZE Specification and Content of RECSIZE Register for an Undefined Record Input from Paper Tape with Shifted Codes

17.3.3. Record Formats in Paper Tape Files

In defining OS/3 paper tape files, one of two record formats may be specified, depending on processing mode. When you are processing in binary mode, the only format you may specify is the *fixed, unblocked* format, but when you are processing in character mode (MODE=STD), you may specify either this or the *undefined* record format. You use the RECFORM keyword of the DTFPT declarative macro, discussed in detail in 17.5.1.2, for specifying the format of records in your file. The two processing modes are described in fully in 17.5.2.

The reason that the undefined format may not be specified in binary mode is that there is no way the paper tape subsystem, when operating in binary, can be made to recognize the end-of-record stop character that delimits the undefined record (17.3.1), yet this recognition is essential for input processing.

The fixed, unblocked record may vary in length from 1 to 4095 bytes. If you are used to processing records that are considerably smaller than 4095 bytes in your applications, you may wonder whether it is possible to block them in OS/3. Paper tape data management does not offer facilities for blocking and unblocking records, and this is the reason you may not specify a blocked format in your DTF; however, because a record is simply what you tell data management it is, your file may indeed consist of blocked records. The only point is that your program must deal with the blocking and deblocking that is necessary, without the assist that data management offers in the magnetic tape and some of the disk access methods.

In character processing mode (MODE=STD), fixed, unblocked records may contain shifted characters and shift codes (they may not in binary mode), and when they do, moreover, you should note that they will seldom be all of one size when punched on paper tape. Figure 17—5, contained in a following paragraph (17.3.4), illustrates this point. Although fixed, unblocked records may not be shifted in binary mode, they may be translated.

The undefined record is simply one that may be of any length up through 4094 bytes; its maximum length is one byte less than the limit for the fixed-length record because of the need for the end-of-record stop character just described (17.3.1). The actual length of individual undefined records may vary from record to record; it is marked by the delimiting stop character and referred to in the general register designated for record size (17.5.1.6). It, too, may actually contain blocked records, but there is no way to specify this fact to data management, which has no facilities for blocking or deblocking. You must do this yourself.

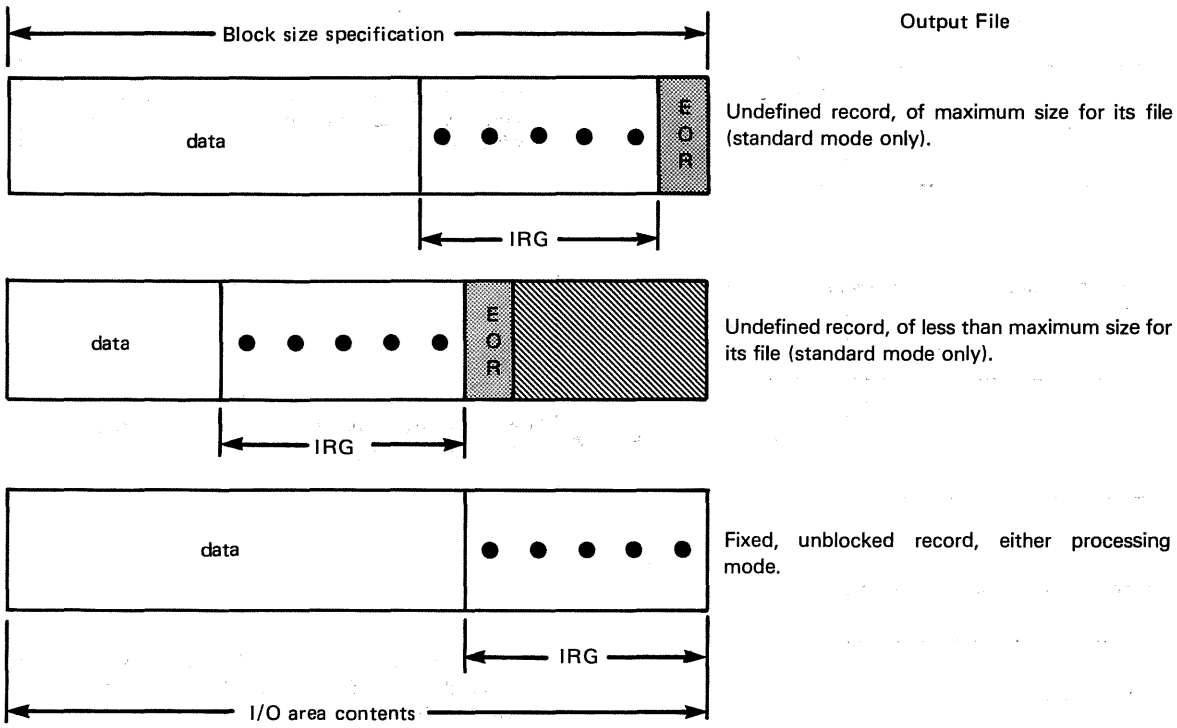
Figures 17—5, 17—6, and 17—7, presented in the following paragraph, show the appearance of records of each format on tape and in the data area. These records are followed by an interrecord gap.

17.3.4. Interrecord Gaps in paper Tape Files

Data management does not provide automatic facilities for punching a string of nondata characters (null or delete characters, for example) after each output record to serve as an interrecord gap. Yet a gap so constituted, especially if it contained a fixed number of these characters, might facilitate error recovery and would serve to distinguish records on tape.

If you want to create an interrecord gap for these reasons, you need only to place the number of null or delete characters that you have decided to use at the end of each record in the output of work area. There are a few considerations to keep in mind, however; these affect both your block size specification and (for undefined records only) the value placed in the general register used to refer to logical record length.

With output files in both binary and standard modes, a byte for every character to be punched at the end of a record as part of an interrecord gap must be included in your calculation of block size, which you specify with the BLKSIZE keyword parameter of the DTFPT declarative macro (17.5.1.3). Furthermore, if your records are undefined, your block size must still allow one byte more for the end-of-record stop character, which data management inserts in the I/O buffer as a delimiter *after* the last record gap character you place there. Therefore, the overall length of data-record-plus record-gap that you form must be at least one byte shy of your BLKSIZE specification. You also count the bytes for the record gap characters at the end of undefined records in the record length that you load into the general register that you must use to refer to record size (the RECSIZE register, 17.5.1.6). Refer to Figure 17—5.



LEGEND:

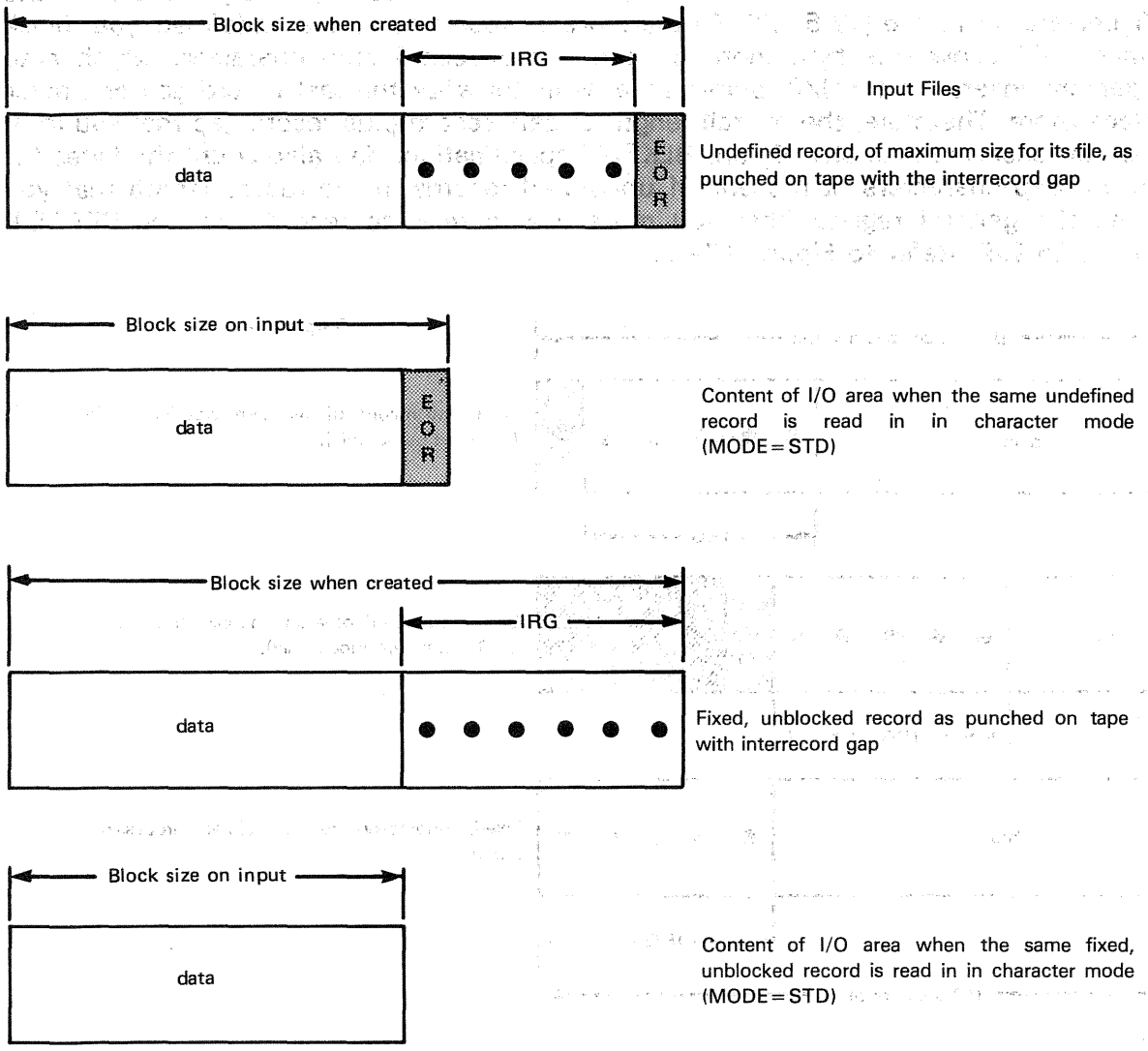
E O R — End-of-record stop character, punched by data management

Bytes in I/O area beyond end-of-record stop, not processed by data management nor to be accessed by user

IRG Length of interrecord gap, characters supplied by user

Figure 17—5. Undefined and Fixed, Unblocked Records Followed by Interrecord Gaps in Output Paper Tape File, Either Processing Mode

With input files processed in character mode (MODE=STD), nulls or deletes at the end of each record are not included in your BLKSIZE specification, because they are not transferred into main storage. For the same reason, data management does not include the bytes for the record gap characters in the record length it loads for undefined records into the RECSIZE register. A standard mode output file containing interrecord gaps would therefore require a smaller BLKSIZE specification when it is read in than it needed when punched. Refer to Figure 17—6.



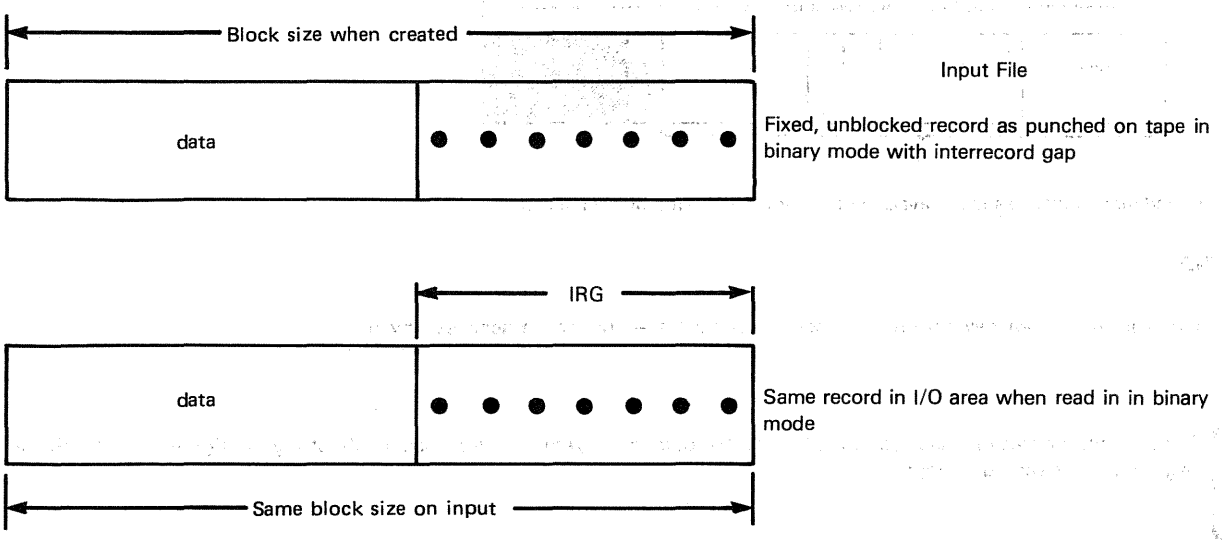
LEGEND:

EOR End-of-record stop character, punched by data management

IRG Length of interrecord gap, characters supplied by user when files were created. Punched on tape, but not transferred to main storage on input in character mode (MOI E=STD)

Figure 17—6. Undefined and Fixed, Unblocked Records Followed by Interrecord Gaps in Input Paper Tape Files, Standard Processing Mode

For input files processed in binary mode, on the other hand, the situation is different. Because the record gap characters *are* transferred into main storage (as nulls, hexadecimal 00) at the end of each record, you must include the fixed number of bytes for calculating block size, as well as programming appropriately to account for their presence. A binary mode file containing interrecord gaps would have the same BLKSIZE specification for input processing as it had when punched. Refer to Figure 17—7.



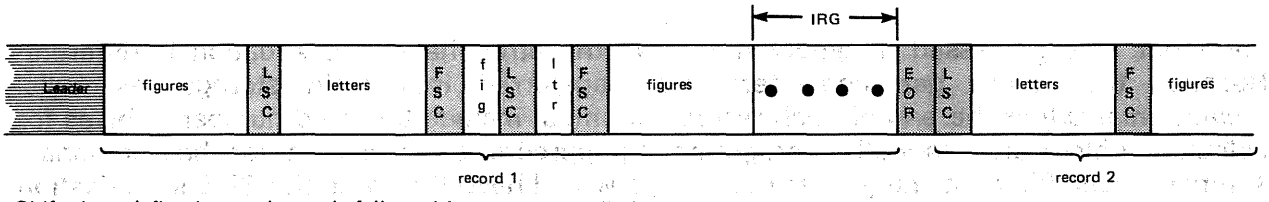
LEGEND:

IRG Length of interrecord gap. Characters supplied by user when file was created. In binary mode, these characters are read into main storage as input and must be included in BLKSIZE specification and in reserving main storage for I/O area.

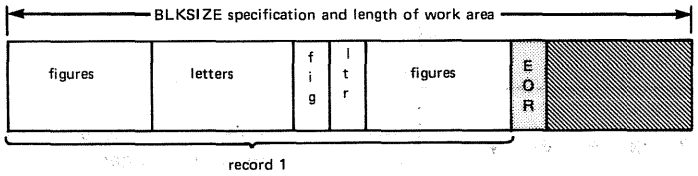
Figure 17—7. Fixed, Unblocked Record Followed by Interrecord Gap in Input Paper Tape File, Binary Processing Mode

It might occur to you that, when you are processing output files in character mode (MODE=STD), you could punch a string of end-of-record stop characters, instead of nulls or deletes, as an interrecord gap between undefined records. Although you *could* punch a series of these characters, when your file is read in, the device would stop tape motion at each of the end-of-record characters, and could not be made to skip over them. Other paper tape systems you are familiar with may allow consecutive end-of-record characters, without intervening data, to be punched in output tapes and skipped over on input — OS/3 does not provide the facility to skip these on input.

Figures 17—8 and 17—9 depict shifted undefined records and shifted fixed, unblocked records as they exist on tape and appear in your work area when processed in character mode (MODE=STD).



Shifted, undefined records, each followed by a user-supplied interrecord gap as they appear on tape



The first undefined record as made available to user in work area by GET macro.

LEGEND:

Bytes in work area beyond end-of-record stop character, not to be accessed by user

FSC — Figure shift character inserted and deleted by data management. Precedes each string of figures except those beginning first record or tape

LSC — Letter shift character inserted and deleted by data management. Precedes each string of letters in every record on tape

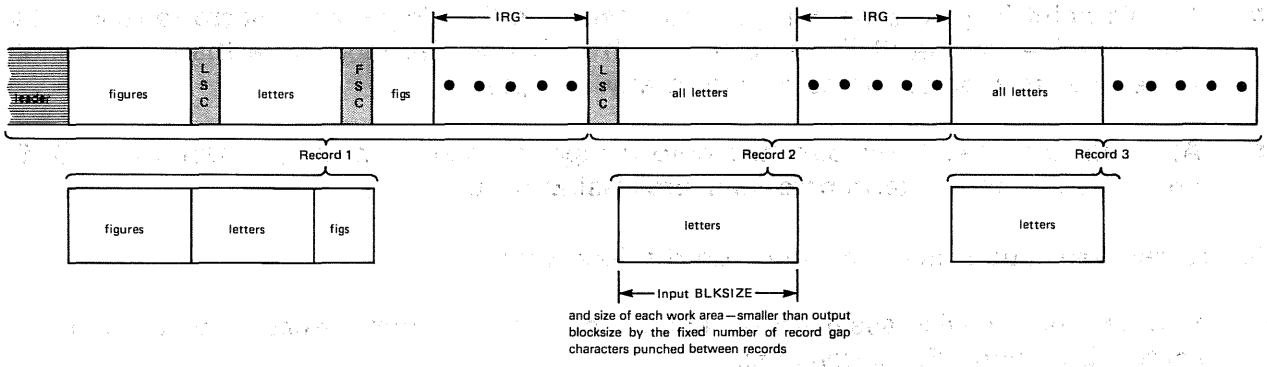
EOR — End-of-record stop character, delimiter for each undefined record. Specified for input file by wiring program connectors board

IRG Interrecord gap, characters supplied by user. Not transferred to main storage in character mode (MODE=STD).

NOTE:

The record length placed by data management in the RECSIZE register does not include the end-of-record stop character — even though this character is read into I/O area and is moved with the record into the work area.

Figure 17—8. Shifted, Undefined Records as They Appear on Paper Tape and in User Work Area: Input File, Character Mode (MODE=STD)



LEGEND:

LSC Letter shift code, inserted and deleted by data management

FSC Figure shift code, inserted and deleted by data management

IRG Interrecord gap, a fixed number of characters supplied by user on output but not transferred on input in character mode

NOTES:

1. Record 1, including its interrecord gap, is two bytes larger than specified by the BLKSIZE keyword at output because of the shift codes required after the first and second fields and inserted there by data management.

2. Record 2, likewise, is one byte larger than the output BLKSIZE specification (which must include the fixed number of record gap characters you supply for each record), because of the letter shift code required. This record begins with a letter, and the preceding record ended in a figure.

3. Record 3 is exactly the length specified by the BLKSIZE keyword at output. No shift code is required: the record contains only letters, and the preceding record ended with a letter.

Figure 17—9. Shifted, Fixed, Unblocked Records on Paper Tape and in Work Areas: Input File, Character Mode (MODE=STD)

17.4. PROCESSING PAPER TAPE FILES

The procedures in processing paper tape files are simply summarized. Before creating your program, you obviously will know what form of tape you are going to read or punch, whether it is to be in binary or character mode, whether it is to be translated into or from the standard EBCDIC, and whether it is to contain shifted characters or rely on only one, unique use of the codes available to you. Once these points have been determined by the nature of your application, you will take the following steps to use the paper tape data management system:

- You define each of your paper tape files, using the keyword parameters of the DTFPT declarative macro to specify your requirements.

- Before reading or punching any data from or to any paper tape file, you issue an OPEN imperative macro to initialize the file.

- At this point in your program, you may issue the GET imperative macro to read data from the file, or the PUT macro to punch data into the tape. Note that you cannot issue both macros to the same file in any one pass.

- After all data has been punched onto a tape, or read from it, you issue a CLOSE imperative macro to terminate your processing of it.

At program execution time, you must do the following:

- Provide proper device assignement and logical file definition for each file, through job control DVC and LFD statements.

- Ensure that the program connector in the 0920 paper tape subsystem is properly set up.

- Ensure that the proper paper tape is mounted and that the device is online.

The following paragraphs describe the four imperative macros you may issue to a paper tape file: OPEN, CLOSE, GET, and PUT.

OPEN

17.4.1. Initializing a Paper Tape File (OPEN)

Before issuing any of the other imperative macros to it, you must issue an OPEN imperative macro to the paper tape file to be processed. The transients and overlays called as a result link your file to the device you have assigned through job control and perform other file initialization procedures.

For an input file, for example, that contains letter/figure shift codes, the OPEN processing initializes the file so that the first record on tape is read in the figures mode. If the first record on a paper tape actually begins with letters, therefore, a letter shift code must be the first character punched on tape, or the record will be misread. This code is inserted automatically for tapes punched by OS/3.

Format:

LABEL	ΔOPERATIONΔ	OPERAND
[symbol]	OPEN	{ filename-1 [,...,filename-n] / (1) / 1 }

Positional Parameter 1:

filename
Is the label in your program of the DTFPT declarative macro defining the paper tape file to be initialized. You may initialize as many as 16 data management files with one OPEN macro call; they need not all be paper tape files.

(1) or **1**
Indicates that you have preloaded general register 1 with the address of the DTFPT declarative macro. You use this form only when you have a single file to initialize.

Examples:

	LABEL (1)	ΔOPERATIONΔ (10)	OPERAND (16) Δ
1.	P001	OPEN	DMPT1,DMPT2
2.		LA	1,DMPT7
		OPEN	(1)

1. Opens paper tape files DMPT1 and DMPT2

2. Opens the paper tape file DMPT7

CLOSE

17.4.2. Terminating Paper Tape File Processing (CLOSE)

When you have completed processing your paper tape file, you must issue the CLOSE imperative macro before taking any action to terminate the job (such as issuing supervisor macros EOJ, CANCEL, DUMP, etc). Transients called by the CLOSE macro ascertain whether all I/O operations have been completed, process errors on the final I/O operations, and so forth. If you require further processing on the paper tape file, you must reopen it with the OPEN macro.

Format:

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

Positional Parameter 1:

filename
Is the label in your program of the DTFPT declarative macro defining the paper tape file whose processing is to be terminated. You may close as many as 16 data management files with one CLOSE macro call; they need not all be paper tape files.

(1) or **1**
Indicates that you have preloaded general register 1 with the address of the DTFPT declarative macro. You use this form only when you have a single file to terminate.

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

Examples:

	LABEL (1)	OPERATION (10)	OPERAND (16)
1.	P030	CLOSE	DMPT1,DMPT2
2.		LA	1,EX1
		CLOSE	1

1. Closes the files labeled DMPT1 and DMPT2

2. Closes the file labeled EX1

GET

17.4.3. Reading a Logical Record from Paper Tape (GET)

You issue the GET imperative macro to an input file to make a logical record available to you either in a work area or in an I/O area. In executing a GET macro, before data management makes an input record available to you in an I/O area or moves it from there to your work area, it has removed all shift or delete characters from the record and has translated all data that requires translation.

If you want to use a work area for processing input records, you must specify the WORKA keyword parameter in the DTFPT declarative macro defining the file and must also specify the address of the work area with each issue of the GET macro. If you do not specify the WORKA keyword, you must access each record in an I/O area. And, if you have specified two I/O areas but no work area, you must use an I/O register to access the record (17.5.1.4).

When your record is an undefined record (RECFORM=UNDEF), the end-of-record character appears in the data area at the end of the record; furthermore, any unused bytes in the area beyond this stop character are not zeroed or otherwise processed by data management (refer back to Figures 17—3 and 17—4, for example). If you wish, you may specify a record size register when you are reading undefined records. Data management then loads this register with the length of the record, as it appears in the data area minus shift and delete characters; this length does not include the end-of-record stop character.

Format:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	GET	{filename / (1) / 1} [, {workarea / (0) / 0}]

Positional Parameter 1:

filename
Is the label in your program of the DTFPT declarative macro defining the input paper tape file from which you are reading a record.

(1) or **1**
Indicates that you have preloaded register 1 with the address of the DTFPT file table.

Positional Parameter 2:

workarea
Is the symbolic address (label) of your work area. You may change this label from one GET macro to another.

(0) or **0**

Indicates that you have preloaded register 0 with the address of a work area.

Examples:

	LABEL (1)	OPERATION (10)	OPERAND (16)
1.		GET	DMPT1
2.		GET	DMPT2,WK1
3.	Q005	LA	0,WK5
		GET	DMPT3,(0)

1. Read a record from paper tape file labeled DMPT1 and leave the record in an I/O area.

2. Read a record from paper tape file labeled DMPT2 and move the record to the work area labeled WK1.

3. Read a record from paper tape file labeled DMPT3 and move the record to the work area labeled WK5.

PUT

17.4.4. Punching a Logical Record into Paper Tape (PUT)

Having provided the data for an output record in an I/O area or work area, you issue the PUT imperative macro to punch it as a logical record into paper tape. If you are using a work area, you must specify the WORKA keyword in the DTFPT declarative macro that defines your output file, and you must specify the address of a work area with each PUT macro you issue. The work area may differ from macro to macro.

Data management moves the output record from the work area to the output area and, before punching the record on tape, inserts letter and figure shift characters and translates data as necessary. If this is the first record on the tape, and if it begins with a letter, data management automatically punches the letter shift character as the first character of the record. If the record format is undefined, data management punches the end-of-record stop character at the end of each record. You will have specified the shift characters with the LSCAN and FSCAN keywords, and the end-of-record character with the EORCHAR keyword, when you defined the file with the DTFPF declarative macro.

To punch out undefined records, you must also have specified a record size register, using the RECSIZE keyword in the DTF, and, determining the length of each record, you must load this length into the register before you issue each PUT macro. The number you load into the RECSIZE register is the number of bytes of data in the record; it does not include bytes for the end-of-record stop character nor for the shift characters, which data management inserts.

Format:

LABEL	Δ OPERATION Δ	OPERAND
[symbol]	PUT	{filename / (1) / 1} [, {workarea / (0) / 0}]

Positional Parameter 1:

filename
Is the label in your program of the DTFPT declarative macro that defines the output paper tape file.

(1) or **1**
Indicates that you have preloaded register 1 with the address of the DTFPT file table.

Positional Parameter 2:

workarea
Is the label in your program of the work area from which you want the output record moved.

(0) or **0**

Indicates that you have preloaded register 0 with the address of the work area from which you want the record moved.

Examples:

	LABEL (1)	OPERATION (10)	OPERAND (16)
1.	EX500	PUT	EX7
2.		PUT	DMPT8,WKA
3.		LA	0,WKA
		PUT	DMPT8,(0)

1. Punch a record which the user has placed in an I/O area, into the paper tape file whose label is EX7.

2. Move the record that the user has placed in the work area labeled WKA to an I/O area and punch it into the paper tape file whose label is DMPT8.

3. Same as 2.

DTFPT

17.5. DEFINING PAPER TAPE FILES (DTFPT)

You define your paper tape files to data management by issuing a DTFPT declarative macro for each file you will process in your BAL program, using its keyword parameters to describe the file. The DTFPT macro does not generate executable code (and therefore should not be issued in the midst of executable instructions or imperative macros); it does generate a file table to contain data about the file and the results of processing.

When your program is assembled, the assembler expands your DTFPT declarative macro into a 215-byte file table, which it uses in a number of ways to control file processing and to record certain results.

As you execute each imperative macro to process your file, for example, data management places an informative reply, indicating normal completion of I/O or exceptional conditions (including unrecoverable error) in a program-addressable field of the DTF file table called *filenameC*. Your use of this field is explained under the ERROR keyword parameter (17.5.9), as is the use of another field, *filenameD*, to access a record containing a parity error.

Following is a format delination of the DTFPT declarative macro, showing the required and optional keyword parameters you will use to define your file and to indicate to data management some of your file processing requirements. Notice that the keywords are listed here in alphabetic order; however, you may issue them in any convenient order, separating them with commas.

Format:

LABEL	Δ OPERATION Δ	OPERAND
filename	DTFPT	BLKSIZE=n [,EOFADDR=symbol] [,EORCHAR=expression] [,ERROR=symbol] [,FSCAN=symbol] [,FTRANS=symbol] ,IOAREA1=symbol [,IOAREA2=symbol] [,IOREG=(r)] [,LSCAN=symbol] [,LTRANS=symbol] [,MODE={BINARY, STD}] [,OVBLKSZ=n] [,OPTION=YES] [,RECFORM={FIXUNB, UNDEF}] [,RECSIZE=(r)] [,SAVAREA=symbol] [,SCAN=symbol] [,TRANS=symbol] [,TYPEFLE={INPUT, OUTPUT}] [,WORKA=YES]

A comma is shown preceding every keyword but the first, to remind you that all keywords coded in a string must be separated by commas. However, a comma must not be coded in column 16 of a continuation line, nor follow the last of the string. Refer to the preface of this manual for OS/3 format statement conventions and to 1.6.3 for rules on continuation.

The symbolic label of the DTFPT declarative macro (*filename*), required for all DTFPT files, is the logical name by which you address the file in your BAL program, it may contain no more than seven alphanumeric characters, the first of which must be alphabetic. Restricting *filename* to seven characters allows data management to generate symbols, such as *filenameC*, which you may reference in each DTF file table by concatenating a letter to the file name. You specify this file name to the imperative macroinstructions you issue to process your file, and it is this name also that you use in the job control *logical file definition* (LFD) statement with which you allocate the file.

Notice that there are two keyword parameters (BLKSIZE and IOAREA1) that you must always specify in every DTFPT macro. Others are required under certain circumstances, and some are entirely up to you.

Notice also that the completion of many of the keyword parameters is *symbol*, representing the fact that with *symbol* you specify the symbolic address of the subroutine, translation table, scan table, or I/O buffer the keyword stands for. In expanding your DTFPT declarative macro, the assembler generates an EXTRN pseudo-op for each symbolic label the macro contains; this means that the corresponding subroutine or table need not be assembled with the DTFPT macro, but, when it is more convenient or advantageous for you to do so, may be assembled separately.

These are the keywords in point:

Keyword	Subsection
EOFADDR	17.5.4
ERROR	17.5.9
FSCAN	17.5.5
FTRANS	17.5.3
IOAREA1	17.5.1.4
IOAREA2	17.5.1.4
LSCAN	17.5.3
LTRANS	17.5.3
SAVAREA	17.5.8
SCAN	17.5.3
TRANS	17.5.3.1

Table 17—1, following the DTFPT format delineation, summarizes the rules for specifying the keywords. The keywords are discussed, in full detail, in the subsequent paragraphs. Information on acceptable variations of some of the keywords is given in 17.6, which discusses the compatibility of OS/3 with certain other data management systems for paper tape.

Table 17—1. Summary of DTFPT Keyword Parameters (Part 1 of 2)

Keyword	Completion	Remarks	Use With File Type		EXTRN	Use With Processing Mode		Use With Shifted Codes	Use Without Shifted Codes	Do not Specify With	Paragraph for Details
			INPUT	OUTPUT		BINARY	STD				
BLKSIZE*	n	Required for all files. Specifies maximum length, in bytes, of largest logical record; *n* is a decimal number in the range 1 through 4095	R	R	–	R	R	–	–	–	13.5.1.3
EOFADDR	symbol	Specifies label of user's end-of-tape processing routine; required for all input files.	R	No	Yes	–	–	–	–	–	13.5.4
EORCHAR	expression	Required for output files with undefined records; specifies end-of-record stop character (delimiter)	No	R	No	No	R	–	–	RECFORM=FIXUNB, MODE=BINARY	13.5.6
ERROR	symbol	Specifies label of user's error routine. If not specified, errors return inline	O	O	Yes	O	O	–	–	–	13.5.9
FSCAN	symbol	Required, with LSCAN keyword, to specify label of user's figure scan table for punching files with shifted codes	No	O	Yes	No	O	R	No	MODE=BINARY	13.5.5
FTRANS	symbol	Required, with LTRANS and SCAN, to specify label of user's figure translation table for processing a shifted input file	O	No	Yes	No	O	R	No	TRANS, MODE=BINARY	13.5.3
IOAREA1	symbol	Required for all files; specifies label of primary I/O buffer	R	R	Yes	R	R	–	–	–	13.5.1.4
IOAREA2	symbol	Specifies label of optional secondary I/O buffer. Requires specification of IOREG if work area processing is not specified (WORKA keyword)	O	O	Yes	O	O	–	–	–	13.5.1.4
IOREG	(r)	Specifies, in mandatory parentheses, the number of the general register to be used as I/O index register. Required when IOAREA2 keyword is specified, but work area processing is not (WORKA keyword)	O	O	–	O	O	–	–	WORKA	13.5.1.4
LSCAN	symbol	Required, with FSCAN keyword, to specify label of user's letter scan table to punch files with shifted codes	No	O	Yes	No	O	R	No	MODE=BINARY	13.5.3
LTRANS	symbol	Required, with FTRANS and SCAN keywords, to specify the label of the user's letter translation table for an input file with shifted codes	O	No	Yes	No	O	R	No	TRANS, MODE=BINARY	13.5.3
MODE	BINARY	Required to specify binary processing mode	O	O	–	R	No	No	–	RECFORM=UNDEF	13.5.2, 13.5.2.1
	STD	Specifies character (nonbinary) processing mode	O	O	–	No	O	Yes	Yes	–	13.5.2, 13.5.2.2
OPTION	YES	Specifies optional file processing	O	O	–	O	O	–	–	–	13.5.7
OVBLKSZ	n	Specifies use and length of oversized I/O buffers for processing fixed, unblocked records containing shifted codes; *n* is a decimal number ranging from 2 through 4095 and must be at least one byte larger than the BLKSIZE specification.	O	O	–	No	O	O	No	RECFORM=UNDEF, MODE=BINARY	13.5.1.5
RECFORM*	FIXUNB	Specifies fixed, unblocked record format	O	O	–	R	O	Yes	Yes	–	13.5.1.2
	UNDEF	Specifies undefined record format (varying length). Records require delimiter (end-of-record stop). Output files require RECSIZE register.	O	O	–	No	O	Yes	Yes	MODE=BINARY	13.5.1.2
RECSIZE	(r)	Specifies, in mandatory parentheses, the number of the general register to be used to refer to length of undefined records. Required for output files; optional for input.	O	R	–	No	R	Yes	Yes	RECFORM=FIXUNB, MODE=BINARY	13.5.1.6
SAVAREA	symbol	Specifies label of 72-byte storage area, fullword-aligned, in which data management saves contents of user's general registers during execution of imperative macros. If not specified, data management expects save area address in register 13.	O	O	Yes	O	O	–	–	–	13.5.8
SCAN	symbol	Specifies label of user's input file shift code scan table. Required for input files with shifted codes (MODE=STD); FTRANS and LTRANS keywords must also be specified. Optional for software character deletion in binary mode. Optional, with TRANS keyword, for character deletion in either mode	O	No	Yes	O	O	R	Yes	–	13.5.3, 13.5.3.1
TRANS	symbol	Specifies label of user's translation table, for any file but a shifted input file	O	O	Yes	O	O	Yes	Yes	FTRANS, LTRANS	13.5.3.1, 13.5.3.2, 13.5.5 13.5.10
TYPEFLE	INPUT	Specifies an input file—read only	R	No	–	O	O	Yes	Yes	–	13.5.1.1
	OUTPUT	Required to specify an output file—punch only	No	R	–	O	O	Yes	Yes	–	13.5.1.1
WORKA	YES	Specifies double buffering via work areas, which user must specify with each GET or PUT macro. Ignored if IOREG keyword is specified	O	O	–	O	O	Yes	Yes	IOREG	13.5.1.4

*Parameter may be changed on DD job control statement.

Table 17—1. Summary of DTFPT Keyword Parameters (Part 2 of 2)

LEGEND:

R Required to be specified, explicitly or by default

O Specification is optional.

Default value assumed by data management if keyword is not specified

— Not pertinent

NOTE:

A "yes" entry in the column headed "EXTRN" indicates that the assembler generates an EXTRN pseudo-op for the symbolic label specified. This means that your coding that defines the subroutine, table, or main storage area in point may be assembled separately from your DTFPT macro. You will need an ENTRY for each EXTRN.

17.5.1. Basic DTFPT Keyword Parameters

The following subparagraphs explain several basic keyword parameters that you use to indicate to data management how to set up the file table according to certain of the fundamentals of your processing. With these keywords, you establish whether your file is an input or an output file, what the record format is, what the record and block sizes are, whether you are using oversized buffers, and which, if either, of the double-buffering options you are using.

17.5.1.1. Specifying File Type (TYPEFLE)

Data management provides two types of paper tape file: input and output; the combined file is not supported. The input file allows the use of the GET imperative macro to read data from a paper tape (17.4.3) and requires you to code a routine for end-of-tape processing (17.5.4). An output file allows the use of the PUT macro to punch data on a paper tape (17.4.4). If your 0920 paper tape subsystem is configured with both a punch and a reader, simultaneous reading and punching are possible, but on different pieces of tape; each of these requires separate definition with a DTFPT declarative macro and allocation with its own DVC — LFD job control device assignment set. You should use a different file name for each DTF.

Keyword Parameter TYPEFLE:

TYPEFLE=INPUT
Indicates that this DTFPT macro defines an input paper tape file, one that you want to read. You must specify the label of an end-of-tape processing routine for every input file, EOFADDR keyword parameter (17.5.4).

TYPEFLE=OUTPUT
Must be specified to define an output paper tape file, one that you want punched.

If you omit the TYPEFLE keyword, data management generates an input file table by default; EOFADDR keyword must still be specified.

17.5.1.2. Specifying Record Format (RECFORM)

In the OS/3 paper tape data management system, you have but two record formats: undefined (that is, records of various lengths) and fixed-length, unblocked. In the standard (character) processing mode, you may use either format, but only fixed, unblocked records may be processed in binary.

The reason for this is that, in the binary processing mode, the 0920 paper tape subsystem cannot be made to recognize the end-of-record stop character used as a delimiter to mark the ends of undefined records, which vary in length from record to record. Input paper tape processing of records with varying lengths depends on the recognition of the delimiter to stop tape motion automatically when a record has been read.

For processing output files with undefined records, you must specify both an end-of-record stop character with the EORCHAR keyword (17.5.6), and a general register to be used for referring to record size (the RECSIZE register, 17.5.1.6). Before you issue the PUT imperative macro to punch an undefined record, you determine its length and place this number in the two least significant bytes of the RECSIZE register. The length is measured in bytes and does not include the byte for the EORCHAR stop character. The number must be positive and in the range 1 through 4094. When you issue the PUT macro, data management places the stop character at the end of each undefined record in the I/O area before punching the whole contents into the tape.

When you are processing an input file containing undefined records, your use of the RECSIZE register is optional; if you specify it, data management loads it with the length of each record read in; again, this length does not include the EORCHAR stop character. You must wire the program connector of the 0920 paper tape subsystem to recognize the end-of-record stop character that is punched in the tape (17.2.1.2).

Keyword Parameter RECFORM:

RECFORM=FIXUNB
Specifies that the record format is fixed and unblocked. Only this format may be used in binary mode. You specify record length with the BLKSIZE keyword.

RECFORM=UNDEF
Specifies that record length varies from record to record and that records are delimited by a wired stop character (EORCHAR keyword). Length of each record is defined via the RECSIZE register. This record format is not valid for binary mode; if so specified, a diagnostic appears in the DTFPT macro expansion in your assembly listing, and RECFORM=FIXUNB is assumed.

If the RECFORM keyword is omitted, data management assumes that the record format is fixed, unblocked.

17.5.1.3. Specifying Block Size (BLKSIZE)

The maximum number of bytes that data management can transfer between paper tape and main storage in one access is 4095 bytes. (If you are familiar with OS/4 data management, you will recognize this figure as the maximum block size in that system as well.)

The maximum block size limits the length of your logical records. For the undefined record (RECFORM=UNDEF, 17.5.1.2), because the EORCHAR stop character must be included in the specified block size, the longest logical record is 4094 bytes. When you are processing undefined records, your I/O buffers and work areas must be at least as long as the number of bytes specified as the block size.

When you are processing fixed, unblocked records, which do not end in a stop character, your block size specification is the maximum length of a logical record and, because it is the number of bytes moved to or from a work area, each work area must be at least as long as the specified block size.

For neither record format does the specified block size need to account for the presence of shift characters, which data management inserts and deletes. When you are processing fixed, unblocked records, however, with shifted codes, you may specify that your I/O areas are larger than specified block size, using the OVBLKSZ keyword (17.5.1.5). When you do this, you must also reserve at least as many bytes of storage for each I/O area as you specified with the OVBLKSZ keyword, but you need not reserve more for each work area than you specified with the BLKSIZE keyword. Do not specify the OVBLKSZ keyword unless your fixed, unblocked records contain shifted characters.

With output tapes, additional bytes for any characters to be punched at the end of a record to serve as an interrecord gap (17.3.4) must be included in your BLKSIZE specification. With character mode input files (MODE=STD), however, nulls or delete characters at the end of each record are not transferred into main storage and must not be included in the BLKSIZE specification. The situation with binary mode input files is the same as for output files: because record gap characters are transferred into main storage (nulls as hexadecimal 00), you must include them in calculating block size, as well as programming as necessary to deal with them.

When you issue the OPEN imperative macro for a paper tape file, the OPEN transients check your BLKSIZE specification for validity. If there is no specification, or if the number of bytes specified is other than 1 to 4095 bytes, the file is not marked open, and you may not process it. Refer to 17.5.9 for the resulting data management error processing.

Keyword Parameter BLKSIZE:

BLKSIZE=n
Required for all input and output paper tape files. Specifies the length of the largest logical record to be processed, where *n*, a decimal number ranging from 1 through 4095, is the measure of this length in bytes.

If omitted, an error message appears in the DTFPT expansion in your assembly listing; the file cannot be opened for processing.

17.5.1.4. Specifying Buffers, Work Areas, and Double-Buffering (IOAREA1, IOAREA2, IOREG, WORKA)

You must always specify at least one I/O buffer for each paper tape file, using the IOAREA1 keyword parameter of the DTFPT declarative macro. When you specify only one, however, and do not use a work area, you have no means of overlapping I/O operations with your processing: each I/O operation must be completed before data management can return control to your program. (The reason for this is to prevent your inadvertently changing the data in your buffer before the I/O operation is completed.)

To increase your throughput over what this situation affords, data management offers two methods of double-buffering: specifying a secondary I/O buffer and an index register to point to the current one (with the IOAREA2 and IOREG keywords), or using one or more work areas for processing while your I/O takes place from or to the IOAREA1 buffer. With either of these mutually exclusive alternatives, you benefit. Double-buffering allows data management to initiate an I/O operation and to return to you *before* it is completed. While the I/O proceeds in one area (don't change the data in this one!), you may process in another. This overlapping of I/O and processing obviously speeds up the execution of your program.

No additional throughput is gained, however, if you specify a secondary I/O buffer *and* use work areas at the same time. Although OS/3 allows this combination of areas, it does not allow both double-buffering methods at once.

If you are going to use work area double-buffering, you must inform data management of this by specifying the WORKA keyword in the DTFPT macro (the specification is WORKA=YES), and you must then specify the address of a work area in the second operand of each PUT or GET macro you issue (17.4.3, 17.4.4). You may use a different work area each time. For an output file, data management moves your data from the work area to the I/O area before initiating an I/O operation to the punch. For input files, data management moves the data from the I/O area to the work area before making it available to you. Each work area you use (and there is no limit set by data management on their number) must be of a length of least equal to the number of bytes you specified with the BLKSIZE keyword (17.5.1.3). The reason for this is that data management always moves to or from a work area exactly this number of bytes.

If, on the other hand, you choose the IOAREA1/IOAREA2 form of double-buffering, and therefore do not want data management to move records to and from your work areas, you must establish an I/O register to keep track of the current buffer, using the IOREG keyword parameter. For an output file, data management's OPEN processing sets up the specified IOREG register to point to the address of one of the I/O areas; it changes this address to that of the other buffer after each output operation. Before you issue a PUT macro, therefore, you must move the data you want punched to the current I/O buffer, using the IOREG register to access the record. For input processing, after each GET macro you issue, the IOREG register points to the I/O buffer that contains the requested data just read from paper tape. You must use the IOREG register to access it.

When your DTFPT declarative macro is expanded, the assembler generates EXTRN pseudo-ops for the symbols you have equated to the IOAREA1 and IOAREA2 keyword parameters; therefore the coding that defines these storage areas may be assembled separately from the coding containing the macro. IOAREA2, if specified, must be of the same size as IOAREA1.

The length of the I/O buffers, defined by DS or DC statements somewhere in your BAL program, should be at least the number of bytes you have specified with the BLKSIZE keyword of this DTFPT declarative macro (17.5.1.3). It must be great enough to contain the longest of your undefined records, including the end-of-record stop character (EORCHAR keyword, 17.5.6), but not including shift characters. Buffer length should also be no less than the BLKSIZE specification for fixed, unblocked records without shifted codes. When, however, your fixed, unblocked records do contain shifted codes, you may specify that your I/O buffers are *larger* than your block size specification, via the OVBLKSZ keyword (17.5.1.5); when you do this, the storage areas you define for IOAREA1 and IOAREA2 must not be less than your OVBLKSZ specification indicates.

Keyword Parameter IOAREA1:

IOAREA1=symbol

Required for all paper tape files, to specify the symbolic address of the main storage area reserved for use as the primary I/O buffer for the paper tape file defined by this DTFPT declarative macro. The length of the buffer is defined elsewhere with a DS or DC statement. It must not be less than the OVBLKSZ specification; if the OVBLKSZ keyword is not specified, buffer length must not be less than the BLKSIZE specification.

If omitted, data management will not open the file defined by this DTFPT declarative macro, and an error message appears in your assembly listing. Refer to 17.5.9 for the resulting data management error processing.

Keyword Parameter IOAREA2:

IOAREA2=symbol

Specifies the symbolic address of a secondary I/O buffer; optional for all paper tape files. Length of IOAREA2 buffer must be the same as IOAREA1, and is subject to the same considerations. When the IOAREA2 keyword is specified, unless you specify work area processing via the WORKA keyword, you must also specify a general register to reference the current I/O buffer, using the IOREG keyword.

Keyword Parameter IOREG:

IOREG=(r)

Specifies the general register that is to be used to reference the current I/O area when both the IOAREA1 and IOAREA2 keywords are specified for double-buffering, and work area processing is not specified with the WORKA keyword. The completion, *r*, must be enclosed in parentheses; it represents the number of the general register used. Registers 2 through 12 are always available for use; if you specify the SAVAREA keyword (17.5.8), register 13 is also available. If you specify the IOREG keyword, you should not also specify the WORKA keyword; if you specify both, an error flag appears in the DTFPT expansion, and the WORKA keyword is ignored.

Keyword Parameter WORKA:

WORKA=YES

Specifies that data management is to provide double-buffering via the IOAREA1 buffer and a user-specified work area. Data management moves an input record from the I/O area to the work area you specify as the second operand of the GET macro. It moves an output record from the work area you specify with the PUT macro to the IOAREA1 buffer. The length of each work area is defined with a DS or DC statement elsewhere in your BAL program; each must have a length at least equal to the number of bytes specified by your BLKSIZE keyword.

If you specify the WORKA keyword, you must not also specify the IOREG keyword. If you specify both, an error flag appears in the DTFPT expansion in your assembly listing, and the WORKA keyword is ignored. The work area processing described cannot take place in this event, nor when the WORKA keyword is omitted.

17.5.1.5. Specifying Oversized Buffers (OVBLKSZ)

To obtain more efficient processing of input or output files containing fixed, unblocked records that are letter/figure shifted, you may define I/O buffers in your program that are larger than your BLKSIZE specification. Recall that your specified block size equals the length of your logical record when you have specified RECFORM=FIXUNB, but that this length does not allow for the insertion of the shift codes by data management (17.5.1.3). When you are using oversized buffers, you must notify data management that you are doing so by specifying the OVBLKSZ keyword, which indicates at the same time how long your oversized buffers are.

What this does for you in processing input files is to allow data management to read in more characters than your fixed record length calls for. Data management removes shift codes and delete characters from this larger block until the "compressed" record thus formed in the buffer does equal your specification. (It has also translated the characters between the shift codes with the specified FTRANS and LTRANS translation tables.) If data management cannot create a record equal in length to your BLKSIZE specification from the data in the buffer, it reads in more. When it has thus created a record of this size, data management either leaves this record in the I/O buffer and returns to you, or it moves the record to your work area before returning to you. If you have defined I/O buffers large enough, this mode of processing reduces the overall number of I/O operations required to process your file. Figure 17—10 depicts the relationship of the various specifications.

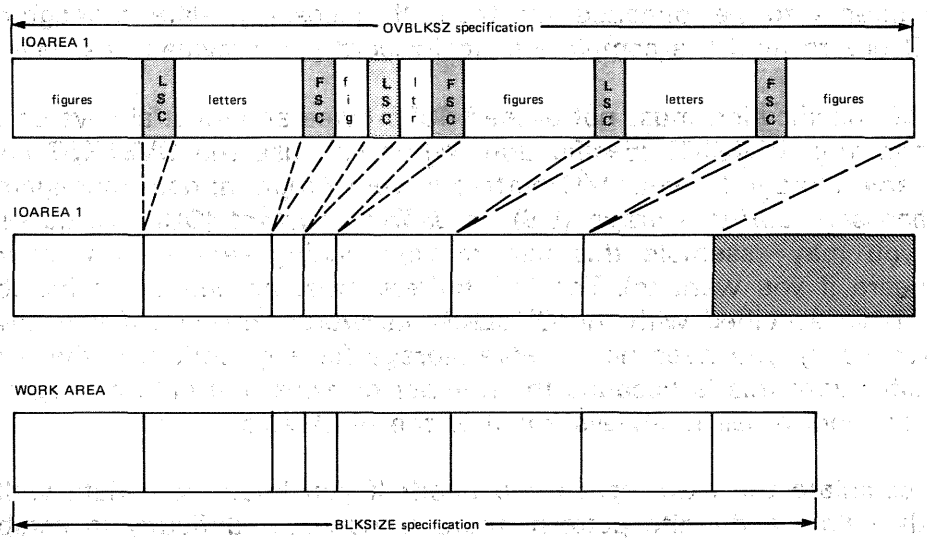


Figure 17—10. Relationships of Logical Record Length, Work Area Length, and I/O Buffer Length to the BLKSIZE and OVBLKSZ Specifications for a Fixed, Unblocked Record Input from Paper Tape with Shifted Codes (Part 1 of 2)

LEGEND:

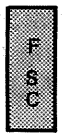
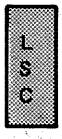


Figure shift character



Letter shift character

Bytes in I/O area not to be processed by user

NOTES:

1. The upper diagram depicts the record as originally read into the oversized I/O buffer; it contains untranslated tape codes and letter/figure shift codes. The hardware delete character has not been transferred to main storage, and data management has removed all software deletes before translation.

2. The center diagram shows the record as data management makes it available to you, left-justified, in the I/O buffer; the letter and figure shift codes have been removed and intervening data translated into EBCDIC. Notice the bytes in the oversized buffer extending beyond your BLKSIZE specification: these are extraneous to your logical record, and you should not access them.

3. The lower diagram shows the record moved into your work area, the length of which should be no less than your BLKSIZE specifiction. The record is left-justified in the work area, also.

Figure 17—10. Relationships of Logical Record Length, Work Area Length, and I/O Buffer Length to the BLKSIZE and OVBLKSZ Specifications for a Fixed, Unblocked Record Input from Paper Tape with Shifted Codes (Part 2 of 2)

For output files, the reverse takes place. Specifying the OVBLKSZ keyword allows data management more room to insert the required shift codes into the fixed-length record you have supplied. It continues doing so until the oversized buffer is filled, when it writes out the buffer contents to be punched on tape. If necessary, data management issues additional I/O orders until the complete logical record you provided has been punched.

Your OVBLKSZ specification must not exceed 4095 bytes and must always be at least one byte more than your BLKSIZE specification. When you use the OVBLKSZ keyword, you must still reserve storage for your I/O buffers in your program; data management cannot issue the necessary *define storage* (DS) or *define constant* (DC) instructions for you. (Recall that you may assemble this part of your coding separately from your DTFPT declarative macro if you want to). The I/O buffers must be defined to include as many bytes as you have specified with the OVBLKSZ keyword; however, if you have specified work area processing, you need not reserve storage for any work area that exceeds your BLKSIZE specification; this is because the number of bytes that data management moves to or from your work areas is always equal to the block size.

How do you calculate the extra space you need? If you know your data well, and your logical records follow a definite pattern, there is no great difficulty in establishing the number of additional characters to allow for the shift codes your records require. Statistical sampling of your records, when you do not know their composition well, may help you estimate the space you need.

The worst case, of course, occurs when there is an exact alternation: every letter being followed by a figure, every figure by a letter. This situation doubles the length of your logical record. The best case occurs when only one shift code is required per record. Do not discount the possibility of timing several test runs, with an adequate sampling of data, using successively larger OVBLKSZ specifications until you are reasonably certain that you have the best trade-off of main storage space for increased processing speed in your application. Nor should you forget that your use of delete codes must also be taken into account.

If you specify the OVBLKSZ keyword, but do not specify the WORKA keyword, do not access data in the I/O buffers at a displacement greater than your BLKSIZE specification. A glance back at Figure 17—10 shows the possibility of accessing extraneous data if you do.

If you do not specify the OVBLKSZ keyword with fixed, unblocked records that contain shift codes, your records are processed in the I/O buffers within areas that are limited in their length to the BLKSIZE specification. Reserving larger buffer space is then without useful effect.

Keyword Parameter OVBLKSZ:

OVBLKSZ=n

Specifies processing input or output records in oversized I/O buffers, when buffers greater than your BLKSIZE specification are defined to increase processing efficiency of fixed, unblocked records containing shifted data. Here, *n* is the decimal number of bytes used for buffer length; *n* must be at least one byte greater than your BLKSIZE specification, but may not exceed 4095 bytes.

When you specify the OVBLKSZ keyword, you must define I/O buffers that have a length at least equal to *n*; buffer length greater than *n* is unused. Work areas, if specified, need be no longer than your BLKSIZE specification.

The OVBLKSZ keyword may not be used if RECFORM=FIXUNB is not specified, explicitly or by default, or if records do not contain shifted data.

If omitted, records are processed in the I/O buffers within areas limited in length to your BLKSIZE specification.

17.5.1.6. Specifying Register for Record Size (RECSIZE)

When you are processing an output file containing undefined records, you must specify a general register in which you will place the length of each record before you issue the PUT imperative macro. The use of a record size register is optional for input processing of undefined records; if you specify such a register, data management places in it the length of the logical record that is available to you in the I/O buffer or in your work area before returning to you from its execution of your GET macro.

The record size placed by you or data management in this register is a fixed-point binary number, measuring the length of the record in bytes; it may range from 1 to 4094 bytes. This length does not include extra bytes for the end-of-record stop character (17.5.6) nor the shift codes, if any, that data management inserts. However, for output undefined records, you must include extra bytes for any characters you want punched at the end of the record to serve as an interrecord gap (17.3.4). Data management does *not* include record gap characters in the input record length it loads into the register.

The record size register is specified with the RECSIZE keyword and is used only in character mode (MODE=STD) for undefined records. Files processed in binary mode may not contain undefined records.

Keyword Parameter RECSIZE:

RECSIZE=(r)

Specifies the number of the general register that is to be used to refer to the size of undefined records, where *r* is the register number and must be coded in parentheses. General registers 2 through 12 are available for this use, as is register 13 when you have also specified the SAVAREA keyword.

The RECSIZE keyword is specified only when processing is in character mode (MODE=STD) and records are undefined (RECFORM=UNDEF). It is required for output processing; before issuing each PUT macro, you must place the length of the undefined record in the RECSIZE register. Use is optional for input processing; data management loads the RECSIZE register before returning to you from each GET macro.

17.5.2. Specifying File Processing Mode (MODE)

The 0920 paper tape subsystem itself operates in either of two modes: binary or nonbinary. The nonbinary mode of the hardware (for which the term *character mode*, used in this manual, is probably more descriptive) is the standard mode of operating the hardware in OS/3 data management.

17.5.2.1. Highlights of Binary Mode Processing (MODE=BINARY)

The binary mode is designed for use only with 1-inch, 8-level paper tape. In this mode, each of the eight bits of a byte in main storage that represents a character corresponds to a hole position on the tape. This correspondence is fixed within the hardware, and you cannot use the program connector board to change it — as you can for character mode.

In binary mode, because you cannot wire the board so that the subsystem can recognize the stop character that delimits undefined records (thus permitting an automatic stop of tape motion when the hardware has read in a full record), you are limited to fixed, unblocked record format. Another way in which the binary mode differs from character mode is that figure and letter shifting is possible only for character mode; you have no need of the keyword parameters used to specify shift codes in character mode, nor concern for the effects of these extraneous characters on the sizes of your buffers and work areas.

Just as the subsystem cannot recognize the stop character in binary mode, so also it cannot recognize a delete (or "rub-out") character. Data management, however, provides you the means for specifying what characters you want to use as deletes; this is the SCAN keyword parameter of the DTFPT declarative macro. When you have used this to specify one or more delete characters for an input file, each record is scanned as it is read in, and any such characters are deleted. As the characters are removed, the record is "compressed", and data management reads more characters from tape, if necessary, until the record of the fixed length you have specified is in your buffer. (For further detail, see the SCAN keyword (17.5.3).)

Although you cannot use shifted codes in binary mode, you may indeed specify that your data is to be translated. Data management translates your input or output data according to the table you provide in your program (or assemble separately), specifying its symbolic address via the TRANS keyboard parameter (17.5.3.2).

As to null characters in binary mode, once your input file has been read past the tape leader (which must contain only null characters: 17.2.2), any null characters encountered after the first non-null are transferred to main storage. If, for example, you are using nulls to denote an interrecord gap (17.3.4), these will appear in your I/O area or work area at the end of each record (recall that you must include them in calculating your BLKSIZE specification); your program must provide for any action that may be necessary. Similarly, because the null characters you must use for your paper tape trailer in binary mode (17.2.3) are also transferred to main storage, you must do what is necessary in your program to deal with them.

Finally, binary mode differs from character mode in that parity checking is not possible.You can neither punch a parity channel on tape nor check parity on an input tape with the hardware. On the contrary, the program connector board on the 0920 paper tape subsystem is completely bypassed when you are processing in binary mode.

Keyword Parameter MODE:

MODE=BINARY

Specifies that the input or output file defined by this DTFPT declarative macro is to be processed in binary mode. Record format must be fixed, unblocked. Data may be translated on input or output, but not shifted. On input files, all null characters occurring after the first non-null on the tape are transferred to main storage.

If the MODE keyword is omitted, data management assumes MODE=STD has been specified (17.5.2.2).

17.5.2.2. Highlights of the Character Mode (MODE=STD)

The character (nonbinary, or standard) mode of paper tape data management is intended for use with 5- through 7-level tapes in 11/16, 7/8, and 1-inch widths. The 7/8-inch width is limited to read-only processing.

In the character mode of processing, you may specify either the fixed, unblocked record format, or the variable-length, undefined format (17.5.1.2). For input files with undefined records, you must wire the program connector board to specify the end-of-record delimiter, and you may specify a record size register, into which data management loads the length of each record it reads in (17.5.1.6). For output files containing undefined records, the RECSIZE register is not optional; you must specify it and load it yourself with record length before you issue the PUT imperative macro to punch each record into the tape. You must also specify the end-of-record delimiter (using the EORCHAR keyword), which data management places at the end of each undefined record to cause tape motion to stop when this character is encountered after reading the record (17.5.6).

To allow more characters to be encoded on paper tape than would otherwise be possible with fewer than eight levels or tracks, data management provides a letter- and figure-shifting capability, described in detail with the keyword parameters that implement it (FSCAN, LSCAN, FTRANS, LTRANS, and SCAN in 17.5.3 and 17.5.5). Data management handles insertion of shift characters on output and deletes shift codes on input, translating intervening data. You specify the shift codes, which may be any of the codes that you can punch on tape with the number of levels (tracks) in use.

Through the TRANS keyword parameter (17.5.3.1 and 17.5.3.2), the character mode is provided with a translation capability that you may use for any but an input file with shifted codes. For the latter, a translation capability is provided through the SCAN, FTRANS, and LTRANS keywords (17.5.3). For output files with shifted codes, translation is performed after inserting shift codes; the shift characters themselves are not translated.

When you are processing in character mode, you may wire the program connector board so that the hardware recognizes one delete character in input files. If you need more than one delete character, you may specify the remainder of them with the SCAN keyword (17.5.3.1), and data management takes care of deleting these. Or you may specify all your deletes with the SCAN table and not have a hardware delete.

Parity signal generation and checking are both facilitated in the character mode of processing. You may wire the program connector board in the 0920 paper tape subsystem to punch an odd or even parity track on an output tape, or to check a parity track, again odd or even, that has been punched into an input tape. You may connect any photocell or punch actuator to any memory bit, except the most significant bit, of a byte in main storage. The parity signals are generated within the subsystem; when you are punching, the parity signal generated is based on all eight bits of the byte in main storage, even though fewer than seven tracks are actually being punched in the paper tape.

When the hardware detects an odd or even parity error on a character of an input record, the most significant bit of the byte in main storage that represents the character is set to binary 1, and data management performs the error processing detailed in 17.5.9.

Keyword Parameter MODE:

MODE=STD

Specifies that the input or output file defined by this DTFPT declarative macro is to be processed in the standard, nonbinary (character) mode. You can read and punch tapes with from five to seven data levels and may use either fixed, unblocked or undefined record format. You may suppress parity signal punching or checking or use the hardware to punch an odd or even parity signal on output tape or check parity on an input tape. In character mode, you may wire the program connector board so that the hardware recognizes one delete character and the end-of-record stop character. Null or delete characters in this mode are never transferred to main storage. Data may be translated on input or output and may contain shifted codes.

17.5.3. Letter/Figure Shifting and Translation, Input Files in Character Mode (SCAN, LTRANS, FTRANS)

The letter/figure shifting capability that data management provides you for use in character mode processing (MODE=STD) allows more data and control characters to be represented by the hole patterns you can punch in tape than would be possible without it.

Consider the 5-level paper tape which can offer only 2^5 distinct combinations of punches in its five tracks. These 32 hole patterns are enough to cover one case of the alphabet, but not the 10 numbers in addition — and this leaves out a null, a delete, and any other characters you might need.

If you assign two of the 32 hole patterns to the null and delete characters however, and two of the remaining 30 to shift codes (one the "letter shift", the other the "figure shift"), you have 28 patterns left for representing data. (Shift codes may be any of the 32.) But these 28 hole patterns can now represent 56 characters, if you establish that each pattern represents one character when it follows the letter shift code on a tape, but a second character when it follows the figure shift. This is exactly what you do with the SCAN, FTRANS, and LTRANS keywords in your DTFPT macro, with which you specify scan and translation tables for shifted input files. You can then handle one case of the alphabet, 10 numerals, and 20 other characters as well. You may place codes representing *any* symbols in *either* table; it makes no difference to data management.

One reason this is possible is that, although data is encoded on tape in a 5-hole system, you are representing it in main storage in an 8-bit system. One convention in OS/3 is data management's assumption that the first record on every paper tape begins with a "figure": one of the 28 characters that you have decided may be represented by the hole patterns that follow the figure shift code on paper tape. However, if your first record actually begins with a "letter" (one of the other 28 characters that the same set of hole patterns may stand for), the letter shift code must be the first non-null character on the tape. Data management automatically punches this on output tapes for you.

Another circumstance making character shifting and translation easier in OS/3 is that data management implements these with the BAL *translate* (TR) instruction and *translate and test* (TRT) instruction. Both of these rely on the ordering of the 256 configurations possible in an 8-bit system that is shown in the EBCDIC columns of Table C—1, and your scan and translation tables should be based on the same order of character positions.

For paper tape input files with shifted codes, in character mode, data management deletes the shift codes it encounters in input records and translates the data between them. (For output files, data management inserts the shift codes; this is described in 17.5.5.) The shift codes you use are entirely up to you; you may select any of the hole patterns that may be contained in the number of levels in the tape you are using.

When you have specified the fixed, unblocked record format (RECFORM=FIXTUNB) for a file containing shift codes, note that your records on paper tape are not actually fixed in length. Usually, each has been made longer by the insertion of one or more shift codes and thus exceeds the fixed length your record had when you provided it to data management to be punched. (For an illustration of this, refer back to Figure 17—5.)

When you are processing fixed, unblocked records with shifted characters, however, you can make for more efficient processing by reserving storage areas for I/O buffers that are somewhat larger than the fixed length you provide to data management with your specification of the BLKSIZE keyword parameter. When you do so for an input file, data management is able to read in more characters at a time; the delete and shift characters are removed, shifted characters are translated as required (and additional tape reads performed, if necessary), until the number of data characters in the buffer equals your BLKSIZE specification. Then the data is made available to you in the buffer, or moved to your work area. To specify this procedure, which speeds up processing by permitting fewer I/O operations to be issued by data management, you specify the OVBLKSZ keyword in your DTFPT macro (17.5.1.5).

Keyword Parameter SCAN:

SCAN=symbol

Specifies the symbolic address of your input file shift code scan table. Required in character mode (MODE=STD) for input files with shift codes; the FTRANS and LTRANS keywords must also be specified whenever the SCAN table specifies which are the shift codes.

An optional use of the SCAN table in character mode is to specify one or more delete characters (instead of or in addition to the one delete character you may specify by wiring the program connector board, which you then do *not* include in the SCAN table). When you use the SCAN table for deletion only, you do not specify the FTRANS or LTRANS keyword, but you may specify the TRANS keyword (17.5.3.1).

The SCAN table, whose address is specified by *symbol*, need be only as long as the number of paper tape codes in the set to be read. It contains a 1-byte entry for each of these; all are hexadecimal 00 except those used to specify the figure shift character, the letter shift character, and the "software" delete characters (if any — there may be one, none, or many). The nonzero entries in the SCAN table are:

Hexadecimal Code	Use
04	Defines the figure shift character; is placed in the byte position of the table that corresponds to the figure shift code
08	Defines the letter shift character; is placed in the table in the byte corresponding to the letter shift code
0C	Indicates a character to be deleted by data management. (This "software" delete is in addition to or instead of the "hardware" delete character that may be specified by wiring on the program connector board). There may be many software delete characters specified in the SCAN table; you may specify deletes with or without the use of a hardware delete character.

Refer to the coding example that follows the descriptions of the FTRANS and LTRANS keywords for an explanation of the SCAN table. See also 17.5.3.1 for a description of the use of the SCAN keyword to delete characters from records processed in binary mode (MODE=BINARY).

Keyword Parameter FTRANS:

FTRANS=symbol

Specifies the symbolic address of your input file figure translation table; required (with the SCAN and LTRANS keywords) to process input files in the character mode (MODE=STD) that contain shifted characters. The label of the translation table is *symbol*.

The translation table specifies the correspondence between a character that is punched after the *figure shift* character on tape and the 8-bit code that data management is to place in your data area. Each position in the table corresponds to a different hole pattern on the tape, beginning with hexadecimal 00 (null, which is never transferred into main storage, but simply marks the first position in the table) and extending through hexadecimal 1F, 3F, or 7F, depending on the level of tape you are using.

The FTRANS table contains a 1-byte entry for each paper tape hole-pattern in the set to be read; all entries are hexadecimal 00 except those in the positions for the hole patterns that may follow the figure shift character on tape. The 1-byte entries for these hole-patterns may be the hexadecimal digits for the 10 EBCDIC numeral graphics (F1, F2, and so on); on the other hand, they may also be *any* codes of your choosing: alphabet, punctuation, ASCII numerals, or whatever you have decided will be "figures". (In the coding example that follows the description of the LTRANS keyword, the codes are hexadecimal F1, F2, etc, for the EBCDIC numerics.)

The FTRANS keyword should be specified only for files processed in character mode (MODE=STD); it is ignored if you specify it for binary mode. You must not specify it with the TRANS keyword; if you do, both FTRANS and TRANS are ignored. In either case, a diagnostic message appears in the DTFPT macro expansion in your assembly listing.

The assembler generates an EXTRN pseudo-op code for *symbol*, which allows you to assemble your figure translation table separately from the DTF if you want.

Keyword Parameter LTRANS:

LTRANS=symbol

Specifies the symbolic address of your input file letter translation table; required (with the SCAN and FTRANS keywords) to process input files in the character mode (MODE=STD) with shifted characters.

The translation table, whose label is *symbol*, contains a 1-byte entry for each paper tape code in the set to be read; all are hexadecimal 00 except those used to specify which tape codes follow the letter shift character on tape and therefore are to be translated as "letters." The 1-byte entries for these tape codes may be the hexadecimal digits for the EBCDIC graphics desired (see Table C—1) or, as a matter of convenience, their character representation. On the other hand, they may be anything you want to designate as "letters."

The LTRANS keyword should be specified only for files processed in character mode (MODE=STD); it is ignored if you specify it for binary mode. You must not specify it with the TRANS keyword; if you do, both LTRANS and TRANS are ignored (as well as SCAN and FTRANS, if these are specified). In either case, an error message appears in the DTFPT macro expansion in your assembly listing.

The assembler generates an EXTRN pseudo-op code for *symbol*; therefore, you may assemble the letter translation table separately from the DTF if you have reason to.

The following coding example illustrates how you might devise scan and letter/figure translation tables for an input file contained on 5-track paper tape, which provides only 32 possible hole patterns. The entirely arbitrary example assumes that you have decided on the following correspondences to the paper tape codes, represented by their hexadecimal positions, 00 through 1F:

Hexadecimal Code	Character
00	null
01 through 1C	"letters" (A-Z)
1D	letter shift code
1E	figure shift code
1F	delete
01 through 09	"figures" (1 through 9)
11	"figure" (0)

Having decided that you need only one delete character, you take care of it by wiring the program connector board, and therefore do not specify it as a software delete in the SCAN table.

Example:

```
   LABEL     OPERATION     OPERAND                    COMMENTS
   1         10            16
1. PAPTAP1   DTFPT         MODE=STD,
                           TYPEFILE=INPUT,
                           SCAN=SCAN1,
                           FTRANS=FTRANS1,
                           LTRANS=LTRANS1,
                              .
                              .
                              .
2. SCAN1     DS            0CL32
             DC            XL29'00'
3.           DC            XL2'0804'
             DC            XL1'00'
4. FTRANS1   DS            0CL32
             DC            XL1'00'
5.           DC            XL9'F1F2F3F4F5F6F7F8F9'
             DC            XL7'00'
6.           DC            XL1'F0'
             DC            XL14'00'
7. LTRANS1   DS            0CL32
             DC            XL1'00'
8.           DC            CL10'ABCDEFGHIJ'
             DC            CL10'KLMNOPQRST'
9.           DC            CL8'UVWXYZ.  '
             DC            XL3'00'
```

NOTES:

1. This is part of the DTFPT declarative macro defining an input paper tape file, PAPTAP1. Note that it is processed in character mode; letter/figure shifting is possible only in this mode. Keywords not relevant to the example are not shown.

2. SCAN1 is the label of the shift code scan table; you assign a 32-byte length attribute to this symbol because there are 32 possible codes on a 5-level paper tape. You have equated this symbol to the SCAN keyword in the DTF.

3. You have placed the hexadecimal code 08 in the 30th byte position in the table; this corresponds to the position of the letter shift code, 1D. In the next byte of the table, corresponding to 1E, you place the hexadecimal code 04 to designate 1E as the figure shift code. All other positions in the SCAN table contain the hexadecimal code 00; you have omitted the code 0C because you have no need for a software delete.

4. FTRANS1 is the label of the figure translation table; like the SCAN table, it is 32 bytes long. You have equated this symbol to the FTRANS keyword in the DTF.

5. Having assigned the 1-byte hexadecimal entry 00 to the first byte of the table (to take care of the null character assignment), you then assign the entries F1, F2, and so on, to the next nine bytes. This takes care of the EBCDIC numerial graphics 1 through 9.

6. Assigning hexadecimal 00 to the next seven bytes, you assign the entry F0 to the next byte. Thus the numeral zero is represented on tape by the punch pattern corresponding to the hexadecimal position 11, when 11 follows the figure shift code, 1E. As indicated by the assignment of hexadecimal 00 to the 14 remaining bytes of the table, there are no further characters to be represented by tape codes that follow the figure shift code.

7. LTRANS1, equated to the LTRANS keyword in the DTF, is the label of the letter translation table for this input paper tape file; it also has a length attribute of 32 bytes.

8. Again, the first byte of the table is assigned the entry 0016 because this position is for the null character. To the next 10 positions, you assign the first 10 letters of the alphabet, declaring the string is a 10-byte character constant for convenience of documentation.

9. The next 10 being assigned in the same way, you conclude your assignment of the 28 available "letter" characters with eight more. The 7th is the period (.), and the 8th is the blank, to be represented on tape by the pattern in hexadecimal position 1C. The translation table concludes with hexadecimal 00 being assigned to the last three bytes; these three bytes (1D, 1E, and 1F) have been dedicated to the letter shift code, the figure shift code, and the "hardware" delete character.

The foregoing example assumes that you have not wired the program connector board in order to change the correspondence between hole patterns on paper tape and those data management encounters in main storage. As you know, you may connect any bit in main storage to any hole position on tape; whatever you do has a definite effect on the layout and content of your FTRANS and LTRANS tables and how the characters on tape are ultimately represented in your buffer.

Another point to be made is that the codes corresponding to shift and delete characters are not translated by either the FTRANS or the LTRANS table. The entries in these tables corresponding to the shift and delete codes may be hexadecimal 00 or any code: they are simply used to fill out the table completely.

17.5.3.1. Character Deletion, Input Files, in Binary or Character Mode (SCAN, TRANS)

As previously noted (17.5.3), an optional use of the SCAN keyword parameter is to specify a SCAN table that is dedicated to assigning one or more software delete characters, to be removed by data management from records read in from input paper tape files in character mode (MODE=STD). These software deletes are usually in addition to the hardware delete that you specify with the wired program connector board; however, you may specify all of your deletes in the SCAN table. When you dedicate the SCAN table to character deletion alone, you do not specify the FTRANS and LTRANS keywords, but you may specify the TRANS keyword parameter.

In addition to this use, you may specify the SCAN keyword parameter in the DTFPT declarative macro, not only for a file processed in the character mode (MODE=STD), but also for one processed in binary (MODE=BINARY). Doing so is the only means data management provides you for automatically deleting characters from records in binary input files; as you will recall, you cannot use the program connector board for specifying a wired or hardware delete when you are processing in binary mode (17.2.1.2).

When the SCAN table is used only to specify software delete characters, there are only two hexadecimal codes that may be used for entries: one is 0C, which you place in each byte position that represents a character to be deleted. The other hexadecimal entry, 00, you place in all the other bytes; these represent characters not to be deleted. The length of the table must equal the number of possible hole patterns that can be punched in the number of tracks or levels in your paper tape: 32 bytes for a 5-level tape, 64 for a 6-level tape, 128 for a 7-level tape, and 256 bytes for the 8-level tape that is used in binary mode.

In the following coding example, the programmer has specified a 256-byte SCAN table, the last six positions of which are assigned to delete characters.

Example:

LABEL (1)	OPERATION (10)	OPERAND (16)
SCANBIN	DS	0CL256
	DC	XL250'00'
	DC	6X'DC'

When your data in an input paper tape file requires translation before you process it (if, for example, it is encoded in ASCII), but also contains characters to be deleted, you may have data management delete them *before* translation by specifying both the SCAN and the TRANS keywords in the DTF. You may do so for both binary and character mode processing, but must not specify the FTRANS or LTRANS keyword.

In this use, the SCAN table you provide may contain only the hexadecimal code 0C, entered for each character to be deleted before translation, and the hexadecimal code 00, entered in the position for each code that is to be translated. In the TRANS table (an example of coding a TRANS table is given in 17.5.3.2), you may enter arbitrary filler codes in positions corresponding to the characters that the SCAN table causes to be deleted, because you will not see any translations for these in your I/O or work areas.

17.5.3.2. Translation for Input Files without Shifted Codes (TRANS)

You may use the TRANS keyword parameter to specify a translation table for any type of file except an input file *with* shifted characters. For the limited translation function provided for files of every kind, via the FTRANS, LTRANS, and SCAN keywords, see 17.5.3. For the use of the TRANS keyword for output paper tape files, see 17.5.5.

The TRANS table that you code for an input file *without* shifted characters is a simple table, not exceeding 256 bytes in length, prepared in the form required for the BAL *translate* (TR) instruction. Essentially, it is a string of 1-byte (8-bit) codes that data management is to substitute in your I/O or work area for the codes originally read in. The number of codes in the string, then, equals the number of unique codes you expect to read from paper tape, less the software and hardware delete characters, which of course are never presented for translation.

When your input paper tape file is a binary file (MODE=BINARY), the 8-bit codes originally read in are the directly corresponding images of the hole patterns punched in the tape. For input files processed in character mode (MODE=STD), the 8-bit configuraitons originally placed in the I/O area are those that result from your wiring of the program connector board. In either case, data management uses these as indexes to your TRANS table, adding their individual values to the address of the table, and the 8-bit code found at each table location thus reached is transferred to the data area to replace the one originally encountered.

The following coding example shows how you might devise a TRANS table for an input paper tape file (without shifting) processed on 6-level tape in character mode. Assume that, of the 64 original tape codes, 26 represent uppercase alphabet, 26 the lowercase, 10 the numerals, and 2 the null and the delete. You now want to replace the lowercase letters with uppercase for your processing.

This is the original assignment of tape codes, which extend from hexadecimal 00 through 3F:

Hexadecimal Code	Character
00	null
01 — 1A	uppercase alphabet, A-Z
1B — 34	lowercase alphabet, a-z
35 — 3E	numerals, 0-9
3F	delete

Assume that the delete character is taken care of by wiring the program connector board (thus making it a hardware delete); this leaves 63 characters to be translated.

Example:

```
      LABEL     ΔOPERATIONΔ          OPERAND                    Δ
      1         10        16
1.    TRANSLC   DS        0CL63
2.              DC        XL1'00'
3.              DC        CL10'ABCDEFGHIJ'
4.              DC        CL10'KLMNOPQRST'
5.              DC        CL6'UVWXYZ'
6.              DC        CL10'ABCDEFGHIJ'
7.              DC        CL10'KLMNOPQRST'
8.              DC        CL6'UVWXYZ'
9.              DC        CL10'0123456789'
```

NOTES:

1. Assigns a 63-byte length attribute to the symbol TRANSLC, which is equated to the TRANS keyword parameter in the DTFPT declarative macro that defines the input paper tape file to be translated.

2. The null character, hexadecimal 00, is unchanged, but an entry for it is necessary in your TRANS table.

3-5. Likewise, the meanings originally assigned to the next 26 codes are unchanged; you still want the tape codes 01 through 1A to be translated as uppercase alphabet, A-Z. But note that you must confirm this by including the same information in your TRANS table as applied when the paper tape file was punched on tape.

6-8. Here is the only new information your TRANS table contains: the next 26 tape codes (hexadecimal 1B through 34) are now to be translated into the EBCDIC uppercase characters, from the lowercase characters that they represent in the records on tape.

9. The last 10 codes covered by this table are unchanged; they represent still the numerals 0 through 9. There is no entry for the 64th tape code, as it is the hardware delete character that your program will never see in the I/O area.

For further details on the preparation of translation tables to be used for the BAL *translate* (TR) instruction, refer to the assembler user guide, UP-8061 (current version).

Keyword Parameter TRANS:

TRANS=symbol

Specifies the symbolic address of your translation table for input or output files, where *symbol* is the label of your table. May be used for all paper tape files except input files that contain shifted characters.

The translation table you code is in the form required for the BAL *translate* (TR) instruction. Data management generates an EXTRN pseudo-op for *symbol*, so that you may assemble your table separately from the DTFPT declarative macro.

You must not specify the TRANS keyword when you specify the LTRANS or FTRANS keywords (or both). If you do so, an error message appears in the DTF expansion in your assembly listing, and data management ignores the following keywords: TRANS, FTRANS, LTRANS, AND SCAN (if the latter is specified).

You may specify the TRANS keyword with the SCAN keyword for input files in binary or character mode, using the SCAN keyword to delete characters that are not to be translated (17.5.3.1).

When you use the TRANS keyword to translate output files with shifted codes, the shift codes are not translated (17.5.5). For translating input files with shifted codes, use the FTRANS and LTRANS keywords (17.5.3).

17.5.4. Specifying the End-of-Tape Routine for Input Files (EOFADDR)

For every input paper tape file, you must code a routine to handle the end-of-tape condition. You must also use the EOFADDR keyword parameter in the DTFPT declarative macro defining each input file to specify the label of this routine to data management, which branches to it automatically whenever end-of-tape is sensed, and when you issue a GET imperative macro to an optional input file, having specified the OPTION keyword under conditions described in 17.5.7. You need not assemble your end-of-tape routine with the DTF, however: in expanding your DTF coding, the assembler generates an EXTRN pseudo-op code for the label of this routine, and you may therefore assemble it separately.

If you have only one strip of tape to read, your end-of-tape routine may simply terminate file processing by issuing the CLOSE imperative macro, as you must do this whenever you have completed all processing (17.4.2). On the other hand, if you need to read a number of strips of tape, you must anticipate that data management branches to your routine at the end of each of them, and that the paper tape subsystem itself goes into the stop state. If you want to return to a routine which has been reading and processing tapes, you have only to branch to the address contained in register 14, as this register holds the address of the next instruction after the last GET macro you issued. However, if you want to issue any imperative macros in your end-of-tape routine before you use this return address, you must remember to store and restore the contents of register 14 to preserve the return address it contained at the entry of your routine.

Remember that, in order for you to read the next tape, the operator must load it into the reader and press the RUN switch on the device. But, if a GET macro is issued to read the first record on the newly inserted tape before the operator can press the RUN switch, the physical IOCS issues him an operator message at the system console (DEVICE xxx STOP STATE RU?) to which he must reply "R" to read. (The reply "U" results in data management error processing, 17.5.9.) To avoid having the operator go from the reader to the console to read every tape, you could program into your end-of-tape routine a time delay long enough to allow him to mount a tape.

Recall that, in order to prevent a false end-of-tape condition being signalled before the last data character is read from paper tape, you must provide a 12-inch paper tape trailer (17.2.3).

Keyword Parameter EOFADDR:

EOFADDR=symbol

Required for all input files to specify the label of your routine for handling the end-of-tape condition, where *symbol* is this label. The assembler generates an EXTRN for label, so that your end-of-tape routine may be assembled separately. You must close the paper tape file when all processing is completed.

If you omit the EOFADDR keyword from the DTFPT macro for an input file (or if a valid EOFADDR routine is not present on file OPEN), data management does not open the file, but issues error message DM61, sets the *DTF error* flag (bit (0) and *error detected in OPEN* flag (bit 4) in *filenameC*, and branches to your error routine. See 17.5.9.

17.5.5. Translation and Letter/Figure Shifting, Output Files (FSCAN, LSCAN, TRANS)

Three additional keyword parameters may come into play in your DTF when you have output files with shifted codes to produce in character mode (MODE=STD):

- FSCAN (required), which specifies the label of your *figure scan table* for this output file. Data management uses the table to ascertain which code it is to punch on tape as the *letter* shift code, and to select out of your output data the groups of charcters for translation as "figures".

- LSCAN (also required), which specifies the label of your *letter scan table*, used by data management in the same way to identify the *figure* shift code and the groups of "letters" in your output data.

- TRANS (optional), which specifies the label of the *translation* table you have coded. This table assigns, to each of the characters that will appear in your output data, one of the hexadecimal tape codes that can be punched in the number of character levels existing in your tape.

The FSCAN and LSCAN tables are, in a way, reciprocals in that, with the FSCAN table, you specify the *letter* shift code (which must be nonzero), by placing it in each position that corresponds to a "letter", and indicate, by placing hexadecimal 00 in all the other positions of the table, which 8-bit configurations in your output data are to be treated as "figures" by data management. With the LSCAN table, you specify the *figure* shift code (also nonzero) in each position corresponding to a "figure" and indicate, by hexadecimal 00 in their positions, those configurations that are to be treated as "letters." These two scan tables, between them, must provide a 1-byte entry for every 8-bit pattern that you may place in an I/O or work area to be punched on tape. Both scan tables are prepared in the format expected as operand 2 of the BAL *translate and test* (TRT) instruction.

If you specify the TRANS keyword, you code the TRANS table in the format expected by the BAL *translate* (TR) instruction, assigning a hexadecimal tape code to each of the 256 possible 8-bit patterns that may appear in your output. To all remaining 8-bit configurations, which are not to be presented in your output data (this includes the EORCHAR stop character and the two shift codes which data management punches but never translates), you could assign the same tape code. This code may be the one that you will insert in your TRANS table to represent the nonprinting EBCDIC graphic character (opposite hexadecimal 40 in Table C—1), but you could use some other code.

The TRANS keyword is not required to produce output files with letter/figure shifting, but its use is a great convenience, especially with 5-level tape. If you do not use a TRANS table, you must work entirely with the data characters that you can punch directly on tape. This means that, with a 5-level tape code, you can have only the hexadecimal codes 01 through 1F in your data buffers. On the other hand, using a TRANS table allows you to work in a more convenient code — EBCDIC is only one example — and still use 5-level tape.

For full details on the preparation of tables to be used as operands of the BAL *translate* (TR) and *translate and test* (TRT) instructions, refer to the assembler user guide, UP-8061 (current version), but first consider the two following coding examples, which are based on these assumptions:

- Your output tape has five levels, offering 32 hole patterns; these patterns are to be represented in your translation and scan tables, and on the program connector board, by the hexadecimal codes 00 through 1F.

- You have 37 EBCDIC characters for which you will present 8-bit codes in your output data. These are the 26 uppercase alphabet characters (A-Z), plus the nonprinting space character (these will constitute your "letters") and the 10 numerals (0-9) (these will constitute your "figures").

- You will use the hexadecimal tape code 00 to represent the null character, 1B for the end-of-record stop character, 1C for the nonprinting space character, 1D for the letter shift code, 1E for the figure shift code, and 1F for the delete character.

Although you may not expect to output any delete characters when you create your paper tape file, you probably intend to use them in routine processing of the file, and therefore must provide room for at least one delete character in your initial assignment of codes. (Later for input processing, you may specify one delete with the program connector board or one or more with the SCAN keyword parameter.)

You must specify the end-of-record stop character with the EORCHAR keyword of the DTF for this output file; data management automatically punches this after the last data character in each undefined record (17.5.6).

Thus, the following correspondences will be used:

- Letters:

Graphic Symbol	Hexadecimal Representation in I/O Area	Hexadecimal Tape Code, After the Letter Shift Code, 1D
A	C1	01
B	C2	02
C	C3	03
D	C4	04
E	C5	05
F	C6	06
G	C7	07

Graphic Symbol	Hexadecimal Representation in I/O Area	Hexadecimal Tape Code, After the Letter Shift Code, 1D
H	C8	08
I	C9	09
J	D1	0A
K	D2	0B
L	D3	0C
M	D4	0D
N	D5	0E
O	D6	0F
P	D7	10
Q	D8	11
R	D9	12
S	E2	13
T	E3	14
U	E4	15
V	E5	16
W	E6	17
X	E7	18
Y	E8	19
Z	E9	1A
SP	40	1C

- Figures:

Graphic Symbol	Hexadecimal Representation in the I/O Area	Hexadecimal Tape Code, After the Figure Shift Code, 1E
0	F0	10
1	F1	01
2	F2	02
3	F3	03
4	F4	04
5	F5	05
6	F6	06
7	F7	07
9	F8	08
9	F9	09

In this example, any 8-bit configurations other than these that you present in the I/O area are to be punched as the nonprinting SP character: that is, *following the letter shift code*, as 1C. Taking its cue from the result of using the FSCAN table on your data, data management inserts the letter shift code automatically.

The end-of-record stop character, 1B, does not require either shift code; you must always be sure that neither it, nor the code 1C, nor either of the shift codes, ever appears in your output data for translation.

The first of the coding examples discusses your DTFPT declarative macro and your TRANS table; the second is devoted to your FSCAN and LSCAN tables.

Example:

```
LABEL    ΔOPERATIONΔ    OPERAND                                  Δ  COMMENTS
1        10             16                                                          72        80
1. PAPTAP2.  DTFPT   TYPEFILE=OUTPUT,                                                X
             MODE=STD,                                                               X
             RECFORM=UNDEF,                                                          X
             EORCHAR=X'1B',                                                          X
             FSCAN=FSCAN2,                                                           X
             LSCAN=LSCAN2,                                                           X
             TRANS=TRANS2,                                                           X
                  .
                  .
                  .
```

(cont)

Example (cont):

```
LABEL    OPERATION   OPERAND                    COMMENTS
1        10          16                         72                  80
TRANS2   DS          0CL256
         DC          193X'1C'
         DC          X'09'01020304050607080 9'
         DC          7X'1C'
         DC          X'09'0A0B0C0D0E0F101112'
         DC          8X'1C'
         DC          X'08'131415161718191A'
         DC          6X'1C'
         DC          X'10'00010203040506070809'
         DC          6X'1C'
```

NOTES:

1. This is part of your DTFPT declarative macro for the output paper tape file PAPTAP2, which is to be processed in character mode (MODE=STD). As PAPTAP2 contains records in undefined record format, you must specify the end-of-record stop character, which data management punches automatically after each record on tape, with the EORCHAR keyword parameter (17.5.6); this is the hexadecimal code 1B. The required BLKSIZE, IOAREA1, and RECSIZE specifications, as well as other keyword parameters not relevant to this example, are not shown.

2. You assign a length attribute of 256 bytes to the symbol TRANS2, thus embracing the following *define constant* (DC) statements. You have equated TRANS2 to the TRANS keyword parameter in your DTF; thus the DC statements constitute your translation table for the paper tape file PAPTAP2. It covers the 256 positions of Table C—1.

3. To the first 193 positions of Table C—1 (decimal 0 through 192), you assign the tape hexadecimal code 1C. Note that the nonprinting EBCDIC space character (hexadecimal 40) is embedded in this 193-byte string; because it is, the effect of your specification is that any of the first 193 EBCDIC characters (if they appear in your output data) are to be represented on tape by the hole-pattern chosen to represent the space.

4. To the next nine positions in the TRANS table (those corresponding in Table C—1 to the EBCDIC graphics for the alphabetic characters A through I), you assign the tape codes 01, 02, and so forth, through 09.

5. The next seven positions are not used, and any 8-bit configurations from this part of the table will be represented on tape by the space code, 1C.

6. To the next string of nine alphabetic characters (J through R), you assign the tape codes 0A, 0B, and so forth, through 12.

7. Eight unused positions follow these.

8. The last eight alphabetic characters, S through Z, are assigned the tape codes 13 through 1A. Tape codes 1B and 1C already have their assignments; codes 1D and 1E will be assigned with the FSCAN and LSCAN tables, as described in the next coding example. You have determined that the tape code 1F is to represent one delete character, but, because you will not use it in this output file and because (like 1D and 1E) it is not to be translated, you do not include a translation for it in the TRANS table.

9. The next six table positions are also assigned the space code, 1C.

10. To represent the 10 EBCDIC numerics in your output data, you assign the 10 tape codes shown here. These are the second assignments for the tape codes in question, which have these meanings when they follow the figure shift code on tape, and the previously assigned ones when they follow the letter shift code.

11. The last six positions of Table C—1 are provided for by assigning the space code, 1C.

The second coding example discusses the figure and letter scan tables you prepare for the same file. Remember that you need not assemble them, nor the translation table, with the DTF because an EXTRN pseudo-op is generated for the label of each of them.

Example:

	LABEL (1)	OPERATION (10)	OPERAND (16)	(72–80)
1.	FSCAN2	DS	0CL256	
2.		DC	27XL1'1D'	
3.		DC	XL1'00'	
4.		DC	212XL1'1D'	
5.		DC	XL16'00'	
6.	LSCAN2	DS	0CL256	
7.		DC	XL240'00'	
8.		DC	10XL1'1E'	
9.		DC	XL6'00'	

NOTES:

1. You assign a length attribute of 256 bytes to the symbolic address FSCAN2, the symbol you equated to the FSCAN keyword parameter in the DTF. This attribute embraces the four DC statements following it, and these constitute your *figure scan table* for the output file PAPTAP2.

2. The first 27 bytes of the figure scan table, covered by this *define constant* (DC) statement, contain the 1-byte hexadecimal entry 1D, which, solely because it is a nonzero entry, specifies to data management that the code 1D is the *letter shift code*. It must appear in each byte position of the scan table that represents a "letter".

Because the first 27 positions of Table C—1 do not correspond to any EBCDIC graphics, letters or otherwise, it may occur to you to question why the FSCAN table shows these as letter positions. The reason is that they are all to be represented on tape by the same code, 1C, which you have also assigned to the nonprint SP character in your TRANS table. Because you have included the SP character among your "letters", its code, 1C, must follow the letter shift code on tape.

Other than the letter shift code, the only entry that you use in the figure scan table is hexadecimal 00; this code must be entered in every byte position that does *not* represent a "letter". Not all of these will necessarily be "figures", of course, but all of the figure codes will be in this subset.

3. In the next byte, you insert the hexadecimal code 00 to indicate to data management that the corresponding tape code, hexadecimal 1B, is not a "letter". (It is also not a "figure"; recall that 1B is specified to data management, via the EORCHAR keyword in your DTF, as the end-of-record stop character. This has no translation, requires no shift code, and should never occur in your output data for translation.)

4. The next 212 bytes also contain the letter shift code, 1D, specifying that the tape codes in these positions represent "letters". These include not only your 26 EBCDIC alphabetic symbols, but also 185 substitutions of this graphic for every other character in this set of 212, all of which must occur on tape after the letter shift code.

5. The hexadecimal code 00 in each of the last 16 bytes of the figure scan table shows that these do not represent "letters". Recall that only the first 10 of these are actually to be translated as "figures".

When your output data is examined and tested by data management, character by character, your data serves, essentially, as operand 1, and the FSCAN table as operand 2, of the *translate and test* (TRT) instruction.

So long as any of the 8-bit configurations in decimal positions 240 through 255 of Table C—1 is encountered in your output data, or the one in decimal position 27 (this one should never be there, as it is equated to 1B), the result byte that data management locates in the FSCAN table is hexadecimal 00. Scanning may continue, and these configurations are selected out for translation with your TRANS table (shown in the preceding coding example) and the *translate* (TR) instruction.

The first nonzero result byte that data management encounters in your FSCAN table stops the scanning process; the *letter* shift code 1D (never translated) is placed by data management in the I/O area after the last figure translation, and then scanning resumes — but this time data management uses your LSCAN table in the same way to select the 8-bit configuration, or group of configurations, to submit for translation by your TRANS table.

6. You assign a 256-byte length attribute to the symbol LSCAN2, which you have equated to the LSCAN keyword in the DTF. This length embraces the subsequent DC statements, which constitute your *letter scan table* for the output file PAPTAP2.

7. In each of the first 240 bytes of the scan table, you enter the hexadecimal value 00. When these result bytes are tested by data management, using the TRT instruction, the all-zero pattern each contains allows the scanning and the translation processes to continue, with your TRANS table and the TR instruction. These 240 bytes include both shift codes, the delete, and the end-of-record stop; except for these (which should never appear in your output data for translation), the 240 bytes represent "letters", and therefore their translations should follow the letter shift code data management has already punched into the tape. Recall, however, that most of these translations are the "dead-end" substitutions of the space character code, 1C, for everything but the 26 alphabetic characters and the space itself.

8. You enter the *figure* shift code, 1E, into each of the next 10 bytes (decimal positions 240 through 249 in Table C—1) to designate these positions as "figures". Because 1E is a nonzero entry in a letter scan table, data management immediately recognizes it as the figure shift code. When any of these 10 bytes is encountered in this LSCAN table by data management's use of the TRT instruction on a byte of your output data, the scanning process stops. Data management first punches the figure shift code on tape, then punches the correct translation, and, having shifted from this scan table back to the FSCAN table, resumes scanning with it.

9. The last six bytes of the LSCAN table contain hexadecimal 00; the "dead-end" translation process continues if any of these bytes is reached by data management.

Keyword Parameter FSCAN:

FSCAN=symbol

Specifies the label of a figure scan table for an output paper tape file processed in character mode (MODE=STD), where *symbol* is the label. The table, which may be assembled separately from the DTF, is prepared in the form required for operand 2 of the BAL *translate and test* (TRT) instruction; it must contain a 1-byte entry for each 8-bit configuration that you might place in an I/O or work area to be punched on tape. Entries in the FSCAN table corresponding to "letters" must contain the nonzero hexadecimal code for the *letter shift* character; all others must contain hexadecimal 00.

The FSCAN keyword must be specified with the LSCAN keyword, to produce output files with shifted codes in character mode (MODE=STD). If only one of these two keywords is specified, a diagnostic message appears in the DTF assembly listing, and the specification is ignored. If specified when processing is in binary mode (MODE=BINARY), a diagnostic appears in the DTF assembly listing, and the specification is ignored.

Keyword Parameter LSCAN:

LSCAN=symbol

Specifies the label of a letter scan table for an output paper tape file processed in character mode (MODE=STD), where *symbol* is the label. The table, which may be assembled separately from the DTF, is prepared in the form required for operand 2 of the BAL *translate and test* (TRT) instruction; it must contain a 1-byte entry for each 8-bit configuration that you might place in an I/O or work area to be punched on tape. Entries in the LSCAN table corresponding to "figures" must contain the nonzero hexadecimal code for the *figure shift* character; all other must contain hexadecimal 00.

To produce output files with shifted codes in character mode (MODE=STD), you must specify both the LSCAN keyword and the FSCAN keyword. Refer to the foregoing paragraph for the effect of misspecifying these keywords.

Keyword Parameter TRANS:

TRANS=symbol

Specifies the label of a translation table you have coded for any paper tape file but an input file with shifted codes, where *symbol* is the label. The TRANS table may be assembled separately from the DTF and is coded in the form required for operand 2 of the BAL *translate* (TR) instruction.

When the TRANS table is used for output files with shifted characters, the shift codes are not translated, but punched automatically by data management.

Refer to 17.5.3 for details on the use of the TRANS keyword with input files.

17.5.5.1. Translation for Unshifted Output Files, Either Mode (TRANS)

When your output file does not contain shifted characters, but merely requires translation from the EBCDIC character data in your I/O or work area to the hexadecimal tape codes you can punch in the levels or tracks that exist in your tape, your task is simpler. You specify the TRANS keyword in the DTF and prepare a translation table that is limited to the EBCDIC codes you will use.

Consider the following example, which assumes a 5-level paper tape and records that contain only the 26 EBCDIC uppercase alphabet, the space, and three punctuation marks: the period and the left and right parentheses. You will not punch out a delete character in creating this output file, but will reserve one tape code, hexadecimal 1F, for punching into the tape by other means as a rub-out character and possible specification later as a hardware or software delete when you read this file in future input processing. For the null character, you set aside the hexadecimal tape code 00, as before. Your 32 available hexadecimal tape codes extend from 00 through 1F; you need only the first 234 decimal positions of Table C—1 to cover the EBCDIC characters through Z.

Example:

```
     LABEL     ΔOPERATIONΔ          OPERAND                    Δ
     1         10         16
1.   TRANSOUT  DS         0CL234
2.             DC         XL1'00'
3.             DC         74XL1'01'
4.             DC         XL1'02'
5.             DC         XL1'01'
6.             DC         XL1'03'
7.             DC         15XL1'01'
8.             DC         XL1'04'
9.             DC         99XL1'01'
10.            DC         XL9'050607080909A0B0C0D'
11.            DC         7XL1'01'
12.            DC         XL9'0E0F101112131415 16'
13.            DC         8XL1'01'
14.            DC         XL8'1718191A1B1C1D1E'
```

NOTES:

1. You assign a 234-byte length attribute to the symbol TRANSOUT, which is equated to the TRANS keyword in your DTF (not shown). This length attribute covers the 13 subsequent DC statements, which constitute your translation table.

2. You insert the hexadecimal tape code 00 in the first byte of your TRANS table. This is unnecessary for an output file (because this code represents the null character, which you do not expect to include in an output record), unless you intend to place a certain (fixed) number of null characters at the end of each record in your output to indicate an interrecord gap (17.3.4). If you are not using the null character in this way, you would include this byte with the next 74.

3. To the next 74 bytes, you assign the hexadecimal code 01. Note that the EBCDIC space character SP is included in the string: this code, then, represents the space and is also assigned to all EBCDIC characters not used.

4. The *device constant* (DC) statement assigns your next tape code, hexadecimal 02, to the EBCDIC period.

5. The EBCDIC less-than character is assigned the same nonprint code, hexadecimal 01, as the other characters to be skipped.

6. The left parenthesis is assigned the hexadecimal tape code 03.

7. Fifteen more EBCDIC codes, unused, are assigned the nonprint code 01.

8. Here, the right parenthesis is assigned hexadecimal 04.

9. Ninety-nine more EBCDIC codes, through the left brace, are skipped.

10. To the next nine bytes, representing the EBCDIC alphabetic characters A through I, are assigned the hexadecimal tape codes 05, 06, and so on, through 0D.

11. The seven subsequent EBCDIC characters, through the right brace, are assigned hexadecimal 01.

12. These nine tape codes cover the alphabet characters J through R.

13. Eight more skips.

14. The last of the EBCDIC uppercase alphabet, S through Z, are assigned your remaining hexadecimal tape codes, 17 through 1E — leaving only hexadecimal 1F, for future use when a delete character is needed.

17.5.6. Specifying the End-of-Record Stop Character for Output Files (EORCHAR)

When you are processing in character mode (MODE=STD) and your file contains records with varying lengths (RECFORM=UNDEF), you have need of a character to delimit these records. With output files, you must specify this character with the EORCHAR keyword for data management to punch at the end of each undefined record when you issue the PUT macro; recall that you must also designate one general register into which you load the length of each undefined record before you issue the macro (the RECSIZE register, 17.5.1.6).

The end-of-record stop character causes tape motion to stop automatically when this character is encountered while an input file containing undefined records is being read. Recall that, for input processing, you must specify this character by wiring the program connector board (17.2.1.2).

For output processing, the character you specify with the EORCHAR keyword may be represented by any of the tape codes you may punch in the number of tracks on your tape; however, you should not use the null character, nor, in fact, any of the codes to which you have assigned a special meaning (such as the delete, or one of the shift codes). The EORCHAR stop character must be excluded from translation tables, as it has no translation, and it must be independent of shift status; therefore, it must also lie outside the range of "figures" and "letters" designated by your scan tables. In fact, you must take pains to exclude the EORCHAR stop from your own output.

For an output file, data management automatically inserts the EORCHAR stop character in your I/O area before it writes the buffer contents out to the punch; your BLKSIZE specification must always include one extra byte for it. When an undefined record is transferred to your I/O area from an input file, the EORCHAR stop character comes with it, and the buffer space you reserve must accommodate this byte; however, the record length that data management places in your optional RECSIZE register does not include the EORCHAR byte. Figures 17—2, 17—3, and 17—4, in 17.3, illustrate these points.

Keyword Parameter EORCHAR:

EORCHAR=expression

Specifies the end-of-record stop character used to delimit each undefined record; required for output files processed in character mode (MODE=STD) when RECFORM=UNDEF is specified. Here, *expression* is either an expression of self-defining terms or a symbol that is equated to an expression of self-defining terms.

Data management places the character defined with the EORCHAR keyword at the end of each undefined output record; it is the stop character punched on tape to delimit each undefined record.

Examples:

```
   LABEL     ΔOPERATIONΔ        OPERAND                Δ
   1         10        16
1. ......    ......    EORCHAR=X'03'
2. ......    ......    EORCHAR=C'P'**X'0F'
3. ......    ......    EORCHAR=CHAREX
   ......    ......       :
   CHAREX    EQU       X'03'
   ......    ......
```

NOTES:

1. Specifies that the hexadecimal tape code 03 is to be punched by data management as the end-of-record stop character to delimit each undefined record on tape.

2. Specifies that the end-of-record stop character to be punched by data management as a delimiter after each undefined record on tape is the logical product (AND) of the EBCDIC character P and the hexadecimal value 0F (this works out to be hexadecimal 07).

3. Specifies that the end-of-record stop character is the expression of self-defining terms equated elsewhere in your program to the symbol CHAREX. The *equate* (EQU) directive for the symbol CHAREX, represented by the last line of coding in example 3, makes this example exactly the equivalent of example 1.

   Notice two points about this way of specifying the EORCHAR keyword: the coding that contains the *equate* (EQU) directive must be placed outside the DTFPT call itself, but it must be assembled with the coding that contains the DTFPT call. The assembler does not generate an EXTRN for *expression*.

17.5.7. Specifying Optional File Processing (OPTION)

Optional file processing for paper tape input or output files is much the same as for punched card files, described in Section 3. As with cards, an "optional" paper tape file is one that your program will not invariably punch or read every time it is executed; you notify data management that this is the case by specifying the OPTION keyword parameter in the DTF defining the file. The specification is simply OPTION=YES.

If you have specified the keyword, there are two circumstances that result in optional file processing by data management:

- You have specified the OPT positional parameter in the job control DVC statement in the device assignment set for the file, and the device is not available at execution time; or

- You have not provided a device assignment set (job control DVC and LFD statements) for the file.

For details on these job control statements, refer to the job control user guide, UP-8065 (current version).

This is what *optional file processing* of a paper tape file involves:

- If the file is an input file, the first GET imperative macro you issue to it results in an immediate branch to your mandatory end-of-tape routine, the label of which you must specify with the EOFADDR keyword (17.5.4). No I/O is performed; you must close the file, using the CLOSE macro (17.4.2).

- If it is an output file, the first PUT macro you issue results in an immediate return to your program, at the first instruction after the PUT macro. No I/O orders are issued by data management. if you have specified the IOREG or RECSIZE registers, data management sets these up so that you may process in exactly the same manner as if you were actually punching a paper tape. Again, you must close the file.

If you have not specified the OPTION keyword parameter, and one of the foregoing circumstances occurs, the paper tape file is not opened by data management and may not be processed. In the error processing that ensues, data management takes the following actions:

- Sets the *error in OPEN* flag, bit 4 of *filenameC*

- Issues error message DM21 to the log: "INVALID OR MISSING DEVICE ASSIGNMENT".

- Branches to your error routine or, if none is specified, returns to you inline, at the first instruction after the OPEN macro you issued to the file.

If you do not code an error routine but take an inline return, then all GET or PUT imperative macros that your program issues to the file result in further data management error processing, as follows:

- Data management sets the *invalid imperative macro* flag, bit 6 of *filenameC*; and
- Issues error message DM13 to the log: "ATTEMPTED ACCESS TO AN UNOPENED FILE".

These actions result for every issue of GET or PUT to the file. Because the file has not been opened, you need not issue the CLOSE imperative macro to the file. For further details on *filenameC* and other aspects of data management's error processing, refer to the ERROR keyword, 17.5.9.

Keyword Parameter OPTION:

OPTION=YES
Specifies that the input or output paper tape file defined by this DTFPT declarative macro is an "optional" file and is not required to be processed every time the program is executed. Data management performs *optional file processing* if this keyword is specified and either no device is assigned to the file, or you have specified the OPT positional parameter in the DVC job control statement for the file and no device is available at execution time.

If you omit the OPTION keyword parameter and one of the foregoing conditions exists, data management does not open the file, and you may not process it. Data management error processing results.

17.5.8. Providing a General Register Save Area (SAVAREA)

In common with the other OS/3 data management systems, paper tape data management requires a 72-byte area in main storage, aligned on a fullword boundary, in which to store your general registers while it is processing.

You must align and reserve storage for this area within your program, but you have two ways of providing its address to data management: loading its address into general register 13 before entering any data management imperative, or specifying the label of the area via the SAVAREA keyword in your DTF. You need only one general register save area per program, but if you specify the SAVAREA keyword in one DTF, you should specify it in all.*

**It is possible to write a valid program in which you preload register 13 with the address of a save area before you issue any imperative macros to a file whose DTF does not contain the SAVAREA keyword, and also, omitting the preload of register 13, issue imperatives to another file whose DTF does contain the SAVAREA keyword. The program could work; the trick might be (in a large program processing numerous files) to be sure which file required which coding. You could not open all files with the same OPEN macro, for example, in such a program, nor terminate them all with one CLOSE — and you might be hampered in your use of register 13 for the IOREG or RECSIZE register.*

When you specify the label of the general register save area with this keyword, you free register 13 for your own use and may, for example, use it for your IOREG or RECSIZE register (17.5.1.4 or 17.5.1.6); otherwise, only registers 2 through 12 are yours. Refer to 1.4 for the content and layout of the save area, which is useful to examine in program dumps or snaps. Note that register 13 is not included, however: if you want to see its contents in a snap, you must provide and load a specific storage area for them.

If the SAVAREA keyword is not present in the DTFPT declarative macro, data management assumes that you have preloaded register 13 with the address of a fullword-aligned, 72-byte general register save area before you issue any imperatives. If you have a BAL program, therefore, written for some other data management system (such as the 9200/9300) in which you are using register 13 for your own purposes, you may easily convert it to run under OS/3. You need merely add a 72-byte register save area, fullword aligned, to your program and specify its label to data management with the SAVAREA keyword.

Because the assembler, in expanding your DTFPT declarative macro, generates an EXTRN pseudo-op for each symbolic label specified via the keywords in the DTF, you may assemble the coding by which you define the register save area separately from the coding in which you define the file.

Keyword Parameter SAVAREA:

SAVAREA=symbol

Specifies the label of an area defined in main storage, fullword aligned and 72 bytes in length, in which data management stores the contents of your user general registers while it is processing, where *symbol* is the label.

Only one such area is required per program, but if you specify the SAVAREA keyword in one DTF, you should specify it in the DTF for every file your program accesses.

If the SAVAREA keyword is omitted, data management expects that, before you issue any imperative macro to a file, you have preloaded register 13 with the address of a 72-byte storage area, fullword aligned, in which it saves the contents of your general registers.

If you specify the SAVAREA keyword, register 13 is available for your own uses; however, its contents are not included in the general register save area for inspection in a snap or dump of your program. Refer to 1.4 for the layout and content of this area.

17.5.9. Data Management Error Processing, Paper Tape Files (ERROR)

When data management detects an error (hardware, software, or logical), it sets one or more bits in the error flag byte of your DTF file table, issues the appropriate numbered data management error message to the log, and branches to your error routine. If you have not coded a routine for error processing and specified its label to data management with the ERROR keyword parameter, control returns to you at the normal inline return point: the next instruction after the imperative macro just issued. This is the point to which data management would have transferred control if no error had occurred.

When data management branches to your error routine, register 14 contains the address of the normal return point, to which you may branch, after you have completed error processing, to resume processing your file. However, if you intend to issue GET or PUT imperative macros in your error routine, you must first store the contents of register 14 and then, having issued all your imperatives, restore register 14 to its initial value before you branch back to your program.

The error flag byte is decimal byte 50 in the file table generated by the assembler in its expansion of your DTFPT declarative macro; the assembler also generates an EXTRN pseudo-op for this byte, assigning it a label that is formed by concatenating the EBCDIC character "C" to your 7-byte logical file name — which is why it is called *filenameC*. If you want to examine it to see what bits were set, you can easily locate *filenameC* in a dump of your program area, because its address is contained in the allocation map and definitions dictionary produced by the linkage editor, or you can include *filenameC* in a snap taken by your error routine; however, it is more useful to access it dynamically. You may do so inline or from your error routine, using the label of the error flag byte as the first operand of a BAL *test under mask* (TM) instruction, for example, to determine which bits were set so that you may take appropriate action.

Each of the eight bits in *filenameC* has a special significance when set to binary 1 as an error flag; Table 17—2 summarizes these meanings. The subsequent paragraphs discuss the errors represented in more detail, with actions you should consider taking in your error routine or elsewhere.

Table 17—2. Significance of Bits in FilenameC, Paper Tape Files

Bit	Hexadecimal if Set Alone	Binary if Set Alone	Significance for DTFPT File	Other Bits Set (1)	Data Management Error Messages Issued to Log
0	80 (2)	1000 0000	DTF error	4	DM61: INVALID DTF FIELD, PARAMETER, OR PARAMETER COMBINATION
1	40	0100 0000	Wrong length error	—	DM25: WRONG LENGTH ERROR DETECTED
2	20	0010 0000	Unique (parity) error	—	(None)
3	10	0001 0000	Unrecoverable error	— (4)	DM23: UNRECOVERABLE I/O ERROR DETECTED
4	08	0000 1000	Error detected in OPEN	—	(None)
5	04	0000 0100	Error detected in CLOSE	—	(None)
6	02	0000 0010	Invalid imperative macro	—	DM13: ATTEMPTED ACCESS TO AN UNOPENED FILE, or:
				—	DM14: INVALID IMPERATIVE MACRO/ MACRO SEQUENCE
7	01	0000 0001	Invalid record size	4	DM17: INVALID BLOCKSIZE SPECIFIED, or:
					DM18: RECORD SIZE INVALID

NOTES:

(1) The "Other Bits Set" column shows only those bits invariably set by data management. Others may also be set, for example, to indicate which errors are detected during OPEN or CLOSE processing.

(2) Bit 4 is always set when bit 0 is set. The resulting binary configuration of *filenameC* is 1000 1000, and the byte then contains the hexadecimal value 88.

(3) When bit 4 is set with bit 7, the resulting binary configuration is 0000 1001, and *filenameC* contains the hexadecimal value 09.

(4) When an unrecoverable error is detected during OPEN processing, bit 4 is also set with bit 3, and *filenameC* contains the hexadecimal value 18. When detected during CLOSE processing, bit 5 is also set with bit 3; *filenameC* contains hexadecimal 14.

- *DTF error* (bit 0)

This bit is set by the OPEN transient overlay to indicate that a serious error has been detected in your DTF. Data management also issues error message DM61. The *error detected in OPEN* flag (bit 4) will also be set to binary 1. Your file is not marked open and cannot be processed.

The *DTF error* bit is set when you have not properly specified the BLKSIZE keyword parameter (17.5.1.3), have omitted the EOFADDR keyword for an input file (17.5.4), have omitted the IOAREA1 keyword (17.5.1.4), or have specified an invalid address in the DTF (that is, some label or symbolic address specified in your DTF is an invalid address — in this case one that is not within your job region).

You should check your DTF assembly listing for error flag messages and your linkage editor map for unresolved EXTRN symbols.

- *Wrong length error* (bit 1)

  This bit is set for input files with undefined (variable length) records; it indicates that data management has filled the I/O buffer with the number of bytes specified by your BLKSIZE specification (shift codes having been removed), but that the hardware has not detected the wired end-of-record stop character that delimits each undefined record.

  The *wrong length error* flag is also set for fixed, unblocked input files if the last record on the tape is not a full-sized record (that is, the number of bytes of data yielded by the final record, stripped of shift characters and deletes, must equal your BLKSIZE specification, or you will receive a wrong length error indication).

  Data management issues error message DM25 in either case. You may either stop processing and close your file, or process the assumed partial record and then, issuing another GET macro, branch to the normal return point to continue processing (remembering to store and restore register 14 as required).

- *Unique (parity) error* (bit 2)

  When a parity error is detected in reading an input paper tape, the physical IOCS issues a standard message to the operator, describing and locating the error. The operator is able to move the paper tape back to the beginning of the record and to retry the command; if the retry is successful, data management does not perform the error processing set forth here. If the retry effort fails, however, you may have recourse to further recovery attempts, as follows.

  The *unique (parity) error* bit is set only for input files, processed in character mode (MODE=STD); the file must have been created with a parity track punched on the tape, and the paper tape subsystem must have been set up (using the program connector board) to check the parity track for odd or even parity.

  Set to binary 1, this bit indicates detection of a parity error in one or more characters on tape. Furthermore, each character on which a parity error has been detected has its most significant bit set to binary 1 in main storage. Your options depend on whether your data contains shifted codes.

  For files with unshifted data, you have three courses open to you in your error routine:

  — You may stop processing records and close the file.

  — You may continue processing the record by branching to the normal return point, at the address contained in register 14.

  — You may store register 14 and issue another GET macro to skip the record containing bad parity and read in the next. If the next record is free of parity errors, you can restore register 14 and branch to the normal return point to resume the processing that was interrupted by the initial detection of parity error. On the other hand, of course, errors detected in the execution of your second GET macro will result in another branch to your error routine.

When parity errors are detected on files with shifted characters, however, your recovery action is somewhat different. Data management does not perform shift processing on the record, but leaves it in the I/O area. (Even though you may have specified work area processing, the record is not moved to your work area.) Its shift codes are not removed, nor the software deletes — nor are the intervening characters translated. Unless you want to stop processing and close the file, you must deal with the erroneous record in your error routine; to skip this record is risky at best, because the shift status is likely to be masked by the parity error, and your subsequent records cannot be assured of being processed correctly.

If you have specified an I/O index register with the IOREG keyword (17.5.1.4), you can locate the error record by referring to this register. On the other hand, if you have specified more than one I/O buffer but do not have an IOREG register, you may refer to the address of the error record that is contained in *filenameD*, a 4-byte field, fullword aligned, and addressed by concatenating the EBCDIC character "D" to your 7-byte file name. Do not modify the contents of *filenameD*.

After locating each character of the record that has a parity error and resetting its most significant bit to binary 0, you may perform the character shifting in your error routine, removing the shift codes and translating the characters between them as required. You should compress the record and leave it left-justified in the I/O area, or, if you have specified work area processing, you must yourself move the record to your work area (17.5.1.4).

You will use your SCAN table as operand 2 of the BAL *translate and test* (TRT) instruction, and your FTRANS and LTRANS tables as operand 2 of the *translate* (TR) instruction; refer to 17.5.3 for details on the use of these tables and instructions by data management. Remember also to take care of removing any of the software delete characters you may encounter in your error record.

- *Unrecoverable error* (bit 3)

  This bit, when set to binary 1, indicates that an unrecoverable hardware or software error has been detected. In most instances, the physical IOCS issues a message to the operator, such as "DEVICE XXX STOP STATE RU?". This message indicates that the paper tape subsystem is in the stop state. If the operator replies "U" to this message, data management branches to your error routine with the *unrecoverable error* bit set and issues error messges DM23 to the log. Under certain conditions, the *error detected in OPEN* or *error detected in CLOSE* bit (4 or 5) may also be set.

- *Error detected in OPEN* (bit 4)

  This bit is set when any errors are detected during the processing performed by the OPEN transients (17.4.1). The file is not opened, and you may not issue any imperative macros to process it. In your error routine, you should not attempt any further processing of the file and should terminate your program. It is not necessary to issue a CLOSE macro to the file.

Other bits may be set with this bit to indicate which error was detected. For example, if you have an invalid DTF, this is detected during OPEN processing, and both bit 0 and bit 4 are set; data management issues error message DM61. Or, if your specification of the BLKSIZE or OVBLKSZ keyword is not a positive decimal number in the range 1 through 4095 (or the OVBLKSZ specification does not exceed block size), the OPEN transient issues error message DM17, INVALID BLOCKSIZE SPECIFIED, and sets both bit 7 and bit 4. Again, if the error detected by the OPEN transient is unrecoverable, data management issues error message DM23 and sets both bit 3 and bit 4. Finally, for the circumstances under which the *error in OPEN* bit is set for an optional file, refer to the OPTION keyword, 17.5.7.

If you have not coded and specified the ERROR routine, but accept error returns inline, data management expects that you will check for errors and deal with them inline. When you do not do so, therefore, each imperative macro your program issues to process an unopened file results in further data management error processing. This includes setting the *invalid imperative macro* flag (bit 6).

- *Error detected in CLOSE* (bit 5)

  This bit is set when errors are detected during the processing performed by the CLOSE transients (17.4.2). Other bits may also be set to indicate which error was detected: for example, bit 3 if the error is unrecoverable.

  CLOSE processing is completed, and you may reopen the file.

- *Invalid imperative macro* (bit 6)

  This bit is set to indicate that you have issued an inappropriate imperative macro to process your file (for example, the GET macro to an output file, or a CNTRL macro, which does not exist in OS/3 paper tape data management). In this circumstance, data management also issues error message DM14, "INVALID MACRO/MACRO SEQUENCE", to the log.

  This bit is also set if you issue any imperative macro except OPEN to an unopened file — including one that could not be opened because of an invalid DTF or because of some other error detected during your OPEN processing. In this case, data management issues error message DM13, "ATTEMPTED ACCESS TO AN UNOPENED FILE".

- *Invalid record size* (bit 7)

  This bit is set only when you are processing an output paper tape file that contains undefined records (RECFORM=UNDEF); it indicates that the number you have placed in the mandatory RECSIZE register (17.5.1.6) is negative, zero, or larger than your BLKSIZE specification minus one byte (BLKSIZE-1). Data management does not punch the record on tape. If this bit is set, data management also issues error message DM18, "RECORD SIZE INVALID". Your error routine should cease processing and close the file.

17.5.10. Processing ASCII Paper Tapes (SCAN, TRANS)

In OS/3, neither the physical IOCS nor paper tape data management provides automatic translation facilities. How data comes in from paper tape and is represented in main storage (and vice versa) are determined solely by the way you set up your program connector board and by the hole patterns on the tape. A hole on the paper tape always corresponds to some bit in main storage.

On the other hand, translation of data, via translation tables that you supply, is always possible for every type of file, be it input or output, binary or nonbinary, with or without shifted characters. If you need to read a paper tape that has been punched in ASCII, or to punch ASCII characters on tape, you must provide the proper translation tables in your program.

The ASCII code is specified in *American National Standard Code for Information Interchange, X3.4—1968,* and is a 128-character, 7-bit code. Another standard, *American National Standard Perforated Tape Code for Information Interchange, X3.30—1971,* specifies the representation of ASCII in perforated tape. The perforations are arranged in eight longitudinal tracks, one for each of the seven information levels, and one for parity. The bits of ASCII are assigned to specific tracks, and the ASCII character represented by each 8-bit pattern is related to its corresponding column and row position in ASCII. The parity bit is always recorded on the number 8 track, and provides an *even* number of holes for each character.

Figure 17—11, adapted from a figure in the standard, depicts a portion of an 8-track paper tape on which have been punched a number of ASCII null characters (NUL — no punches except the feed, or sprocket hole), the 10 ASCII numerical characters, and the ASCII delete character (DEL). The rectangles above the diagram of the tape itself relate the standard hole patterns punched in the tape to the ASCII character positions in the columns and rows of the standard: the ASCII DEL character, for example, occupies column 7, row 15; the ASCII numeral 9 occupies column 3, row 9. (There is not shown in these rectangles the column/row position (0/0) that is occupied by the ASCII null character, NUL, because there are seven of these punched in the tape in the figure.)

The following example, showing how you might code a translation table and a scan table to read an ASCII paper tape in OS/3, using data management, assumes that:

- Your ASCII input file is processed in binary mode, and the contents of its records are limited to the ten ASCII numerical characters and the ASCII space character.
- These eleven ASCII characters are to be translated into the corresponding EBCDIC characters for your processing in OS/3.
- The tape also contains the ASCII NUL character and the standard delete character, DEL. (Recall that in binary mode, you must specify this as a "software delete".)

- The hexadecimal representations in main storage of the 13 codes punched on the ASCII tape are the following:

Hexadecimal	ASCII Character
00	NUL
20	SP
30	0
31	1
32	2
33	3
34	4
35	5
36	6
37	7
38	8
39	9
7F	DEL

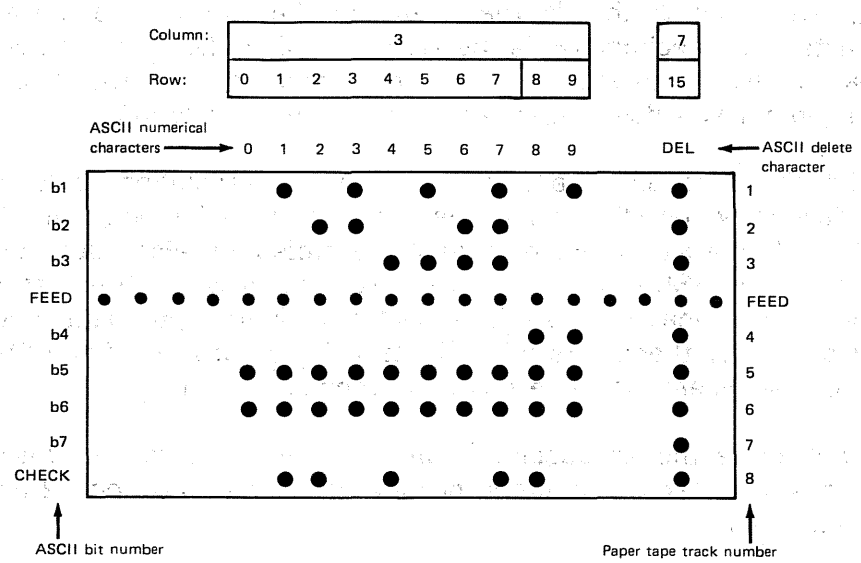


Figure 17—11. Portion of ASCII Punched Paper Tape, Showing Correspondence Between Hole Patterns and the Bits of the ASCII Code

Example:

```
   LABEL     OPERATION     OPERAND                          COMMENTS
1            10            16                               72        80
1. INASCII   DTFPT         TYPEFILE=INPUT,                  X
                           MODE=BINARY,                     X
                           TRANS=TRANSCII,                  X
                           SCAN=DEL7F,                      X
                              .                             X
                              .                             X
2. TRANSCII  DS            0CL127
3.           DC            XL32'00'
4.           DC            XL1'40'
5.           DC            XL15'00'
6.           DC            XL10'F0F1F2F3F4F5F6F7F8F9'
7.           DC            XL69'00'
                              .
                              .
                              .
8. DEL7F     DS            0CL128
9.           DC            XL127'00'
10.          DC            XL1'0C'
```

NOTES:

1. This is part of your DTFPT declarative macro, defining an input paper tape file, INASCII, processed in binary mode. Required keyword parameters not relevant to the example are not shown.

2. This is the *define storage* (DS) statement, coded elsewhere in your program, by which you assign a 127-byte length attribute to the symbol TRANSCII. Because you have equated this symbol to the TRANS keyword parameter in your DTF, data management recognizes the following *define constant* (DC) statements as constituting your translation table for this file. The table need not be 128 bytes long, even though this is the length of the ASCII code, because the 128th character (hexadecimal 7F) is the standard ASCII delete, which you must specify as a "software delete" to data management, via the SCAN table. Data management will delete this character before translation.

3. In the the first 32 byte positions of your translation table, you insert the hexadecimal value 00. This is the common code for both the EBCDIC and the ASCII null characters, but note that this statement also substitutes the EBCDIC null for each ASCII character in the remaining 31 of the first 32 positions in Table C—1. None of these is expected in your input data, and you have no translations for them. You might instead have substituted the EBCDIC space — but not the delete, because deletion precedes translation.

4. You substitute the hexadecimal value 40, which represents the EBCDIC space, for the ASCII space characters, occupying decimal position 32 (hexadecimal 20) in Table C—1.

5. The next 15 bytes of your TRANS table are also filled with the hexadecimal value 00, nullifying any unexpected ASCII maverick codes between the SP character and the first of the ASCII numerals.

6. Here you substitute ten EBCDIC hexadecimal values, F0, F1, F2, and so on, for the hexadecimal values expected in your input data for the ten ASCII numeral characters.

7. The remainder of the ASCII codes should not appear in your input data and are hence nullified — except 7F, the standard ASCII delete, which you provide for in the following scan table.

8. This *define storage* (DS) statement assigns a 128-byte length attribute to the symbol DEL7F, which you have equated to the SCAN keyword parameter in your DTF. Data management recognizes the next two DC statements as your SCAN table for the input file INASCII. This table must be 128 bytes long to include all 128 characters of ASCII.

9. You insert the hexadecimal value 00 in the first 127 bytes of this table. Here, this value has nothing to do with the ASCII NUL character: it ensures that a zero result-byte is encountered by data management when any of the hexadecimal values in the first 127 positions of Table C—1 is read in your input data, and the data is submitted to the BAL *translate and test* (TRT) instruction. Any of these 127 characters occurring in your input data is eligible for translation, using your table TRANSCII.

10. This statement inserts the hexadecimal value 0C in the 128th byte of your scan table. When the standard ASCII delete character, hexadecimal 7F, is encountered in your data, the result-byte data management accesses in the scan table contains this nonzero value. The character 7F is therefore not translated; instead, data management deletes it before testing your next byte of input data and thus "compresses" your record.

17.6. COMPARISON OF OS/3 WITH OTHER PAPER TAPE SYSTEMS

The OS/3 paper tape data management system is comparable with the paper tape systems in SPERRY UNIVAC Operating System/4 (OS/4) data management, SPERRY UNIVAC 9200/9300 Series Operating System IOCS, and the IBM System/360 Disk Operating System (DOS). The following paragraphs discuss areas of compatibility.

17.6.1. Compatibility with OS/4

OS/3 paper tape data management is compatible with the OS/4 paper tape system documented in the OS/4 data management system programmer reference, UP-7629 (current version). The maximum block sizes of the two systems are the same: 4095 bytes. The OS/4 ERRO keyword parameter is accepted by OS/3, but it is not implemented. All OS/4 keyword spellings are accepted as alternates by the OS/3 DTFPT declarative macro.

17.6.2. Compatibility with the 9200/9300 Series

All 9200/9300 Series DTFPT keywords, as documented in the operating system IOCS programmer reference, UP-7526 (current version), are accepted by the OS/3 DTFPT declarative macro. Of these, the following are accepted but not implemented; the remainder are implemented:

ATTN
CHNL
DEVA
FIGS
LTRS

Moreover, you should note that the 9200/9300 Series letter/figure shifting capability is not supported for input files by OS/3. To run 9200/9300 Series paper tape programs that process input files with shifted characters, you should remove the FIGS and LTRS keywords from the DTFPT declarative macro call, substituting the FTRANS, LTRANS, and SCAN keyword parameters and providing the necessary figure and letter translation tables and scan table elsewhere in your program.

Another point of difference is maximum block size. OS/3 allows 4095 bytes; the 9200/9300 Series allows only 256. Paper tape files created under the 9200/9300 paper tape data management system may be processed under OS/3; whether these should be restructured, or existing programs modified to exploit OS/3's 4095-byte maximum block size, are matters of judging the trade-offs between increased throughput and the programming effort involved.

A fourth point of difference is that OS/3 has no combined paper tape file capability. To punch a paper tape and then read the tape for error checking in OS/3, you should code two DTFPT declarative macros (one for input processing, one for output, using different file names) and should specify two separate job control DVC-LFD device assignment sets (one for each DTF).

17.6.3. Compatibility with IBM System/360 DOS

The OS/3 DTFPT declarative macro accepts all System/360 DTFPT keyword parameters documented in *IBM Systems Reference Library: DOS Supervisor and I/O Macros,* Twelfth Edition (February 1972), Order No. GC24-5037-11. Of these keywords, the following are accepted but not implemented in OS/3:

DELCHAR
DEVADDR
DEVICE
ERROPT
MODNAME
SEPASM
WLRERR

A second point of difference is maximum block size. IBM System/360 DOS provides 32,767 bytes; OS/3 allows 4095. Paper tape files containing fixed, unblocked records larger than 4095 bytes, or undefined records larger than 4094 bytes, must be restructured to be processed in OS/3, and existing programs exploiting the larger IBM maximum block size must be modified.

A third point of difference is that, unlike IBM System/360 DOS, OS/3 paper tape data management does not provide for skipping over strings of consecutive end-of-record stop characters without intervening data when processing input files containing undefined records.

PART 6. APPENDIXES

Appendix A. Functional Characteristics of Peripheral Devices

The tables in this appendix summarize the functional characteristics of the peripheral hardware available in the SPERRY UNIVAC Series 90 Data Processing Systems that are relevant to OS/3 data management usage.

Table A—1. SPERRY UNIVAC Card Reader Subsystems Characteristics (Part 1 of 2)

0717 Card Reader Subsystem	
Characteristic	**Description**
Card orientation (80-, 66-, and 51-column cards)	Face in, with column 1 leading, and row 9 down
Card rate	500 cpm (maximum)
Read technique	Dual redundant, solar cell technique using photo transistors. Column 0 amplifier checking
Read modes	Image mode: 160 six-bit characters per card Translate mode: 80 characters per card Available code: 8-bit EBCDIC
Read station sensing	Column by column
Hopper capacity	2400 cards
Stacker capacity	2000 cards
0716 Card Reader Subsystem	
Card orientation (80-, 66-, and 51-column cards)	Face in, with column 1 leading and row 9 down
Card rate	1000 cpm
Read technique	Dual redundant, solar cell technique using photo transistors. Column 0 amplifier checking
Read modes	Image mode — 160 six-bit characters per card Translate mode — 80 characters per card Three available codes: ■ 8-bit ASCII ■ 8-bit EBCDIC (required) ■ Compressed code
Read station sensing	Column by column
Hopper capacity	2400 cards
Stacker capacity Normal (stacker 2) Reject (stacker 1)	 2000 cards 2000 cards
0719 Card Reader Subsystem	
Card orientation (80-, 66-, and 51-column cards)	Face down, column 1 to left and row 9 facing away
Card rate	300 cpm
Read technique	Two columns of photosensitive sensors and light-emitting diodes Dual redundant. Column amplifier checking

Table A—1. SPERRY UNIVAC Card Reader Subsystems Characteristics (Part 2 of 2)

0719 Card Reader Subsystem	
Characteristic	**Description**
Read modes	Image mode: 160 six-bit characters per card Translate mode: 80 characters per card
Read station sensing	Column by column
Hopper capacity	1000 cards
Stacker capacity Normal Reject	1000 cards

Table A—2. SPERRY UNIVAC Card Punch Subsystems Characteristics (Part 1 of 2)

0605 Card Punch Subsystem	
Characteristic	**Description**
Media	80-column cards
Punch mode	2-column serial
Check mode	Punch motion check
Feed mode	On demand
Punch rate	75 cpm (full card) 160 cpm (28 columns only)
Input capacity	700 cards
Output capacity	700 cards (primary stacker) 100 cards (reject stacker)
Reading	Optional
Read rate	160 cpm
Punch translation Image mode Translate mode Available code: EBCDIC	 160 six-bit characters per card 80 characters per card
0604 Card Punch Subsystem	
Media	80-column cards
Punch mode	Row
Check mode	Read of punched data
Feed mode	On demand
Punch rate	250 cpm
Input hopper capacity	1000 cards

Table A—2. SPERRY UNIVAC Card Punch Subsystems Characteristics (Part 2 of 2)

0604 Card Punch Subsystem	
Characteristic	**Description**
Output stacker capacity	1000 cards (normal and select stacker)
Reading	Optional
Read rate	250 cpm
Punch translation Image mode Compressed mode	 160 six-bit characters per card 80 characters per card

Table A—3. SPERRY UNIVAC Printer Subsystems Characteristics (Part 1 of 5)

0773 Printer Subsystem			
Characteristic	**Description**		
Print speed	110 to 680 lpm, depending on character contingencies:		
	Available character sets	Characters sets per band	Nominal print rate (lpm)
	48-character business 63-character print 48/16-character print 85-character print 128-character special 96/(16-16)-character ASCII 256-character special	5 4 4 3 (plus 1 character) 2 2 1	500 400 400/670 310 217 217/500 114
Line advance timing	8.75 ms for spacing first line; for skipping each subsequent line: 3.33 ms at 6 lpi 2.50 ms at 8 lpi	8.75 ms for spacing first line; for skipping each subsequent line: 2.22 ms at 6 lpi 1.67 ms at 8 lpi	8.75 ms for spacing first line; for skipping each subsequent line: 1.67 ms at 6 lpi 1.25 ms at 8 lpi
Number of print positions	120 print positions (columns) by standard printer; 132 or 144 columns by feature		
Form advance control	Vertical format buffer		
Line advance rate	Single space only, 22 inches/second		
Form dimensions	3 to 18.75 inches wide 1 to 24 inches long		
Character set	Standard 48-character set. Any number of characters, up to 256, with options.		
Horizontal spacing	10 characters per inch		
Vertical spacing	6 or 8 lines per inch, operator-selectable		

Table A—3. SPERRY UNIVAC Printer Subsystems Characteristics (Part 2 of 5)

0770 Printer Subsystem			
Characteristic	**Description**		
Print speed	**0770—00**	**0770—02**	**0770—04**
	112 to 1435 lpm, depending on character contingencies 112 lpm — 384 contiguous characters 800 lpm — 48 contiguous characters 1435 lpm — 24 contiguous characters	213 to 2320 lpm, depending on character contingencies 213 lpm — 384 contiguous characters 1400 lpm — 48 contiguous characters 2320 lpm — 24 contiguous characters	337 to 3000 lpm, depending on character contingencies 337 lpm — 384 contiguous characters 2000 lpm — 48 contiguous characters 3000 lpm — 24 contiguous characters
Line advance timing	Advance and print	Time in ms	
		6 lpi	8 lpi
	1 line 2 lines 3 lines n+1 line	120.0 127.6 135.2 120+7.6n	118.0 123.7 129.4 118+5.7n
Number of print positions	Full print width of 132 print positions placed anywhere on a 16.5-inch form. With 22-inch form, only central 13.2-inch portion can be used (160 print positions with feature).		
Form advance control	Vertical format buffer		
Line advance rate	50 ips	75 ips	100 ips
Form dimensions	Continuous forms with standard edge sprocket-holes from 4 to 22 inches in width. Carbons may be attached or unattached with multicopy forms to a maximum of six parts. Recommended pack thickness not to exceed .0155 inch for high quality print.		
Character set	Standard 48-character set. Any number of characters up to 384 with options.		
Horizontal spacing	10 characters per inch		
Vertical spacing	6 or 8 lines per inch, as determined by program		

Table A—3. SPERRY UNIVAC Printer Subsystems Characteristics (Part 3 of 5)

0768 Printer Subsystem			
Characteristic	**Description**		
Print speed	**0768—00**	**0768—02**	**0768—99**
	900 through 1100 lpm	840 through 2000 lpm	1200 through 1600 lpm
Line advance timing	11.5 + 5.1 (n—1) ms — 6 lines per inch 11.5 + 5.7 (n—1) ms — 8 lines per inch where: n = number of lines advanced		
Number of print positions	132 character print positions including spaces		
Form advance control	Vertical format buffer and loop control; up to 132 lines per command		
Line advance rate	25 ips		
Form dimensions	4 to 22 inches wide 1 to 22 inches long		
Character set	0768—00 63 characters 0768—02 94 characters 0768—99 132 characters		
Horizontal spacing	10 characters per inch		
Vertical spacing	6 to 8 lines per inch		

Table A—3. SPERRY UNIVAC Printer Subsystems Characteristics (Part 4 of 5)

0776 Printer Subsystem

Characteristic	Description				
Print speed	Available character sets	Character sets per band	Nominal print rate (lpm)		
			0776—00, 01	0776—02	0776—03
	384	1	115	150	145
	192	2	225	290	280
	128	3	325	420	400
	96	4	420	540	520
	64	6	600	750	730
	48	8	760	940	900
	32	12	1030	1250*	1220
	24	16	1090*	1250*	1250*
			*For duty cycle reasons, maximum speed in lpm is limited by a minimum time between consecutive line feeds: 55 ms for the 0776—00, 01, and 48 ms for the 0776—02, 03.		
Line advance timing	Advance and print	Time in ms			
		6 lpi	8 lpi		
	1 line	16	14.2		
	2 lines	23.6	19.9		
	3 lines	31.2	25.6		
	4 lines	38.8	31.3		
	5 lines	46.4	37		
	n+1 lines	16+7.6n	14.2+5.7n		
Number of print positions	136 print positions including spaces				
Form advance control	Vertical format buffer				
Line advance rate	22 inches/second				
Form dimensions	4 to 18.75 inches wide 1 to 24 inches long				
Character set	Standard 48-character set. Any number of characters up to 384 with options.				
Horizontal spacing	10 characters per inch				
Vertical spacing	6 or 8 lines per inch, as determined by program				

Table A—3. SPERRY UNIVAC Printer Subsystems Characteristics (Part 5 of 5)

0778 Printer Subsystem			
Characteristic	**Description**		
Print speed	240 to 560 lpm, depending on character contingencies, at 6 lines per inch (lpi) (2.36 lines per cm), and single line spacing.		
	Available character sets	Nominal print rate (lpm)	
		Basic 00/01	Speed Upgrade 02/03
	48-character business	300	510
	64-character print	240	415
	48/16-character print*	240/255	415/560
	128-character print	120	240
	335-character print*	—	—
Line advance timing (in milliseconds)	Number of lines	6 lpi (2.36 lpcm)	8 lpi (3.15 lpcm)
	single	35 ms	35 ms
	double	53 ms	51 ms
	triple	61 ms	57 ms
Number of print positions	120 print positions (columns) per line; 136 columns by feature.		
Number of characters	Standard 48- or 64-character set, with five sets on a 240-character band; or up to 256 characters with expanded character set control feature; 48-character set band repeats five times resulting in 240 characters; 64-character set band repeats four times resulting in 256 characters.		
Forms advance control	Vertical format buffer		
Line advance rate	Single space only, 22 inches (55.88 cm) per second (slew rate).		
Ribbon feed	Bidirectional, self-reversing; ribbon removal without rewinding		
Ribbon type	Fabric or plastic film		
Codes	EBCDIC, ASCII, or any 8-bit code		
Form dimensions	Continuous single-part and multipart (up to six parts or up to 0.018 inch (0.457 mm) thick) with standard edge sprocket holes. Printer can also accept continuously sprocketed, 1-part card stock forms of weights typically used for punch cards, postcards, or offset masters. Form widths from 4.0 inches (101.60 mm) to 18.75 inches (476.2 mm) and lengths up to 24 inches (609.6 mm) can be accommodated. Forms longer than 17 inches (431.8 mm) can be run with casework door open, but noise level increases with door open.		
Horizontal spacing	10 charcters per inch (2.54 mm per print position)		
Vertical spacing	6 lpi (4.23 mm per line) or 8 lpi (3.17 mm per line), operator-selectable.		

*The "16" array is commonly a numeric subset. Extra 16 arrays are included in the 96/16—16 arrangement to make up a total number of 256 characters in a band.

Table A—4. SPERRY UNIVAC Disk Subsystems Characteristics

Characteristic	Description									
	8411 Disk Subsystem	8413 Diskette Subsystem	8414 Disk Subsystem	8415 Disk Subsystem	8416 Disk Subsystem	8418 Disk Subsystem	8424 Disk Subsystem	8425 Disk Subsystem	8430 Disk Subsystem	8433 Disk Subsystem
Data capacity (8-bit bytes)	7.25 million	242,944 bytes (using tracks 1—73 for data)	29.17 million	33.1 million per track	28.95 million	28.9 million or 57.9 million	58.35 million	58.35 million	100 million	200 million
Number of disk units	1 to 8	2 to 4	2 to 8	1 to 2	2 to 8	2 to 8	1 to 4	2 to 8	1 to 8 (with optional feature up to 16)	1 to 8 (with optional feature up to 16)
Disc/diskette speed	2400 rpm	360 rpm	2400 rpm	2800 rpm	2800 rpm	2800 rpm	2400 rpm	2400 rpm	3600 rpm	3600 rpm
Rotation period (ms/rotation)	25	166.7	25	21.5	21.5	21.5	25	25	16.7	16.7
Data bit rate	1.25 MHz	250 MHz	2.5 MHz	5.0 MHz	5.0 MHz	5.0 MHz	2.5 MHz	2.5 MHz	6.45 MHz	6.45 MHz
Bit density	1100 ppi	3268 ppi	2200 ppi	4040 fixed 4040 removable	4040 pulses per inch (ppi)	4040 ppi	2200 ppi	2200 ppi	4040 ppi	4040 ppi
Track density	100 tracks per inch (free format)	48 track per inch	200 tracks per inch	370 fixed 185 removable	192 tracks per inch	370 tracks per inch	400 tracks per inch	400 tracks per inch	192 tracks per inch	370 tracks per inch
Track capacity (byte)	3625	3,328 bytes per track	7294	10,240*	10,240*	10,240	7294	7294	13,030	13,030
Number of tracks	200 + 3 spare usable tracks per disc surface	77 total, 73 for data use per disc surface	200 + 3 spare usable tracks per disc surface	808+7 spare tracks 404+4 spare tracks	404 + 7 spare usable tracks per disc surface	404 or 808+7 spare usable tracks per disc surface	400 + 6 spare usable tracks per surface	400 + 6 spare usable tracks per surface	404 + 7 spare usable tracks per disc surface	808 + 7 spare tracks per disc surface
Number of surfaces per disk unit	10	1 surface	20	Data 3 Positioning 1 fixed Data 2 removable	Data 7 Positioning 1	Data 7 Positioning 1	20	20	19	19
Positioning time (seek time) Minimum Average Maximum	 25 ms 75 ms 135 ms	 — 83.33 ms —	 25 ms 60 ms 130 ms	 10 ms 33 ms 60 ms	 10 ms 30 ms 60 ms	 10 ms 27 ms 45 ms	 10 ms 30 ms 55 ms	 7.5 ms 29 ms 55 ms	 7 ms 27 ms 50 ms	 10 ms 30 ms 55 ms
Transfer rate	156 kilobytes per second	128 bytes in < 6ms	312 kilobytes per second	625 kilobytes per second	625 kilobytes per second	628 kilobytes per second	312 kilobytes per second	312 kilobytes per second	806 kilobytes per second	806 kilobytes per second

*In fixed 256-byte sectors, 40 sectors per track

Table A—5. UNISERVO Subsystems Characteristics

Characteristic	Description										
	UNISERVO 12		UNISERVO 16		UNISERVO 20	UNISERVO VI-C		UNISERVO 10		UNISERVO 14	
Tape units per subsystem	1 to 16		1 to 16		1 to 16	2 to 8		2 to 8		2 to 8	
Data transfer rate (maximum)	68,320 frames per second		192,000 frames per second		320,000 frames per second	34,160 frames per second		40,000 frames per second		96,000 frames per second	
Tape speed	42 7 inches per second		120 inches per second		200 inches per second	42.7 inches per second		25 inches per second		60 inches per second	
Tape direction Reading Writing	Forward or backward Forward		Forward or backward Forward		Forward or backward Forward	Forward or backward Forward		Forward or backward Forward		Forward or backward Forward	
Tape length (maximum)	2400 ft		2400 ft		2400 ft	2400 ft		2400 ft		2400 ft	
Tape thickness	1 5 mils		1.5 mils		1.5 mils	1.5 mils		1.5 mils		1.5 mils	
Block length	Variable		Variable		Variable	Variable		Variable		Variable	
Maximum block size (bytes)	65,535		65,535		65,535	8191		8191		65,535	
Minimum block size (bytes)	18		18		18	18		18		18	
Interblock gap	9-track 0.6 in.	7-track 0.75 in.	9-track 0.6 in.	7-track 0.75 in.	0.6 in.	9-track 0.6 in.	7-track 0.75 in.	9-track 0.6 in.	7-track 0.75 in.	9-track 0.6 in.	7-track 0.75 in.
Interblock gap time Nonstop Start-stop	14.1 ms 20.1 ms	17.6 ms 23.6 ms	5.0 ms 8.0 ms	6.25 9.25	3.0 ms 5.0 ms	14.1 ms 20.1 ms	NA	24 ms 30 ms	30 ms 38 ms	14 ms 20 ms	17 ms 23 ms
Pulse density	1600 ppi 800 ppi	800 ppi 556 ppi 200 ppi	1600 ppi 800 ppi	800 ppi	1600 ppi	800 ppi	800 ppi 556 ppi 200 ppi	1600 ppi 800 ppi	800 ppi 556 ppi 200 ppi	1600 ppi 800 ppi	800 ppi 556 ppi 200 ppi
Recording mode	Phase encoded	NRZI	Phase encoded	NRZI	Phase encoded	NRZI		Phase encoded or NRZI	NRZI	Phase encoded or NRZI	NRZI
Reversal time	25 ms		10 ms		16 ms	25 ms		16 ms		10 ms	
Rewind time	3 min		2 min		1 min	3 min		3 min		3 min	
Simultaneous operation	Optional		Optional		Optional	Optional		Optional		Optional	

Table A—6. SPERRY UNIVAC 0920 Paper Tape Subsystem Characteristics

Characteristic	Description
Reader mounting	Mounted on a 7- by 9-inch panel having a pin spindle for handling reels containing up to 50 feet of tape (for tape reader without an optional spooler)
Tape read	Unidirectional (right to left)
Tape channel capacity	Capable of reading 11/16-inch, 7/8-inch, or 1-inch paper tape; 3-position tape guide available to adjust to tape width used
Read speed	300 characters per second at 10 characters per inch
Type of tape	All conventional perforated tapes with a light transmissivity of 40% or less
Stop and start capacities	Can stop on character or before next character; on start, unit reaches full speed within two characters
Tape spooler	Up to 5-inch reels can be used with the spooler to allow reeling of approximately 300 feet of paper tape
Tape leader	Approximately 3 feet of tape leader required when spooler mechanism is used
Tape trailer	A 12-inch trailer must be provided to prevent false broken tape indication
Punch mounting	Mounted within a 14- by 19-inch panel
Tape channel capacity	Handles paper tape width of 11/16 inch or 1 inch; five levels of tape characters with 11/16-inch paper tape being used; or 5, 6, 7, or 8 levels of tape characters with 1-inch paper tape in use. Tape guide adjusts to conform to paper tape width.
Punch speed	110 characters per second at 10 characters per inch
Type of tape	Oil base paper tape is provided. A compatible tape utilizing a paper-plastic-paper sandwich is also available.
Stop and start capabilities	Punching is performed one character at a time. Tape punch is capable of stopping and starting between characters.
Tape feeding	The tape punch handles a paper tape reel of 1000 feet with sensing signals to indicate low paper tape supply.

Appendix B. Error and Exception Handling

B.1. GENERAL

All programs using the services of OS/3 data management execute imperative macroinstructions to obtain specific processing. OS/3 data management performs part of the desired data manipulation itself, but frequently calls on other OS/3 system programs (such as the physical IOCS or disk space management) to perform other parts of the task. Most of the time, the desired service is performed exactly as requested, and control is returned to you inline with no need to issue messages to the system console or to the log. Sometimes, however, errors or exceptions to desired performance occur; these may be detected by data management or the other system programs at various points in processing.

OS/3 data management is responsible for noting all errors and exceptions reported to it by the other system programs, as well as for testing, within itself, for other types of error or exceptions. When any such condition is detected, OS/3 data management will always:

- make appropriate entries in certain fields of the DTF file table, which your program may address in order to learn of exceptions and errors and take the proper course of action when control returns to you;
- log and display messages that call for operator intervention or are helpful in after-the-fact tracing of your program's action;
- branch to an error/exception routine in your program.

B.2. RETURN OF CONTROL

The design policy of OS/3 data management is never to terminate a user program. This means that data management will always return control to you after detecting an error or exception. If you provide the address of an error/exception routine in your DTF macroinstruction, data management returns control to this address for all conditions of error or exception. If you do not provide this address, data management returns control to you inline, at the next sequential instruction after the macro call. Retries by PIOCS have already been performed at this point in the processing.

B.2.1. Error Handling with ISAM

When OS/3 ISAM detects certain logical errors, the processor sets a bit in the DTFIS table that prohibits further reference to the file, other than to close it. These logical errors (also listed in Table B—3) are:

- Invalid macro sequence
- Invalid ID
- Invalid index search
- File space exhausted

B.3. SYSTEM ERROR MESSAGES

In OS/3, system error messages are contained in a general file of canned messages, which are listed or displayed under the control of the OS/3 supervisor. When OS/3 data management detects a loggable error, it acts through a logging transient routine to provide the supervisor with the proper code for the specific message to be logged, which the transient translates from an error code field in the DTF file table. This internal error code may also be accessed by your program; it is placed in byte 56 of the DTF file table by data management.

B.3.1. Data Management Error Messages

The error messages issued by OS/3 data management are shown in Table B—1, the first column of which lists the internal error code placed in byte 56 of every DTF file table *(filenameE)*. When these messages are printed or displayed, they will include, between the message number and the text, the 7-byte logical filename (LFD name) and the channel and device address which are maintained in the physical unit block (PUB) on which the file in question is assigned. Table B—1 also provides for each message an explanation of the probable cause of the error, a notation as to the data management module which issues it, and suggested actions by which you may recover from the error. Note, error messages relating to unit record also apply to the 8413 diskette.

Table B—1. OS/3 Data Management Error Messages (Part 1 of 6)

Internal Hexadecimal Code	Message Number and Text		Issuing Module	Suggested Action
01	DM 01	OPEN ISSUED TO OPENED FILE	OPEN	T, L
02	DM 02	filename REQUIRES channel/device vsn WITH RING R C	SAM tape	C (See note.)
03	DM 03	FCB NOT FOUND/INVALID	OPEN/CLOSE	T, C
04	DM 04	filename REQUIRES channel/device vsn WITH {NO BKNO / BKNO} R * C	SAM tape	C (See note.)
05	DM 05	I/O ERROR DETECTED WHILE ACCESSING VTOC	OPEN/CLOSE	T, V
06	DM 06	FORMAT-1 LABEL NOT FOUND	OPEN/CLOSE	T, V, C
07	DM 07	VOLUME SEQUENCE ERROR RIC	OPEN	O (See note.)
08	DM 08	FILE SERIAL NUMBER ERROR R*C	OPEN	T, C, (See note.)
09	DM 09	CREATION DATE ERROR RIC	OPEN	T,C (See note.)
10	DM 10	PREFORMAT WRITE ERROR DETECTED	OPEN/EXTEND	T, M
11	DM 11	SPECIFIED NON-EXTENDABLE	OPEN	T, C
12	DM 12	FILE SECURITY CHECK RIC	OPEN	H (See note.)
13	DM 13	ATTEMPTED ACCESS TO AN UNOPENED FILE	All DMS	T, L
14	DM 14	INVALID IMPERATIVE MACRO/MACRO SEQUENCE ISSUED	All DMS	T, L
15	DM 15	INVALID DTF, TYPE=nn (1)	OPEN/CLOSE	T, S, L
16	DM 16	PARTITION INVALID FOR SPECIFIED DTF, TYPE=nn (1)	OPEN	T, S, L

Table B—1. OS/3 Data Management Error Messages (Part 2 of 6)

Internal Hexadecimal Code	Message Number and Text		Issuing Module	Suggested Action
17	DM 17	INVALID BLOCK SIZE SPECIFIED	OPEN/SAM/NI UNIT RECORD	T, S
18	DM 18	RECORD SIZE INVALID	ISAM/SAM/NI UNIT RECORD	H, or T, D
19	DM 19	INVALID DEVICE CHARACTERISTICS SPECIFIED	OPEN	T, C
20	DM 20	NO BKNO SUPPORT IN SUPERVISOR	SAM tape	S, T
21	DM 21	INVALID OR MISSING DEVICE ASSIGNMENT OR DEVICE NOT AVAILABLE	UNIT RECORD	T, C
22	DM 22	HARDWARE ERROR – CHECK ERROR STATUS/SENSE BYTES	All DMS	T, M
23	DM 23	UNRECOVERABLE I/O ERROR DETECTED	All DMS	T, M
24	DM 24	INVALID REQUEST (ID) – EXCEEDS FILE LIMITS GETCS ERROR: –	Disc DMS	T, D
25	DM 25	WRONG LENGTH ERROR DETECTED	Disk DMS, SAM tape	T, M
26	DM 26	DATA CHECK DETECTED	Disc DMS	T, M
27	DM 27	READ ERROR ON RUNLIB DEVICE OR SPOOL FILE	OPEN	T, M
28	DM 28	PUNCH DOES NOT HAVE READ FEATURE	UNIT RECORD	T, S
29	DM 29	NO HARDWARE FOR STUB READ	UNIT RECORD	T, S
30	DM 30	VALIDITY CHECK ERROR	UNIT RECORD	H
31	DM 31	(No console message appears, but this code in the DTF means: record not found for random function.)	Disc DMS	H
32	DM 32	RECORD NOT FOUND FOR SEQUENTIAL FUNCTION	SAM/NI/ISAM/ IRAM/MIRAM	T, D
33	DM 33	INVALID FUNCTION ISSUED FOR OPTIONAL FILE	Disc DMS	T, L, C
34	DM 34	(No console message appears, but this code in the DTF means: end of data detected for sequential operation.)	Disc DMS	H

Table B—1. OS/3 Data Management Error Messages (Part 3 of 6)

Internal Hexadecimal Code	Message Number and Text		Issuing Module	Suggested Action
35	DM 35	ADD OF RECORDS RESTRICTED BY PREVIOUS OPERATION	ISAM add	H
36	DM 36	DUPLICATE RECORD—REJECTED	ISAM add	H
37	DM 37	SEQUENCE ERROR – RECORD REJECTED	ISAM load	H
38	DM 38	END OF DATA RETURNED BY SYSTEM—ILLOGICAL CONDITION	ISAM	T, D
39	DM 39	INVALID FILE CONDITION – INDEX INVALID	ISAM add and retrieve	T, R
40	DM 40	INDEX SPACE WILL NOT SUPPORT PRIME DATA	ISAM SETFL	T, R
41	DM 41	FILE SPACE EXHAUSTED	ISAM, SAM tape	T, R
42	DM 42	CHARACTER MISMATCH	Printer	T, D
43	DM 43	INVALID CONTROL CHARACTER	Printer	T, D
44	DM 44	LINE TRUNCATED	Printer	H, S
45	DM 45	EXTENT TABLE EXHAUSTED	Disc OPEN/EXTND UNIT RECORD	T, C
46	DM 46	filename channel/device vsn DATA BLKS: READ = nnnnnn EOV = nnnnnn IC	SAM tape	C (See note.)
47	DM 47	ERROR DURING LABEL PROCESSING	SAM/DAM/NI	H
48	DM 48	KEY LENGTH/LACE FACTOR INVALID	OPEN disc	T, S, C
49	DM 49	PROCESSING INHIBITED BY ERROR CONDITION	ISAM	T, L
50	DM 50	RECSIZE REGISTER NOT SPECIFIED FOR UNDEF FORMAT	SAM tape	T, S
51	DM 51	INVALID SUBFILE NUMBER SPECIFIED	NI	T, S
52	DM 52	NO SPACE AVAILABLE FOR SUBFILE ENTRIES	NI	S, T
53	DM 53	HARDWARE ERROR DURING FILE CONTROL BLOCK UPDATE	All disc	T, M
54	DM 54	INVALID JCL SPECIFIED OR INVALID USE OR NAME IN VFB OR LCB JOB CONTROL STATEMENT INVALID USE OR NAME IN VCB OR LCB STATEMENT FOR printer file	SAM tape UNIT RECORD	C

Table B—1. OS/3 Data Management Error Messages (Part 4 of 6)

Internal Hexadecimal Code	Message Number and Text		Issuing Module	Suggested Action
55	DM 55	STD SYSTEM/USER LABEL NOT FOUND	SAM tape UNIT RECORD	C, T
56	DM 56	FILE NOT FOUND	SAM tape	C, T
57	DM 57	WRITE ATTEMPTED ON UNEXPIRED FILE RIC	SAM tape, Disc DMS UNIT RECORD	C, T (See note.)
58	DM 58	FSN DOES NOT MATCH FIRST VOLUME VSN	SAM tape	C, T
59	DM 59	STD LABEL FIELDS DO NOT MATCH JCL SPECS	SAM tape	C, T
60	DM 60	TAPEMARK NOT FOUND AT FILE BOUNDARY	SAM tape	C, T
61	DM 61	INVALID DTF FIELD: PARAMETER, OR PARAMETER COMBINATION, TYPE=nn ①	All DMS	
62	DM 62	80 COLUMN CARDS READ WITH BLOCK SIZE 81-96	UNIT RECORD	W
63	DM 63	OPEN ERROR: BINARY MODE CARD INPUT FILE CANNOT BE SPOOLED IN	UNIT RECORD	C, T
64	DM 64	COMBINED CARD FILE CAN'T BE SPOOLED IN	UNIT RECORD	C, T
65	DM65	ILLEGAL KEY CHANGE DURING UPDATE	MIRAM	B
80	DM 80	BEGIN OUTPUT FILE PUNCH RECOVERY. R,U?	Read/Punch	O
81	DM 81	PERFORM PUNCH RECOVERY STEP 2A. R,U?	Read/Punch	O
82	DM 82	PERFORM PUNCH RECOVERY STEP 2B. R,U?	Read/Punch	O
83	DM 83	BEGIN OF FILE MARKER NOT COMPLETE.**I,C	All paper tape	O
84	DM 84	IS IT END OF FILE OR END OF TAPE. **F,T	All paper tape	O
85	DM 85	INSUFFICIENT SPACE ALLOCATED FOR PRINTER SKIP CODES	Printer	C
86	DM 86	SPOOL FILE FOR CARD READER FILE WAS NOT CREATED	UNIT RECORD	O
87	DM 87	UNRECOVERABLE ERROR WHILE LOADING THE VERTICAL FORMAT BUFFER OR LOAD CODE BUFFER.	Printer	O
88	DM88	jobname WAITING FOR LOCK lbl-file-name	FILE LOCK	W
89	DM89	DISKETTE DRIVE NOT AVAILABLE	UNIT RECORD	Y

Table B—1. OS/3 Data Management Error Messages (Part 5 of 6)

Internal Hexadecimal Code	Message Number and Text		Issuing Module	Suggested Action
90	DM 90	BEGIN ERROR RECOVERY ERROR CARD. R,U?	Read/Punch	O
91	DM 91	HAVE BLANK CDS BEEN PLACED IN HOPPER? R,U?	Read/Punch	O
92	DM 92	DO RECVRY STEP 2. REFILE LAST (2) CD(S)? R,U?	Read/Punch	O
93	DM 93	PERFORM PUNCH RECOVERY STEP 3. R,U?	Read/Punch	O
94	DM 94	PREPUNCHED CARD DETECTED DURING ERROR RECOVERY	Read/Punch	H
95	DM 95	PUNCH OFF-LINE.R,U?	Read/Punch	O
96	DM 96	PUNCH MISFEED. R,U?	Read/Punch	O
98	DM98	LOGICAL END OF FILE REACHED RI*	Tape extend	O
99	DM99	ILLEGAL EXTEND, STANDARD LABEL NOT SPECIFIED	Tape extend	S
9A	DM9A	ILLEGAL EXTEND, HDR2 NOT FOUND	Tape extend	S
9B	DM9B	ILLEGAL EXTEND, EOF1 OR EOV2 NOT FOUND	Tape extend	S
9C	DM9C	ILLEGAL EXTEND, RECFORM INVALID	Tape extend	S
9D	DM9D	ILLEGAL EXTEND, RECSIZE INVALID	Tape extend	S
9E	DM9E	ILLEGAL EXTEND, BLKSIZE INVALID	Tape extend	S
9F	DM9F	ILLEGAL EXTEND, FILE FOLLOWS THE FILE TO BE EXTENDED	Tape extend	S

NOTE:

Operators choose one of the following action messages to reply to data management error messages.

R Retry after mounting correct volume.
I Ignore the error condition.
C Cancel job.
U Unrecoverable; user error routine required.
(1) nn is a message type subcode that is used to provide additional information as to why the associated message was displayed or printed. Refer to Table B—1A for the subcodes and their explanations.

Table B—1. OS/3 Data Management Error Messages (Part 6 of 6)

LEGEND:

Suggested actions

B. Check your data and rerun the job.

C. Control stream content should be checked.

D. Checking of program dump is recommended.

H. Program should handle this occurrence, proceeding or otherwise as programmed.

L. Program logic should be checked.

M. Maintenance check of disk pack check.

O. Operator intervention is needed. See system messages operator/programmer reference, UP-8076 (current version).

R. Reorganize file, getting more space or rebuilding index by new load.

S. Program specifications to data management should be checked.

T. Program termination is recommended.

V. VTOC should be printed out for check.

W. Warning message.

Y. Rerun when device is available.

Table B—1A. Data Management Error Message Subcodes (Part 1 of 2)

Associated Data Management Error Message	Message Type Subcode*	Explanation
DM15	01	Invalid DTF address
	02	Invalid DTF type code
	03	Invalid DTF partition control appendage (PCA)
	04	Invalid DTF partition control appendage (PCA) address
	06	File type code in DTF does not match type code in IOCS processor.
DM16	01	Wrong key location
	02	Invalid DTF address in partition control appendage (PCA)
	03	Missing extent table entry for partition control appendage (PCA)
DM61	01	Single mount specification does not match specification used to create file. Single mount specifications do not match between Format 2 label and DTF
	02	Variable record specifications do not match between Format 2 label and DTF
	03	Two I/O areas are not contiguous. Index buffer not contiguous with I/O area 1. I/O area 2 address not contiguous with I/O area 1 address. Index buffer not contiguous with I/O area 1 address.
	04	Index operations intended, but no index buffer or key argument specification Seek address not specified Key argument not specified Index buffer not specified
	05	Key location does not match specification used to create file. Key location values do not match between Format 2 label and DTF Key location value less than 4 with variable file Key specifications not zero after last valid key entry Key flag values do not match between Format 2 label and DTF. Key size does not equal original keysize used to create file.
	06	Nonindexed output intended to an indexed file Indexed access intended to a nonindexed file
	07	No work area or I/O register specification I/O register specified incorrectly Work area and I/O register specified together

Table B–1A. Data Management Error Message Subcodes (Part 2 of 2)

Associated Data Management Error Message	Message Type Subcode*	Explanation
DM61 (cont)	08	Double buffering with update or random mode
		Double buffering with input and add
		Double buffering with indexed input
	09	Variable build register specified incorrectly
	10	Forward direction not specified with output file
	11	STD labels not specified with ASCII file
		When specifying ASCII, BLKSIZE not greater than 9999
	12	BKNO=YES not specified with block numbered tape
	13	The reader does not have the 96-column read feature
	14	Block size and overflow percentage are too large for disk with low number of tracks/cylinders (8415)
	15	Format other than fixed unblocked or variable unblocked
	16	I/O area 2 not specified with combined file
		Extend not allowed with combined file
		Multisector I/O invalid with combined file
	17	Block size or record size equals zero
	18	An address in the DTF is not within the bounds of the user program
	19	Invalid DTF, CR type not appropriate when Format 2 label active in multivolume
	20	User specified seek address is not word aligned or I/O area's are not half-word aligned
	21	With 7 track and convert on, block size not multiple of 3
	22	Error while performing recovery of IRAM/MIRAM file
	23	Invalid specification in //DD job control statement

*When error condition occurs, the related subcode (in hexadecimal) is placed in byte 44 of the DTF file table.

B.3.2. Disk Space Management Error Codes

The disk space management routines of the OS/3 supervisor do not generate error messages, but instead load a hexadecimal error code into general register 0 for the error or exception conditions listed in Table B–2. The first column of this table contains the hexadecimal error code, which is loaded by disk space management into register 0, byte 3. This is followed by an interpretation of the error or exception condition and suggestions for recovery action.

Table B—2. OS/3 Disk Space Management Error Codes

Internal Hexadecimal Code	Interpretation	Suggested Recovery Action
30	Unrecoverable hardware I/O error on WRITE command; VTOC may be disturbed.	List and examine VTOC, using OBTAIN macro. Attempt to copy all files to another disc; then reprep the suspected defective disc.
31	Unrecoverable hardware I/O error on READ command; VTOC is disturbed.	
32	Unrecoverable hardware I/O error on READ command; VTOC not disturbed.	Same as error code 31
33	Indicated device (PUB) either not allocated or nonexistent.	Check the VOL job control statement and the volume sequence number of the disc volume.
34	File ID error:	
	■ For EXTEND, SCRTCH, RENAME, OBTAIN: the format 1 label record cannot be found on specified volume.	List VTOC and check all format 1 labels. Check also all parameters in the job control device assignment set.
	■ For ALLOC: a file with the same file ID already exists on this volume.	
35	No empty label records in VTOC.	Eliminate unused files or expand VTOC area.
36	No space available on this volume.	Eliminate unused files or change to a noncontiguous request.
37	No file control block (FCB) found for this internal filename (LFD-name).	Check LFD job control statement.
38	For OBTAIN, the disk address specified is invalid.	Provide correct disk address and rerun.
	For track aligned files, SCRTCH is invalid.	Use release that recognizes track aligned files.
39	'Y' is specified as first three characters of file ID to SCRTCH macro (PREFIX function).	System files may not be deallocated with SCRTCH macro.
	$VTOC is named as file to be scratched.	A $VTOC file cannot be scratched.
	For RENAME, the file to be renamed is not a format label file.	Data set label diskettes cannot be scratched.
3A	VTOC format error is detected.	For diskette, check for duplicate or overlapping space or a duplicate name. For disk, list VTOC and check format 4 label.
3B	Request for extension of file will exceed number of allowable extents (16 for all but split files, which are allowed 13).	Create a new file with a single extent large enough to accommodate the contents of the old file.
3C	Error detected in your parameter table.	Review formats presented in this manual and in supervisor user guide, UP-8075 (current version).

B.3.3. Disk File Extension Error Handling

Three types of extend failures can occur, each associated with a data management error diagnostic:

DM45 EXTENT TABLE EXHAUSTED

No space exists in the logical extent table for additional space acquired by file extension.

DM41 FILE SPACE EXHAUSTED

No physical space exists for file extension.

DM11 SPECIFIED NON-EXTENDABLE

DTF specifies the file as nonextendable by:

— maintaining a secondary allocation increment of zero (via the EXT card)

— defining the secondary allocation percent (UOS) as zero

— setting the nonextendable flag in the partition table flag byte.

Errors occuring during file extend operations are always associated with inability to acquire output space for a buffer and consequent loss of output data. On extend failure errors, file extend procedures now minimize loss of output data to one record.

B.4. ERROR FLAGGING PROCEDURES

All OS/3 data management pgorams set bits in a special field of the DTF file table to serve as error flags, providing you with particular information on the error. Disk and tape programs set the bits in this field and then call the logging transient (B.3); card and printer modules go directly to the logging transient. When an error is detected during the execution of a data management transient routine, the logging transient is called after the setting of the error flag bits is completed or bypassed.

B.4.1. FilenameC

The error flag field of the DTF file table is designated *filenameC;* it may be accessed by your program through the *test under mask* instruction (TM), using for operand 1 your 7-byte logical filename, to which you have concatenated the letter C. Note that the size of *filenameC* varies with the type of file: for card and printer files, it is only one byte long; for tape and disk files, it is four bytes long. Table B–3 lists the significance of the bits that are set to binary 1 in *filenameC* for certain error and exception conditions. For information on paper tape error processing, refer to 17.5.9.

Table B—3. Significance of Bits in filenameC (Part 1 of 4)

BYTE 0					
Bit	**DTFMI DTFIR DTFIS**	**DTFSD DTFDA DTFNI**	**DTFMT**	**DTFPR**	**DTFCD**
0	Last block on track accessed	→	Reserved	Line truncated (data too large)	Record size invalid (too large or too small)
1	Invalid ID	→	Reserved	Invalid control character	Reserved
2	Invalid DTF	Invalid PCA/DTF	Invalid DTF	Character mis-match	Validity check (unique unit error)
3	Hardware error	→	→	→	→
4	Error found in OPEN	→	→	→	→
5	Error found in CLOSE	→	→	→	→
6	Invalid macro sequence	→	→	→	→
7	Reserved	(DTFSD: reserved) WAITF required	Reserved	Record size invalid (too small)	Reserved

Table B—3. Significance of Bits in filenameC (Part 2 of 4)

BYTE 1			
Bit	DTFMI DTFIR DTFIS	DTFSD DTFDA DTFNI	DTFMT
0	I/O completed	→	→
1	Unrecoverable error	→	→
2	Unique unit error	→	→
3	Record not found	→	Reserved
4	Unit exception	→	→
5	Wrong length found	→	Reserved
6	End of track	→	Reserved
7	End of cylinder	→	Reserved

Table B—3. Significance of Bits in filenameC (Part 3 of 4)

BYTE 2			
Bit	DTFMI DTFIR DTFIS	DTFSD DTFDA DTFNI	DTFMT
0	Command rejection	→	→
1	Intervention required	→	→
2	Output parity check	→	→
3	Equipment check	→	→
4	Data check	→	→
5	Overrun	→	→
6	STOP state	→	Word count = zero
7	Device check	→	Data converter check (7-track only)

Table B—3. Significance of Bits in filenameC (Part 4 of 4)

BYTE 3			
Bit	**DTFMI DTFIR DTFIS**	**DTFSD DTFDA DTFNI**	**DTFMT**
0	Invalid record size	→	→
1	Logical end of file	→	→
2	File space exhausted (DTFIS) Logical end of volume (DTFIR and DTFMI)	Logical end of volume	→
3	Processing inhibited	Invalid subfile	Wrong length error
4	Invalid index	Reserved	Reserved
5	Sequence error (DTFIS and DTFIR only)	Reserved	Reserved
6	Duplicate record	Reserved	Reserved
7	ADD rejected (DTFIS only)	Reserved	Reserved

B.4.2. Other DTF Fields

Certain of the OS/3 data management modules place, in other fields of the DTF file table than *filenameC*, additional information that is of value to you in monitoring the processing of your files. The details are documented for each specific use in the appropriate section of this manual; these fields are designated *filenameA*, *filenameP*, *filenameS*, etc., and are addressed in the same manner as *filenameC*: by concatenating the letter designation to your 7-byte logical filename.

Appendix C. Code Correspondences

C.1. GENERAL

This appendix presents a cross-reference table and figures useful to you for visualizing the correspondences among the following codes commonly used in data processing and in OS/3:

- Hollerith punched card code
- EBCDIC (Extended Binary Coded Decimal Interchange Code)
- ASCII (American National Standard Code for Information Interchange)
- Binary bit-pattern (bit-configuration) representation for an 8-bit system.
- Hexadecimal representation
- Compressed code for punched cards
- Binary (image) mode for punched cards

C.2. EBCDIC/ASCII/HOLLERITH CORRESPONDENCE

Table C—1 is a cross-reference table depicting the correspondences among the Hollerith punched card code, ASCII, and EBCDIC. The table is arranged in the sorting (or collating) sequence of the binary bit-patterns that have been assigned to the codes, with 0000 0000 being the lowest value in the sequence and 1111 1111 the highest.

Note that the column headed *Decimal* uses decimal numbers to represent the positions of the codes and bit patterns in this sequence, but counts the position of the lowest value as the 0th (zeroth) position rather than the first. Thus, the position of the highest value bit-pattern 1111 1111 is represented in the decimal column by 255, whereas it is actually the 256th in the sequence. This scheme corresponds to the common convention for numbering bytes, in which the first byte of a group is byte 0, and is convenient when you are constructing a 256-byte translation table. (See the MODE keyword parameter of the DTFCD declarative macroinstruction (3.3).)

The column headed *Decimal* also represents the collating sequence for the EBCDIC graphic characters shown in the fourth column of the table; the fifth column, *Hollerith Punched Card Code,* contains the hole patterns assigned to these EBCDIC graphics. Empty space in the fourth column represents the positions of the EBCDIC control characters; the EBCDIC space charater is represented in the fourth column by the conventional notation SP at decimal position 64, and the corresponding card code is "no punches."

The ASCII graphic characters, listed in the sixth column of Table C—1, are also in their collating sequence, and the hole patterns in the seventh column correspond to the ASCII graphics. The ASCII space character is represented by the notation SP in the sixth column at decimal position 32; the corresponding card code is, again, "no punches." The empty space in the sixth column represents the positions of the ASCII control characters. The shading in the ASCII graphic character column indicates where the 128-character ASCII code leaves off: there are no ASCII graphic or control characters that correspond to the bit patterns higher in collating sequence than 0111 1111 (the 128th in Table C—1).

C.2.1. Hollerith Punched Card Code

The standard Hollerith punched card code specifies 256 hole-patterns in 12-row punched cards. Hole-patterns are assigned to the 128 characters of ASCII and to 128 additional characters for use in 8-bit coded systems. These include the EBCDIC set. Note that no sorting sequence is implied by the Hollerith code itself.

C.2.2. EBCDIC

EBCDIC is an extension of Hollerith coding practices. It comprises 256 characters, each of which is represented by an 8-bit pattern. Table C—1 shows the EBCDIC graphic characters only; the EBCDIC control characters are not indicated.

C.2.3. ASCII

ASCII comprises 128 coded characters, each represented by an 8-bit pattern, and includes both control characters and graphic characters. Only the latter are shown in Table C—1. ASCII is used for information interchange among data processing communication systems and associated equipment.

Table C—1. Cross-Reference Table: EBCDIC/ASCII/Hollerith (Part 1 of 5)

EBCDIC					ASCII	
Decimal	Hexa-deci-mal	Binary	EBCDIC Graphic Character	Hollerith Punched Card Code	ASCII Graphic Character	Hollerith Punched Card Code
0	00	0000 0000		12-0-9-8-1		12-0-9-8-1
1	01	0000 0001		12-9-1		12-9-1
2	02	0000 0010		12-9-2		12-9-2
3	03	0000 0011		12-9-3		12-9-3
4	04	0000 0100		12-9-4		9-7
5	05	0000 0101		12-9-5		0-9-8-5
6	06	0000 0110		12-9-6		0-9-8-6
7	07	0000 0111		12-9-7		0-9-8-7
8	08	0000 1000		12-9-8		11-9-6
9	09	0000 1001		12-9-8-1		12-9-5
10	0A	0000 1010		12-9-8-2		0-9-5
11	0B	0000 1011		12-9-8-3		12-9-8-3
12	0C	0000 1100		12-9-8-4		12-9-8-4
13	0D	0000 1101		12-9-8-5		12-9-8-5
14	0E	0000 1110		12-9-8-6		12-9-8-6
15	0F	0000 1111		12-9-8-7		12-9-8-7
16	10	0001 0000		12-11-9-8-1		12-11-9-8-1
17	11	0001 0001		11-9-1		11-9-1
18	12	0001 0010		11-9-2		11-9-2
19	13	0001 0011		11-9-3		11-9-3
20	14	0001 0100		11-9-4		9-8-4
21	15	0001 0101		11-9-5		9-8-5
22	16	0001 0110		11-9-6		9-2
23	17	0001 0111		11-9-7		0-9-6
24	18	0001 1000		11-9-8		11-9-8
25	19	0001 1001		11-9-8-1		11-9-8-1
26	1A	0001 1010		11-9-8-2		9-8-7
27	1B	0001 1011		11-9-8-3		0-9-7
28	1C	0001 1100		11-9-8-4		11-9-8-4
29	1D	0001 1101		11-9-8-5		11-9-8-5
30	1E	0001 1110		11-9-8-6		11-9-8-6
31	1F	0001 1111		11-9-8-7		11-9-8-7
32	20	0010 0000		11-0-9-8-1	SP	No punches
33	21	0010 0001		0-9-1	!	12-8-7
34	22	0010 0010		0-9-2	"	8-7
35	23	0010 0011		0-9-3	#	8-3
36	24	0010 0100		0-9-4	$	11-8-3
37	25	0010 0101		0-9-5	%	0-8-4
38	26	0010 0110		0-9-6	&	12
39	27	0010 0111		0-9-7	'	8-5
40	28	0010 1000		0-9-8	(	12-8-5
41	29	0010 1001		0-9-8-1	)	11-8-5
42	2A	0010 1010		0-9-8-2	*	11-8-4
43	2B	0010 1011		0-9-8-3	+	12-8-6
44	2C	0010 1100		0-9-8-4	,	0-8-3
45	2D	0010 1101		0-9-8-5	—	11
46	2E	0010 1110		0-9-8-6	.	12-8-3
47	2F	0010 1111		0-9-8-7	/	0-1
48	30	0011 0000		12-11-0-9-8-1	0	0
49	31	0011 0001		9-1	1	1
50	32	0011 0010		9-2	2	2
51	33	0011 0011		9-3	3	3
52	34	0011 0100		9-4	4	4
53	35	0011 0101		9-5	5	5
54	36	0011 0110		9-6	6	6

Table C—1. Cross-Reference Table: EBCDIC/ASCII/Hollerith (Part 2 of 5)

EBCDIC					ASCII	
Decimal	Hexa-deci-mal	Binary	EBCDIC Graphic Character	Hollerith Punched Card Code	ASCII Graphic Character	Hollerith Punched Card Code
55	37	0011 0111		9-7	7	7
56	38	0011 1000		9-8	8	8
57	39	0011 1001		9-8-1	9	9
58	3A	0011 1010		9-8-2	:	8-2
59	3B	0011 1011		9-8-3	;	11-8-6
60	3C	0011 1100		9-8-4	<	12-8-4
61	3D	0011 1101		9-8-5	=	8-6
62	3E	0011 1110		9-8-6	>	0-8-6
63	3F	0011 1111		9-8-7	?	0-8-7
64	40	0100 0000	SP	No punches	@	8-4
65	41	0100 0001		12-0-9-1	A	12-1
66	42	0100 0010		12-0-9-2	B	12-2
67	43	0100 0011		12-0-9-3	C	12-3
68	44	0100 0100		12-0-9-4	D	12-4
69	45	0100 0101		12-0-9-5	E	12-5
70	46	0100 0110		12-0-9-6	F	12-6
71	47	0100 0111		12-0-9-7	G	12-7
72	48	0100 1000		12-0-9-8	H	12-8
73	49	0100 1001		12-8-1	I	12-9
74	4A	0100 1010	[	12-8-2	J	11-1
75	4B	0100 1011	.	12-8-3	K	11-2
76	4C	0100 1100	<	12-8-4	L	11-3
77	4D	0100 1101	(	12-8-5	M	11-4
78	4E	0100 1110	+	12-8-6	N	11-5
79	4F	0100 1111	!	12-8-7	O	11-6
80	50	0101 0000	&	12	P	11-7
81	51	0101 0001		12-11-9-1	Q	11-8
82	52	0101 0010		12-11-9-2	R	11-9
83	53	0101 0011		12-11-9-3	S	0-2
84	54	0101 0100		12-11-9-4	T	0-3
85	55	0101 0101		12-11-9-5	U	0-4
86	56	0101 0110		12-11-9-6	V	0-5
87	57	0101 0111		12-11-9-7	W	0-6
88	58	0101 1000		12-11-9-8	X	0-7
89	59	0101 1001		11-8-1	Y	0-8
90	5A	0101 1010	]	11-8-2	Z	0-9
91	5B	0101 1011	$	11-8-3	[	12-8-2
92	5C	0101 1100	*	11-8-4	\	0-8-2
93	5D	0101 1101	)	11-8-5	]	11-8-2
94	5E	0101 1110	;	11-8-6	^	11-8-7
95	5F	0101 1111	^	11-8-7	_	0-8-5
96	60	0110 0000	—	11	`	8-1
97	61	0110 0001	/	0-1	a	12-0-1
98	62	0110 0010		11-0-9-2	b	12-0-2
99	63	0110 0011		11-0-9-3	c	12-0-3
100	64	0110 0100		11-0-9-4	d	12-0-4
101	65	0110 0101		11-0-9-5	e	12-0-5
102	66	0110 0110		11-0-9-6	f	12-0-6
103	67	0110 0111		11-0-9-7	g	12-0-7
104	68	0110 1000		11-0-9-8	h	12-0-8
105	69	0110 1001		0-8-1	i	12-0-9
106	6A	0110 1010	¦	12-11	j	12-11-1
107	6B	0110 1011	,	0-8-3	k	12-11-2
108	6C	0110 1100	%	0-8-4	l	12-11-3
109	6D	0110 1101	—	0-8-5	m	12-11-4

Table C—1. Cross-Reference Table: EBCDIC/ASCII/Hollerith (Part 3 of 5)

EBCDIC					ASCII	
Decimal	Hexa-deci-mal	Binary	EBCDIC Graphic Character	Hollerith Punched Card Code	ASCII Graphic Character	Hollerith Punched Card Code
110	6E	0110 1110	>	0-8-6	n	12-11-5
111	6F	0110 1111	?	0-8-7	o	12-11-6
112	70	0111 0000		12-11-0	p	12-11-7
113	71	0111 0001		12-11-0-9-1	q	12-11-8
114	72	0111 0010		12-11-0-9-2	r	12-11-9
115	73	0111 0011		12-11-0-9-3	s	11-0-2
116	74	0111 0100		12-11-0-9-4	t	11-0-3
117	75	0111 0101		12-11-0-9-5	u	11-0-4
118	76	0111 0110		12-11-0-9-6	v	11-0-5
119	77	0111 0111		12-11-0-9-7	w	11-0-6
120	78	0111 1000		12-11-0-9-8	x	11-0-7
121	79	0111 1001	`	8-1	y	11-0-8
122	7A	0111 1010	:	8-2	z	11-0-9
123	7B	0111 1011	#	8-3	{	12-0
124	7C	0111 1100	@	8-4	|	12-11
125	7D	0111 1101	'	8-5	}	11-0
126	7E	0111 1110	=	8-6	~	11-0-1
127	7F	0111 1111	"	8-7		12-9-7
128	80	1000 0000		12-0-8-1		11-0-9-8-1
129	81	1000 0001	a	12-0-1		0-9-1
130	82	1000 0010	b	12-0-2		0-9-2
131	83	1000 0011	c	12-0-3		0-9-3
132	84	1000 0100	d	12-0-4		0-9-4
133	85	1000 0101	e	12-0-5		11-9-5
134	86	1000 0110	f	12-0-6		12-9-6
135	87	1000 0111	g	12-0-7		11-9-7
136	88	1000 1000	h	12-0-8		0-9-8
137	89	1000 1001	i	12-0-9		0-9-8-1
138	8A	1000 1010		12-0-8-2		0-9-8-2
139	8B	1000 1011		12-0-8-3		0-9-8-3
140	8C	1000 1100		12-0-8-4		0-9-8-4
141	8D	1000 1101		12-0-8-5		12-9-8-1
142	8E	1000 1110		12-0-8-6		12-9-8-2
143	8F	1000 1111		12-0-8-7		11-9-8-3
144	90	1001 0000		12-11-8-1		12-11-0-9-8-1
145	91	1001 0001	j	12-11-1		9-1
146	92	1001 0010	k	12-11-2		11-9-8-2
147	93	1001 0011	l	12-11-3		9-3
148	94	1001 0100	m	12-11-4		9-4
149	95	1001 0101	n	12-11-5		9-5
150	96	1001 0110	o	12-11-6		9-6
151	97	1001 0111	p	12-11-7		12-9-8
152	98	1001 1000	q	12-11-8		9-8
153	99	1001 1001	r	12-11-9		9-8-1
154	9A	1001 1010		12-11-8-2		9-8-2
155	9B	1001 1011		12-11-8-3		9-8-3
156	9C	1001 1100		12-11-8-4		12-9-4
157	9D	1001 1101		12-11-8-5		11-9-4
158	9E	1001 1110		12-11-8-6		9-8-6
159	9F	1001 1111		12-11-8-7		11-0-9-1

Table C—1. Cross-Reference Table: EBCDIC/ASCII/Hollerith (Part 4 of 5)

EBCDIC					ASCII	
Decimal	Hexa-deci-mal	Binary	EBCDIC Graphic Character	Hollerith Punched Card Code	ASCII Graphic Character	Hollerith Punched Card Code
160	A0	1010 0000		11-0-8-1		12-0-9-1
161	A1	1010 0001	~	11-0-1		12-0-9-2
162	A2	1010 0010	s	11-0-2		12-0-9-3
163	A3	1010 0011	t	11-0-3		12-0-9-4
164	A4	1010 0100	u	11-0-4		12-0-9-5
165	A5	1010 0101	v	11-0-5		12-0-9-6
166	A6	1010 0110	w	11-0-6		12-0-9-7
167	A7	1010 0111	x	11-0-7		12-0-9-8
168	A8	1010 1000	y	11-0-8		12-8-1
169	A9	1010 1001	z	11-0-9		12-11-9-1
170	AA	1010 1010		11-0-8-2		12-11-9-2
171	AB	1010 1011		11-0-8-3		12-11-9-3
172	AC	1010 1100		11-0-8-4		12-11-9-4
173	AD	1010 1101		11-0-8-5		12-11-9-5
174	AE	1010 1110		11-0-8-6		12-11-9-6
175	AF	1010 1111		11-0-8-7		12-11-9-7
176	B0	1011 0000		12-11-0-8-1		12-11-9-8
177	B1	1011 0001		12-11-0-1		11-8-1
178	B2	1011 0010		12-11-0-2		11-0-9-2
179	B3	1011 0011		12-11-0-3		11-0-9-3
180	B4	1011 0100		12-11-0-4		11-0-9-4
181	B5	1011 0101		12-11-0-5		11-0-9-5
182	B6	1011 0110		12-11-0-6		11-0-9-6
183	B7	1011 0111		12-11-0-7		11-0-9-7
184	B8	1011 1000		12-11-0-8		11-0-9-8
185	B9	1011 1001		12-11-0-9		0-8-1
186	BA	1011 1010		12-11-0-8-2		12-11-0
187	BB	1011 1011		12-11-0-8-3		12-11-0-9-1
188	BC	1011 1100		12-11-0-8-4		12-11-0-9-2
189	BD	1011 1101		12-11-0-8-5		12-11-0-9-3
190	BE	1011 1110		12-11-0-8-6		12-11-0-9-4
191	BF	1011 1111		12-11-0-8-7		12-11-0-9-5
192	C0	1100 0000	{	12-0		12-11-0-9-6
193	C1	1100 0001	A	12-1		12-11-0-9-7
194	C2	1100 0010	B	12-2		12-11-0-9-8
195	C3	1100 0011	C	12-3		12-0-8-1
196	C4	1100 0100	D	12-4		12-0-8-2
197	C5	1100 0101	E	12-5		12-0-8-3
198	C6	1100 0110	F	12-6		12-0-8-4
199	C7	1100 0111	G	12-7		12-0-8-5
200	C8	1100 1000	H	12-8		12-0-8-6
201	C9	1100 1001	I	12-9		12-0-8-7
202	CA	1100 1010		12-0-9-8-2		12-11-8-1
203	CB	1100 1011		12-0-9-8-3		12-11-8-2
204	CC	1100 1100		12-0-9-8-4		12-11-8-3
205	CD	1100 1101		12-0-9-8-5		12-11-8-4
206	CE	1100 1110		12-0-9-8-6		12-11-8-5
207	CF	1100 1111		12-0-9-8-7		12-11-8-6
208	D0	1101 0000	}	11-0		12-11-8-7
209	D1	1101 0001	J	11-1		11-0-8-1

Table C—1. Cross-Reference Table: EBCDIC/ASCII/Hollerith (Part 5 of 5)

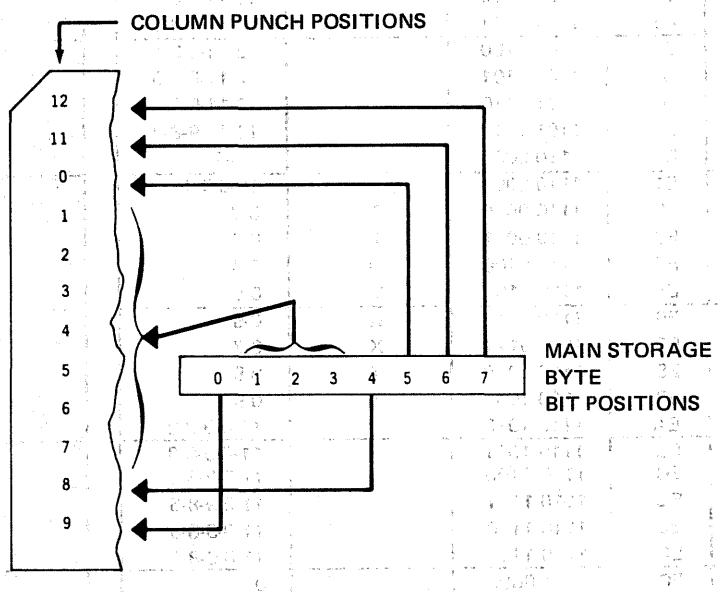
EBCDIC					ASCII	
Decimal	Hexa-deci-mal	Binary	EBCDIC Graphic Character	Hollerith Punched Card Code	ASCII Graphic Character	Hollerith Punched Card Code
210	D2	1101 0010	K	11-2		11-0-8-2
211	D3	1101 0011	L	11-3		11-0-8-3
212	D4	1101 0100	M	11-4		11-0-8-4
213	D5	1101 0101	N	11-5		11-0-8-5
214	D6	1101 0110	O	11-6		11-0-8-6
215	D7	1101 0111	P	11-7		11-0-8-7
216	D8	1101 1000	Q	11-8		12-11-0-8-1
217	D9	1101 1001	R	11-9		12-11-0-1
218	DA	1101 1010		12-11-9-8-2		12-11-0-2
219	DB	1101 1011		12-11-9-8-3		12-11-0-3
220	DC	1101 1100		12-11-9-8-4		12-11-0-4
221	DD	1101 1101		12-11-9-8-5		12-11-0-5
222	DE	1101 1110		12-11-9-8-6		12-11-0-6
223	DF	1101 1111		12-11-9-8-7		12-11-0-7
224	E0	1110 0000	\	0-8-2		12-11-0-8
225	E1	1110 0001		11-0-9-1		12-11-0-9
226	E2	1110 0010	S	0-2		12-11-0-8-2
227	E3	1110 0011	T	0-3		12-11-0-8-3
228	E4	1110 0100	U	0-4		12-11-0-8-4
229	E5	1110 0101	V	0-5		12-11-0-8-5
230	E6	1110 0110	W	0-6		12-11-0-8-6
231	E7	1110 0111	X	0-7		12-11-0-8-7
232	E8	1110 1000	Y	0-8		12-0-9-8-2
233	E9	1110 1001	Z	0-9		12-0-9-8-3
234	EA	1110 1010		11-0-9-8-2		12-0-9-8-4
235	EB	1110 1011		11-0-9-8-3		12-0-9-8-5
236	EC	1110 1100		11-0-9-8-4		12-0-9-8-6
237	ED	1110 1101		11-0-9-8-5		12-0-9-8-7
238	EE	1110 1110		11-0-9-8-6		12-11-9-8-2
239	EF	1110 1111		11-0-9-8-7		12-11-9-8-3
240	F0	1111 0000	0	0		12-11-9-8-4
241	F1	1111 0001	1	1		12-11-9-8-5
242	F2	1111 0010	2	2		12-11-9-8-6
243	F3	1111 0011	3	3		12-11-9-8-7
244	F4	1111 0100	4	4		11-0-9-8-2
245	F5	1111 0101	5	5		11-0-9-8-3
246	F6	1111 0110	6	6		11-0-9-8-4
247	F7	1111 0111	7	7		11-0-9-8-5
248	F8	1111 1000	8	8		11-0-9-8-6
249	F9	1111 1001	9	9		11-0-9-8-7
250	FA	1111 1010		12-11-0-9-8-2		12-11-0-9-8-2
251	FB	1111 1011		12-11-0-9-8-3		12-11-0-9-8-3
252	FC	1111 1100		12-11-0-9-8-4		12-11-0-9-8-4
253	FD	1111 1101		12-11-0-9-8-5		12-11-0-9-8-5
254	FE	1111 1110		12-11-0-9-8-6		12-11-0-9-8-6
255	FF	1111 1111		12-11-0-9-8-7		12-11-0-9-8-7

C.3. OTHER CARD CODES

Two other punched card coding systems can be handled with OS/3 data management and all card reader and card punch subsystems in the SPERRY UNIVAC 90/30 System: the compressed code and the column binary, or image, code.

C.3.1. Compressed Card Code

Figure C—1 indicates the construction of the compressed card code; each card column is represented by an 8-bit pattern in one byte of main storage.



NOTE:

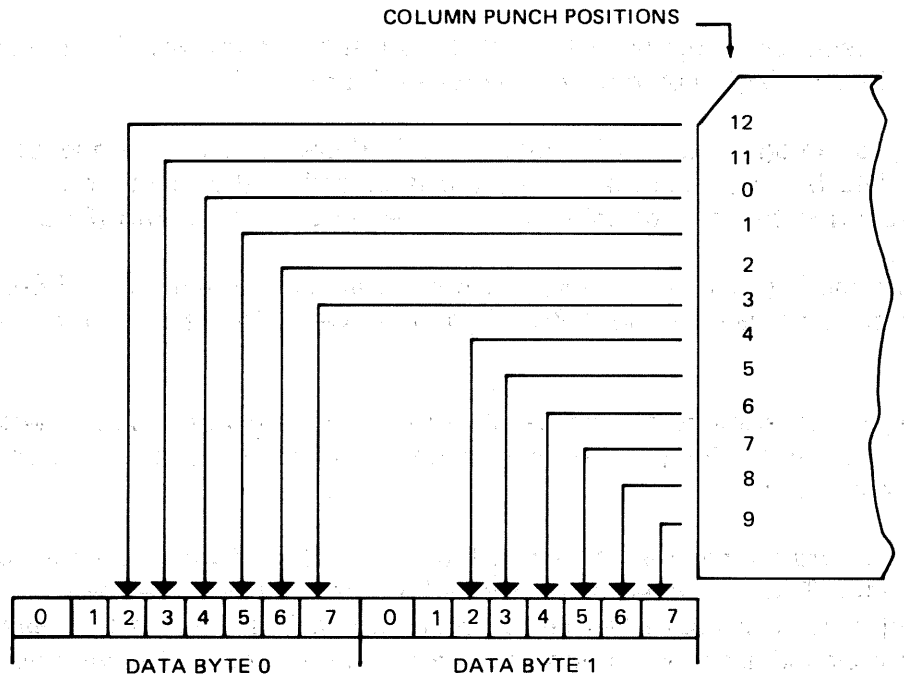
PUNCH POSITIONS 1 THROUGH 7 ARE INDICATED IN BITS 1 THROUGH 3, ACCORDING TO THE FOLLOWING TABLE:

PUNCH ROWS 1 THRU 7	BITS 123
NONE	000
1	011
2	101
3	001
4	010
5	100
6	111
7	110

Figure C—1. Compressed Card Code

C.3.2. Column Binary (Image) Code

Figure C—2 indicates the construction of this code. Note that each card column requires two bytes of main storage; an I/O area of 160 bytes is required for an 80-column card.



NOTE:

BITS 0 AND 1 ARE CLEARED TO ZEROS ON AN IMAGE READ.

Figure C—2. Column Binary (Image) Card Code

C.4. DATA CONVERSION

In OS/3 data management, there are five ways in which your data, held in main storage in 8-bit bytes, may be converted into hole-patterns in punched cards, and vice versa:

- Standard mode (EBCDIC)
- Standard mode (ASCII)
- Compressed code mode
- Binary (image) mode
- Translate mode for reading or punching

In EBCDIC standard mode (MODE=STD), data in main storage in EBCDIC code is punched into cards in the Hollerith punched card code. Cards are read in Hollerith, and the data stored in EBCDIC.

In ASCII standard mode (MODE=STD and ASCII=YES), data management translates data stored in the 8-bit ASCII code in main storage to EBCDIC, and then the cards are punched in Hollerith. The reverse process is used when cards are read, unless a hardware ASCII translate feature is available on the card reader, when data management omits the EBCDIC-to-ASCII translation.

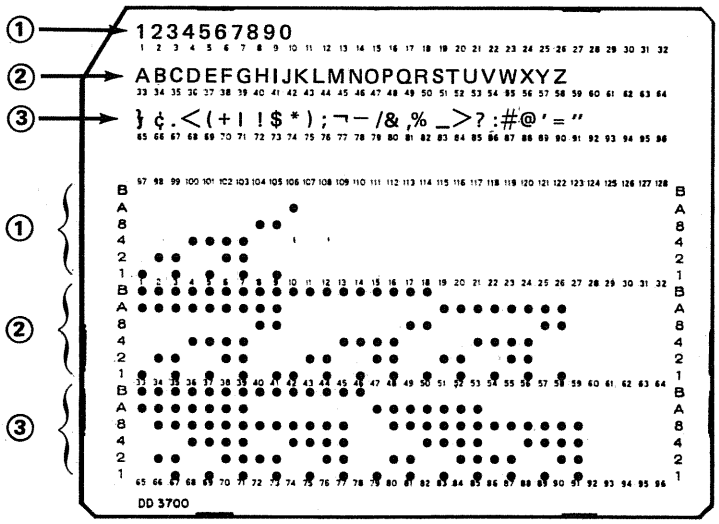
In the compressed code mode (MODE=CC), an 8-bit data byte is converted by data management into a single-column hole-pattern (Figure C—1).

In the binary or image mode (MODE=BINARY), there is a one-to-one correspondence between 12 data bytes in main storage (data is stored in the least significant six bits of two 8-bit bytes) and the 12 possible row punches in a card column (Figure C—2).

In the translate mode (MODE=TRANS), you make your own assignment of 8-bit patterns to the 256 hole-patterns listed under EBCDIC in Table C—1, in the order these are shown in the table.

The preceding considerations should be of little concern to you because, with OS/3 data management, you can always use any mode with any peripheral equipment in your installation's configuration.

The 96-column card format (as shown in Figure C—3) hold 96 characters of data at six bits per character. The characters are arranged on the card as three rows of 32 characters each. The fact that each character can be represented in six bits is the reason no binary read mode is provided. Depending upon the translate features in the hardware, and MODE keyword specification, each 6-hole character on a card is transferred into the user's I/O area as an EBCDIC or ASCII 8-bit byte.



① **NUMERIC CHARACTERS**

		1	2	3	4	5	6	7	8	9	0
Zone Punches	B										
	A										A
	8								8	8	
Digit Punches	4				4	4	4	4			
	2		2	2			2	2			
	1	1		1		1		1		1	

② **ALPHABETIC CHARACTERS**

		A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Zone Punches	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B								
	A	A	A	A	A	A	A	A	A	A										A	A	A	A	A	A	A	A
	8								8	8								8	8							8	8
Digit Punches	4				4	4	4	4						4	4	4	4					4	4	4	4		
	2		2	2			2	2				2	2			2	2			2	2			2	2		
	1	1		1		1		1		1	1		1		1		1		1		1		1		1		1

③ **SPECIAL CHARACTERS**

		}	¢	.	<	(	+	|	!	$	*	)	;	¬	–	/	&	,	%	_	>	?	:	#	@	'	=	"	(blank) Δ
Zone Punches	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B														
	A	A	A	A	A	A	A	A								A	A	A	A	A	A	A							
	8		8	8	8	8	8	8	8	8	8	8	8	8			8	8	8	8	8	8	8	8	8	8	8	8	
Digit Punches	4				4	4	4	4			4	4	4	4					4	4	4	4			4		4	4	
	2		2	2			2	2	2	2			2	2			2	2			2	2	2	2			2	2	
	1			1		1		1		1		1		1		1		1		1		1		1		1		1	

Figure C—3. 96-Column Card Punch Codes

Appendix D. Labels for Disk Files

D.1. GENERAL

This appendix describes the system standard labels for disk files in OS/3 as well as the optional user standard labels that are supported by OS/3 data management for processing disk files described by the DTFSD, DTFNI, and DTFDA macros. Note that OS/3 ISAM does not support user labels for DTFIS files. (The user standard labels are described in D.4.)

Because your files within a disk volume may be stored in various locations, a directory listing the addresses of the fragments of the files is required. This directory, called the volume table of contents (VTOC), and your files within a disk volume require various standard labels in predefined formats to describe the properties of the files and the volumes on which they reside.

The system standard disk labels include the volume label (VOL1 label) and seven types of format labels. These labels may, according to their use, be separated into two distinct groups:

- Volume Information Group
  - VOL1 label
  - Format 4 label
  - Format 5 label
  - Format 6 label
  - Format 0 label
- File Information Group
  - Format 1 label
  - Format 2 label
  - Format 3 label

The VOL1 label has a length of 84 bytes; all format labels are 140 bytes long.

D.2. VOLUME INFORMATION GROUP

The volume information group, comprising the VOL1 label and the format 4, 5, 6, and 0 labels, identifies the volume and defines the VTOC, the status of the VTOC, the available space within the volume, and the device-dependent characteristics of the volume on which the group resides.

Standard linkages maintained within the group are shown in Figure D—1. The VOL1 label, normally the first label in the group to be referenced, is written at cylinder 0, track 0, record 3 on each volume. The VOL1 label identifies the volume and contains a link to the format 4 label. The format 4 label defines the extent occupied by the VTOC and the device-independent characteristics of the volume; it also supplies a link to the first format 0 label.

Each label in the VTOC that describes label records not in use is defined as a format 0 label and is linked to the next available format 0.

The format 4 label also supplies a link to the first format 6 label, which defines space available within the extent areas of files sharing extents (split cylinder allocation). If more format 6 labels are required, they are linked in the same manner as format 0 labels. The format 6 label and its link from the format 4 label will be present only if split-cylinder allocation has taken place.

The first (or only) format 5 label immediately follows the format 4 label, supplying an implied linkage. The format 5 label defines unused space on the volume in terms of full cylinders. Successive format 5 labels, if required, are linked one to another. The VTOC extent, as specified in the format 4 label, supplies an additional linkage because it is this area that must be searched in order to access the file information groups.

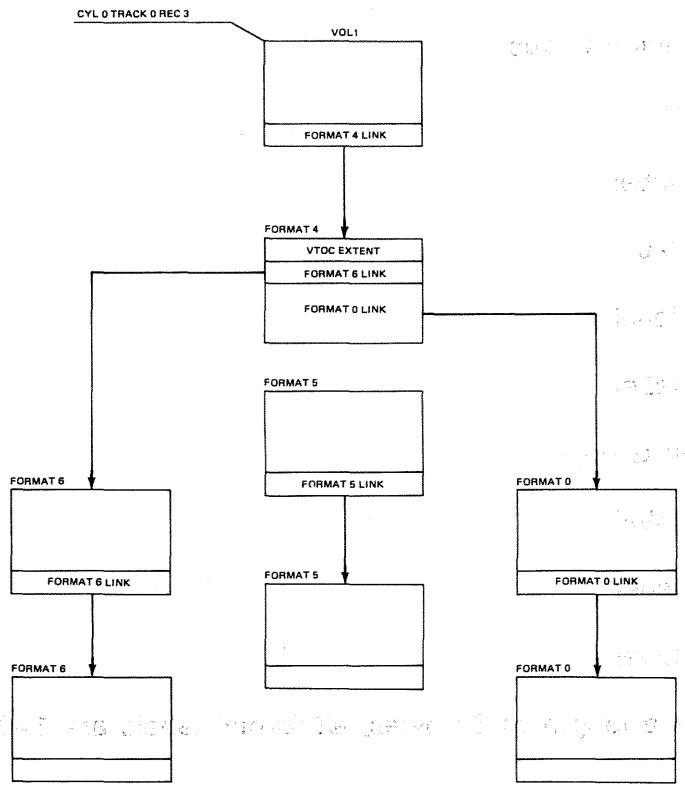


Figure D—1. VTOC Volume Information Label Group

D.2.1. VOL1 Label

As each disk volume enters the system, it is given a unique identification code or volume serial number and the rudiments of a VTOC. The volume serial number and the address of the VTOC are placed in the VOL1 label.

The VOL1 label, identified by a key field and label identification field containing "VOL1", is written by the disk initialization routine at cylinder 0, head 0, record 3.

The VOL1 label is the standard volume label in the OS/3. All reference to the VTOC of a given volume is made by first obtaining the VOL1 label, verifying the volume serial number, and, because the location of the VTOC may vary from volume to volume, using the VTOC address contained in the VOL1 label to locate the VTOC itself.

The format of the VOL1 label is shown in Figure D—2; Table D—1 summarizes its contents.

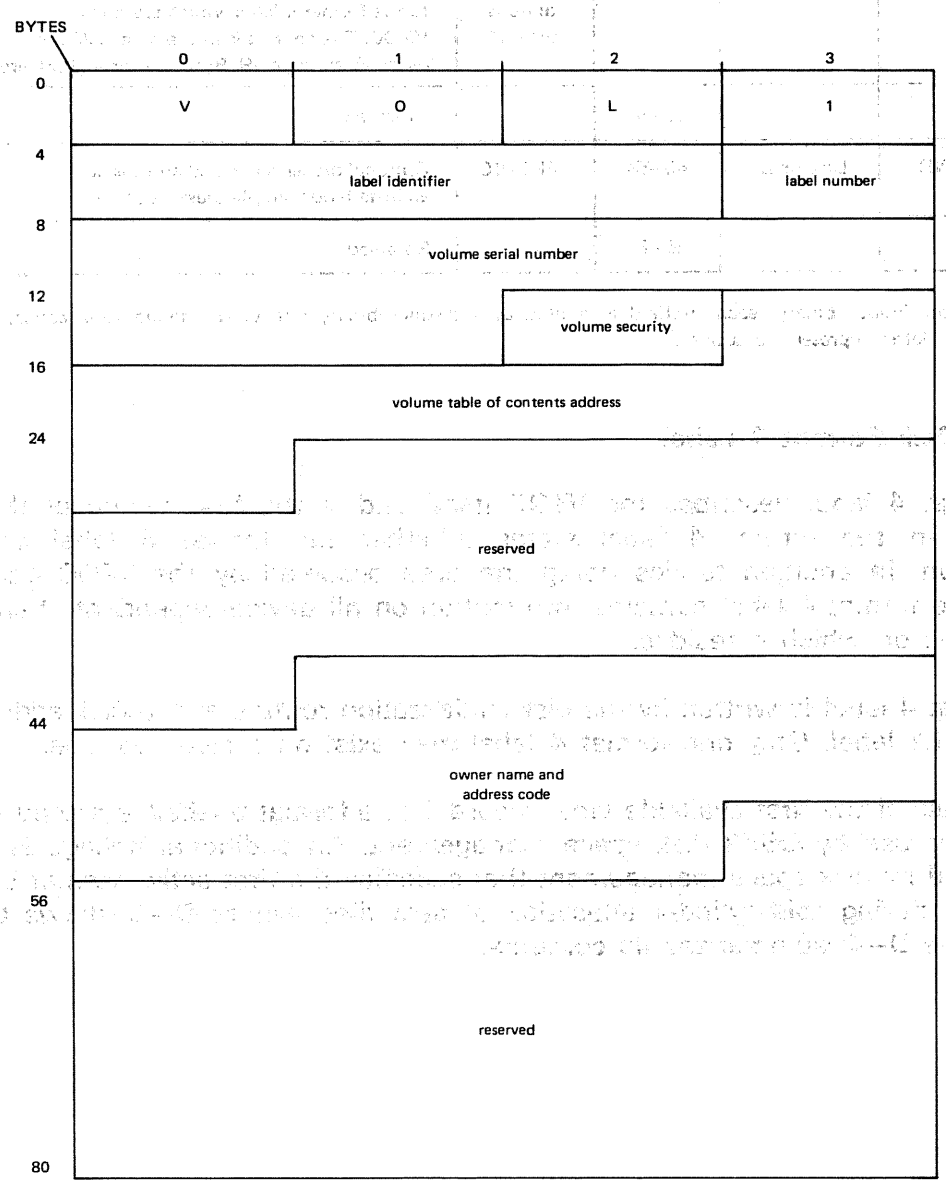


Figure D—2. VTOC VOL1 Label

Table D—1. Contents of VOL1 Label

Label	Initialized by	Bytes	Code	Description
DL$VL	Disc prep	0–3	EBCDIC	Key – contains VOL1.
DL$VL1		4–6		Label identifier – VOL.
		7		Label number – always 1.
DL$VSN		8–13		Volume serial number – a unique code assigned to a disc pack when it enters the system. The same code should appear visually on the disc pack for operator identification.
DL$VSB	Disc prep	14	Binary	Volume security – reserved for future use.
DL$VTC		15–24	Discontinuous binary*	VTOC address – This field is used to point to the format 4 label, which starts the VTOC. The address is in the form *cchhr* in bytes 15 through 19. Bytes 20 through 24 are 0.
		25–44		Reserved
DL$ONR	Disc prep	45–54	EBCDIC	Optional owner name and address code – an installation-supplied user identifier.
		55–83		Reserved

* For discontinuous binary, each subfield is treated as a distinct binary entity. In the notation *cchhr*, each different letter represents a subfield.

D.2.2. Disk Format 4 Label

The format 4 label describes the VTOC itself and is the first record of the VTOC. An indicator in the format 4 label states whether the format 5 label contains valid information. In addition to describing the area occupied by the VTOC and its current status, the format 4 label contains information on all device-dependent characteristics of the volume on which it resides.

The format 4 label is written by the disk initialization routine at the disk address specified in the VOL1 label. Only one format 4 label may exist on a given volume.

The address of the first available label record (i.e., a format 0 label) is placed in the format 4 label for use by OS/3 disk space management. An additional linkage is created and maintained by disk space management that specifies the first active format 6 label and is used only during split-cylinder allocation of data files. Figure D—3 shows the format 4 label; Table D—2 summarizes its contents.

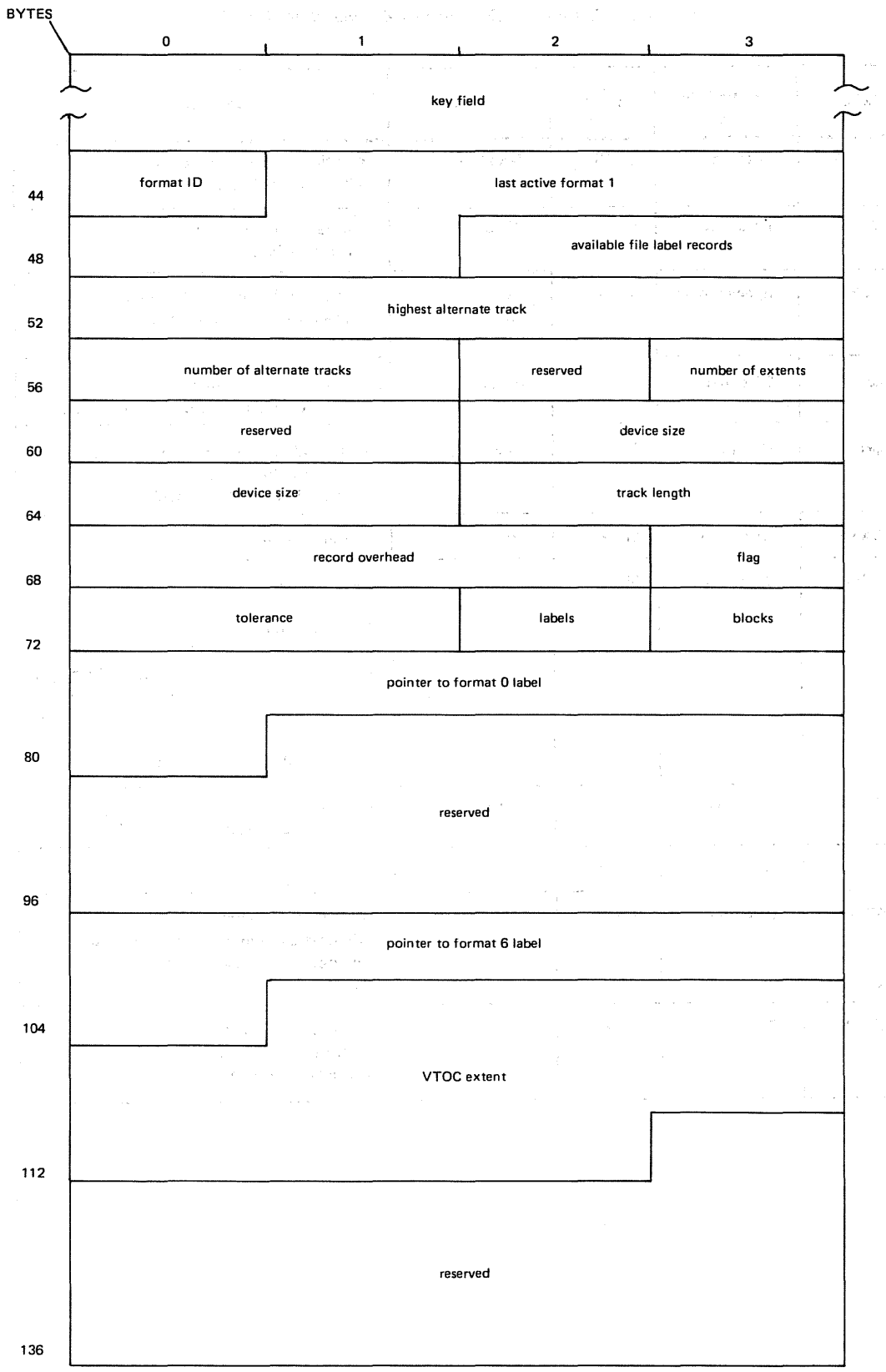


Figure D—3. Disk Format 4 Label

Table D—2. Contents of Disk Format 4 Label (Part 1 of 2)

Label	Initialized by	Bytes	Code	Description
DL$KY4	Disc prep	0–43	Hexadecimal	Key field — Each byte of this field contains 04_{16}.
DL$ID4	Disc prep	44	EBCDIC	Format ID — always 4 for format 4 label.
DL$LF4	Space mgmt	45–49	Discontinuous binary	Last active format 1 — the address, in the form *CCHHR*, used for a search on filename.
DL$AF4	Disc prep	50–51	Binary	Available file label records — number of unused records in the VTOC.
DL$HA4	Disc prep	52–55	Discontinuous binary	Highest alternate track — address, in the form *CCHH*, of alternate tracks set aside in case of bad tracks.
DL$AT4	Disc prep	56–57	Binary	Number of alternate tracks.
DL$VI4	Space mgmt	58		Reserved for VTOC indicators — Bit — Contents — Meaning 0 — 1 — A format 5 label, if present, contains invalid information. 1–7 — 0 — Unused
DL$XC4	Disc prep	59	Binary	Number of extents — contains 01_{16} to indicate the one extent in the VTOC.
	Disc prep	60–61		Reserved
DL$DS4	Disc prep	62–65		Device size — indicates the number of cylinders and the number of heads per cylinder on the device, in the form *CCHH*.
DL$TL4	Disc prep	66–67		Track length — number of available bytes on a track exclusive of home address and record 0.

Table D—2. Contents of Disk Format 4 Label (Part 2 of 2)

Label	Initialized by	Bytes	Code	Description
DL$RO4	Disc prep	68—70		Record overhead — ILK describes overhead bytes on track, where I is for keyed record which is not the last on track, L is for keyed record which is the last on track, and K is a decrement applied to records which have no key.
DL$FG4	Disc prep	71	Binary	Flag — Bit — Meaning 0—5 Reserved 6,7 Device-dependent characteristics
DL$TO4	Disc prep	72—73		Tolerance — a device-dependent factor which is used to calculate effective record lengths for that device.
DL$LT4	Disc prep	74		Labels per track — a device-dependent factor specifying the number of 140-byte labels possible in a VTOC track.
DL$BK4	Disc prep	75		Blocks per track — a device-dependent factor specifying the number of directory blocks of a partitioned file which can be written on a track.
DL$FO4	Disc prep	76—80	Discontinuous binary	Format zero address in the form *CCHHR* — points to the first available format zero record in the VTOC.
		81—99		Reserved.
DL$F64	Space mgmt	100-104		Format 6 address in the form *CCHHR* — points to the first format 6 label created by space management.
DL$VX4	Disc prep	105—114		VTOC extent — describes the extent occupied by the VTOC itself. The format of this field is identical to the fields describing the extent in the format 1 and 3 labels.
		115—139		Reserved.

D.2.3. Disk Format 5 Label

The format 5 label is the second record in the VTOC and is used to maintain control of the available extents in the volume at any time.

The format 5 label is initialized by the disk initialization routine and maintained by the disk space management routine. Each format 5 label may define up to 26 available extents. Format 5 labels may be linked together should more than one become necessary. Figure D—4 shows the format 5 label; Table D—3 summarizes its contents.

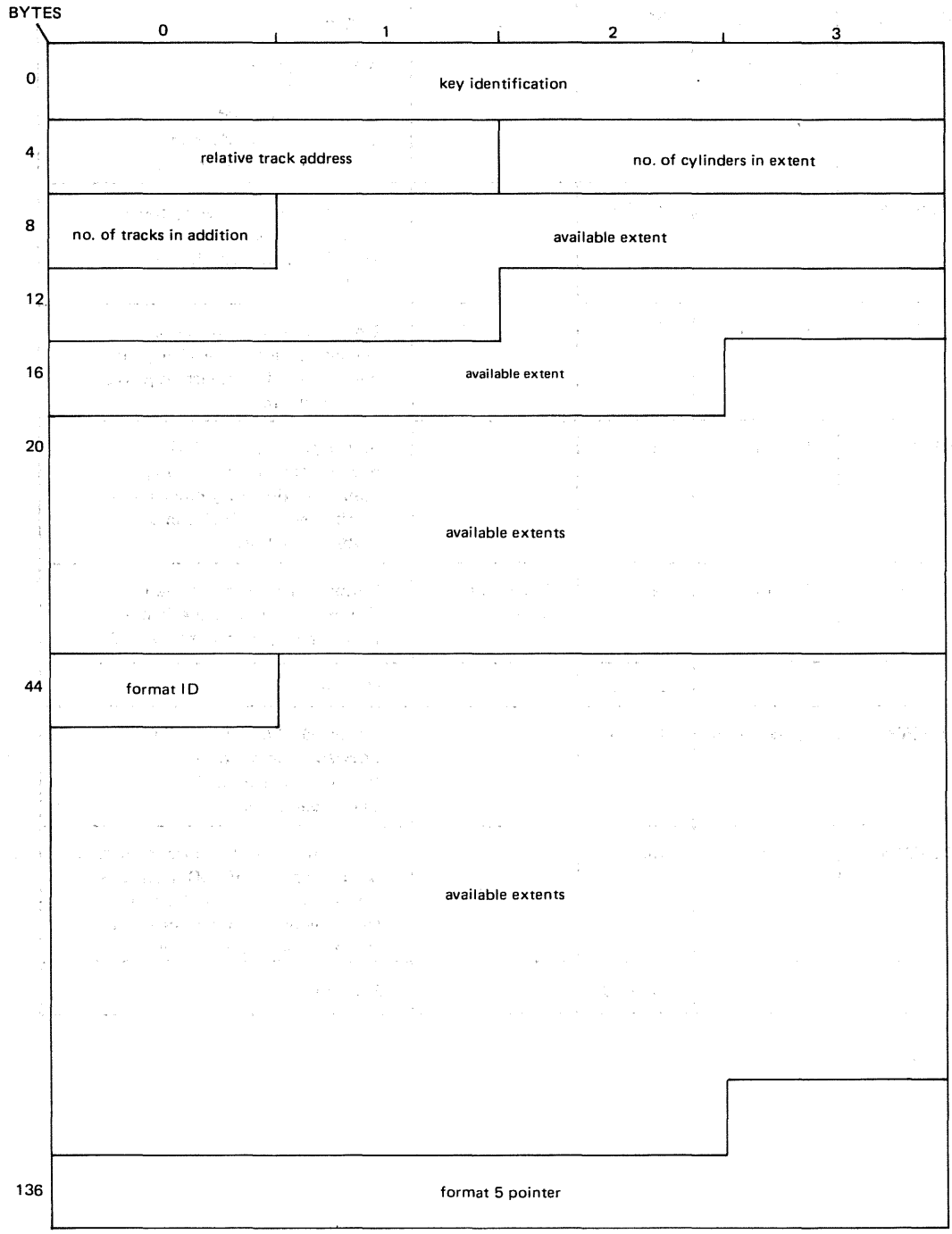


Figure D—4. Disk Format 5 Label

Table D—3. Contents of Disk Format 5 Label

Label	Initialized by	Bytes	Code	Description
DL$ID5	Disc prep	0–3	Hexadecimal	Key identification — Each byte of this field contains 05_{16}.
DL$XT5	Disc prep	4–5	Discontinuous binary	Relative track address — start of extent.
DL$XC5	Disc prep	6–7	Binary	Number of cylinders in extent.
DL$XE5	Disc prep	8	Binary	Number of tracks in extent in addition to the cylinders.
	Space mgmt	9–13		Available extent — describes another extent in fields with the same format as bytes 4 through 8 above.
	Space mgmt	14–43		Six more available extents.
DL$FI5	Disc prep	44	EBCDIC	Format ID — always 5, for format 5 label.
DL$XS5	Space mgmt	45–134		Eighteen more available extents.
DL$CP5	Space mgmt	135–139	Discontinuous binary	Pointer — indicates the address of another format 5 label, in the form *CCHHR*. Binary 0 if no further label.

D.2.4. Disk Format 6 Label

The format 6 label is used to control split-cylinder allocation. Each format 6 label contains a code that identifies all member files sharing the same extent area. Each member file is allocated from 1 to *n* tracks within each cylinder allocated to the set, where *n* is the number of tracks per cylinder, minus one. Additionally, a head pool is maintained that specifies all tracks not currently allocated and available for use by new members of the same split-cylinder set. A format 6 label will be created for each split-cylinder set defined.

The format 6 label is created and maintained by the disk space management routines. Each label contains the disk address of the format 3 label that defines the extents allocated to that split member set. The disk address of the first format 6 label is maintained in the format 4 label. If more than one format 6 label is required, they are linked together.

Note that no extent information is maintained in the format 1 label of a split-cylinder file and that all members of a split-cylinder set share a common format 3 label. Figure D—5 shows the format 6 label; Table D—4 summarizes its contents.

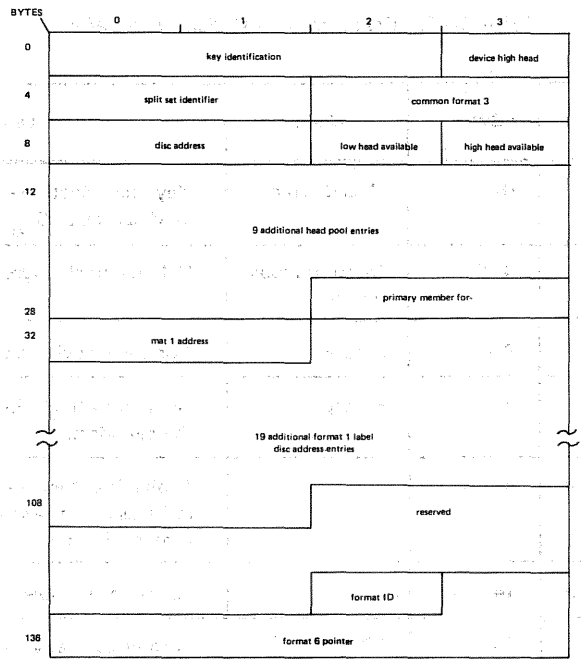


Figure D—5. Disk Format 6 Label

Table D—4. Contents of Disk Format 6 Label

Label	Initialized by	Bytes	Code	Description
DL$ID6	Space mgmt	0–2	Hexadecimal	Key identification — always 060606_{16}
DL$HH6		3		Device high head.
DL$SET		4–5		Set identifier — identifies each member file of the split-cylinder set.
DL$IDF36		6–9	Discontinuous binary	Disk address of the format 3 label shared by all member files
DL$LHA6		10	Hexadecimal	Low head available in the specified extent areas.
DL$HHA6		11	Hexadecimal	High head available in the specified extent areas.
	Space mgmt	12–29	Hexadecimal	Nine additional entries for low and high available head.
DL$IDF16		30–33	Hexadecimal	Format 1 disc address of primary member (CCRH) 19 additional split set format 1 disc address entries in the same format as bytes 30–33.
		34–109	Hexadecimal	Format 1 label disk addresses of up to 19 additional members of the split-cylinder set in the same format as bytes 30–33
		110–133		Reserved
DL$FI6		134	Hexadecimal	Format identification X'F6'.
DL$CP6		135–139	Discontinuous binary	Pointer to next format 6 label in the form *CCHHR*.

D.2.5. Disk Format 0 Label

The format 0 label is used to identify label records in the VTOC not currently in use.

Format 0 records are initialized by the disk initialization routine. The address of the first format 0 is placed in the format 4 label, and each format 0 label is linked to the next. The remainder of the label is filled with binary 0's. Figure D—6 shows the format 0 label; Table D—5 summarizes its contents.

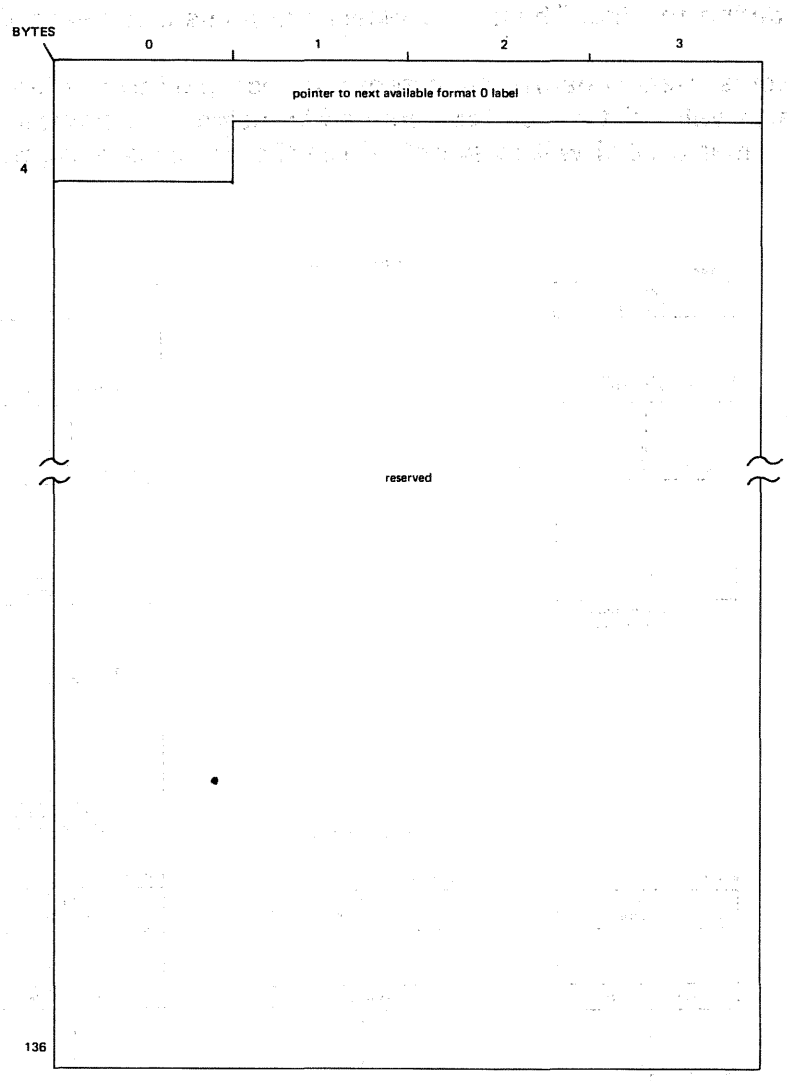


Figure D—6. Disk Format 0 Label

Table D—5. Contents of Disk Format 0 Label

Label	Initialized by	Bytes	Code	Description
	Disc prep	0–4	Discontinuous binary	Disc address in the form *CCHHR* of the next available format 0 label.
		5–139	Binary zero	Reserved.

D.3. FILE INFORMATION GROUP

The file information group (Figure D—7) is composed of the format 1, format 2, and format 3 labels. The format 1 label is normally the first referenced label of the group. It is obtained by executing a key search for the file ID in the VTOC extent defined in the format 4 label.

The format 1 label defines the characteristics of the file and may define up to three extents occupied by the file. The format 1 label is linked to the format 2 label, which is used to further define the file. These two labels are present for each file in the volume.

The format 3 label is used to define the extent area occupied by the file and is an optional label, except that it will exist for all files created by using split-cylinder allocation. For all other files, the format 3 label will exist only if the file occupies more than three separate extent areas.

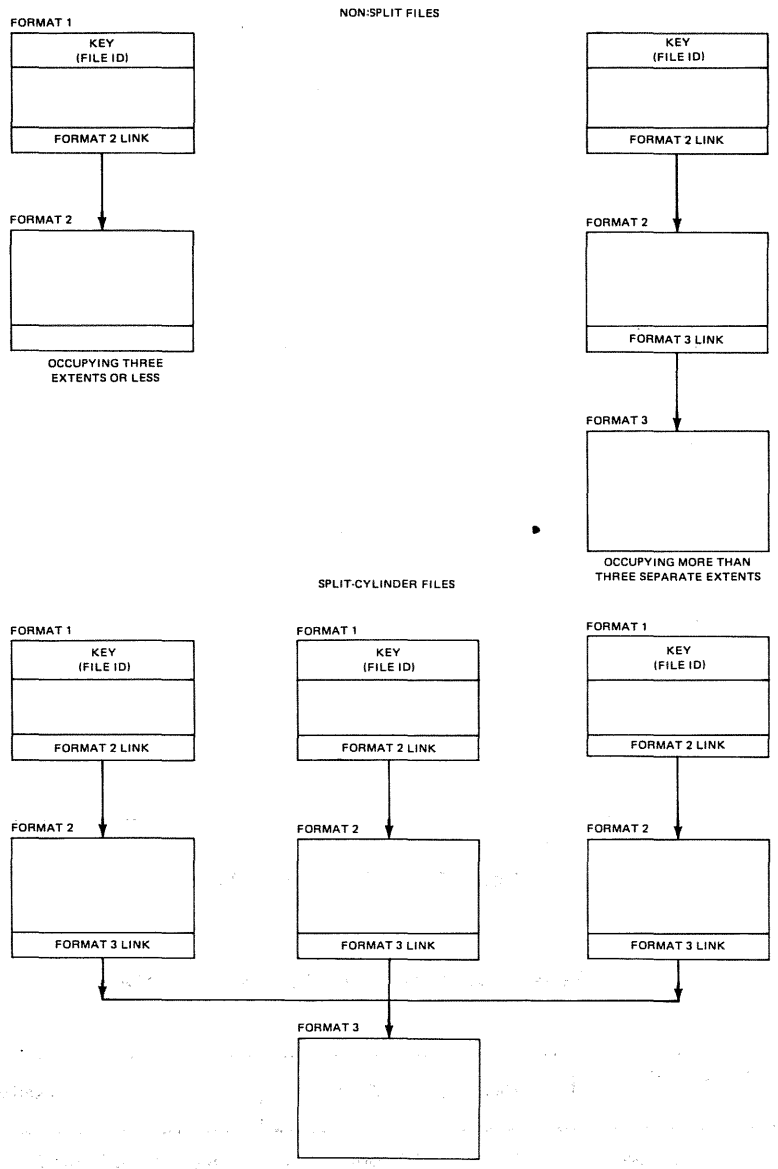


Figure D—7. File Information Group Label Chain

D.3.1. Disk Format 1 Label

A format 1 label exists for each file in a volume. As many as three extents of a file may be described in the format 1 label, provided that the file is not a member of a split-cylinder set.

The format 1 label is initialized by the disk space management routines. It is maintained by both the space management and data management routines. The format 1 label contains a pointer to the format 2 label. Figure D—8 shows the format 1 label; Table D—6 summarizes its contents.

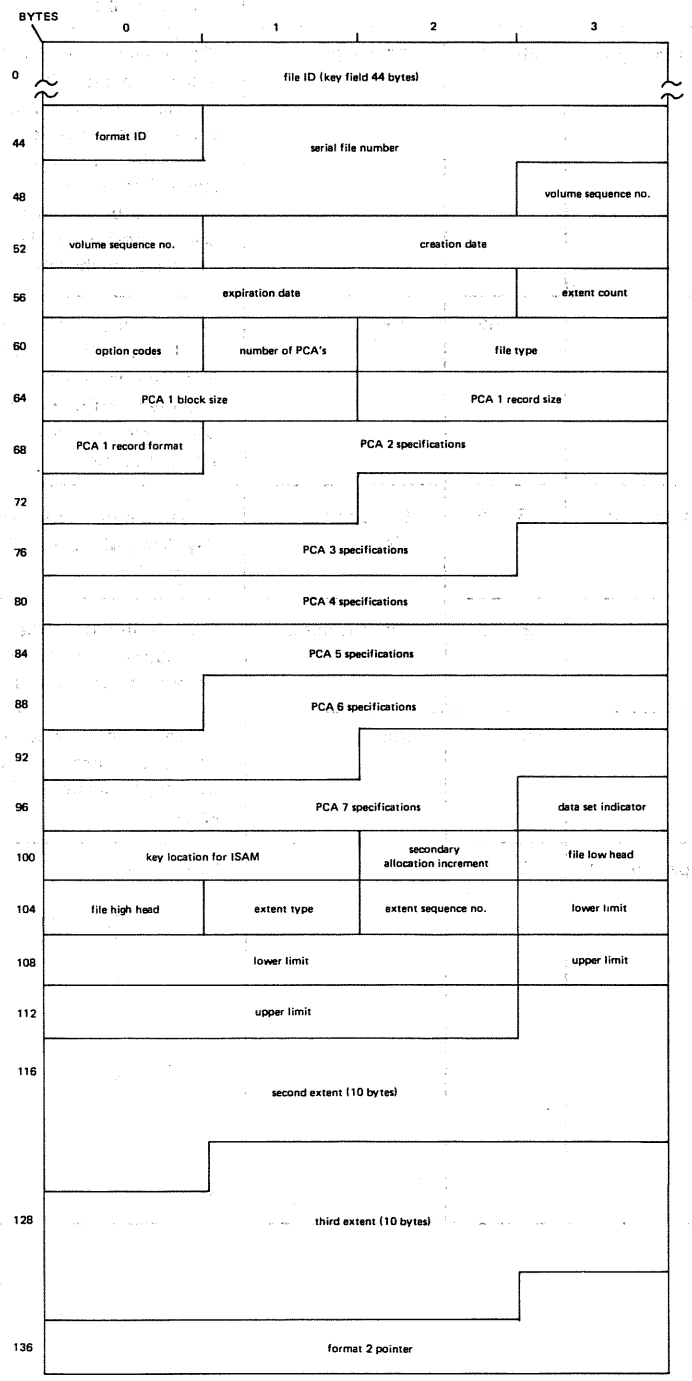


Figure D—8. Disk Format 1 Label

Table D—6. Contents of Disk Format 1 Label (Part 1 of 5)

Label	Initialized by	Bytes	Code	Description
DL$KEY1	Space mgmt	0—43	EBCDIC	File identifier — Each file must have a unique 1- to 44-byte name in this key field, the first six bytes of which may be a lock ID. A search of the VTOC is made on this name.
DL$ID1		44	EBCDIC	Format identifier — always 1, for format 1 label.
DL$FS1	Data mgmt	45—50	EBCDIC	File serial number—identifies the volume on which the file starts, is a 6-digit alphanumeric number, and is the same as the volume serial number of the volume on which the file starts. The first volume of a file is defined by the first job control DVC statement in the device assignment set for the file.
DL$VS1		51—52	Binary	Volume sequence number — indicates the number of this volume relative to the first volume in the file. The first volume of a file is defined by the first job control DVC statement in the device assignment set for the file.
DL$CD1	Space mgmt	53—55	Discontinuous binary	Creation date — format is *YDD* (year-day-day), where *Y* is 0 to 99, and *DD* is 1 to 366.
DL$ED1		56—58	Discontinuous binary	Expiration date — the date when the file may be deleted. Format is the same as the creation date.
DL$XC1		59	Binary	Extent count — specifies the number of extents currently constituting the file, or portions of it, on this volume.
DL$OC1	Space mgmt	60	Binary	Option codes Bit — Content — Meaning 0 — 1 — Preformatted by VTOC 1 — 1 — Allocation by cylinder 2 — 1 — New file 3 — 1 — Partitions cylinder aligned 4—7 — Unused

Table D—6. Contents of Disk Format 1 Label (Part 2 of 5)

Label	Initialized by	Bytes	Code	Description
DL$PC1	Data mgmt	61	Binary	PCA count — number of partitions which constitute the file.
DL$FT1	Data mgmt	62	Hexadecimal	File type Hexadecimal Code — Meaning 20 — Sequential (DTFSD) 40 — Direct access (DTFDA) 60 — Nonindexed (DTFNI) 80 — Indexed sequential (DTFIS) 90 — IRAM (DTFIR) or MIRAM (DTFMI) 02 — SAT (DTFPF) 00 — Undefined
		63	Hexadecimal	File type Hexadecimal Code — Meaning 00 — IRAM file, nonindexed 11 — IRAM file, indexed 80 — MIRAM file, IRAM characteristics C0 — MIRAM file, MIRAM characteristics NOTE: This byte is meaningless unless byte 62 equals X'90'.
DL$BL1	Data mgmt	64—65	Binary	Reserved for PCA1 block length — size of fixed-length blocks or maximum size of variable-length blocks.
DL$RL1	Data mgmt	66,67	Binary	Reserved for PCA1 record length — size of fixed-length records or maximum size of variable-length records.
DL$RF1	Data mgmt	68	Binary	Reserved for PCA1 record format Bit — Content — Meaning 0,1 — Reserved 2 — 0 — Records have no keys. 2 — 1 — Records have keys.

Table D—6. Contents of Disk Format 1 Label (Part 3 of 5)

Label	Initialized by	Bytes	Code	Description
				(Record format, cont) Bit — Content — Meaning 3 — 1 — Fixed-length blocked records 4 — 1 — Variable-length blocked records 5 — 1 — Fixed-length unblocked records 6 — 1 — Variable-length unblocked records 7 — 1 — Records are interlaced.
	Data mgmt	69—73	Discontinuous binary	Partition descriptor 2; block size, record size, and record format for partition 2.
	Data mgmt	74—78	Discontinuous binary	Partition descriptor 3.
	Data mgmt	79—83	Discontinuous binary	Partition descriptor 4.
	Data mgmt	84—88	Discontinuous binary	Partition descriptor 5.
	Data mgmt	89—93	Discontinuous binary	Partition descriptor 6.
	Data mgmt	94—98	Discontinuous binary	Partition descriptor 7.
DL$DS1	Space mgmt	99	Binary	Data set indicators — reserved for future use.
DL$KL1	Data mgmt	100-101	Binary	Key location — high order position of key field within each data record of an indexed-sequential file.

Table D—6. Contents of Disk Format 1 Label (Part 4 of 5)

Label	Initialized by	Bytes	Code	Description
DL$SA1		102	Binary	Secondary allocation increment — the number of cylinders of disc storage to be requested for each dynamic extension of the file.
DL$LH1		103	Hexadecimal	File low head — split cylinder allocation.
DL$HH1		104	Hexadecimal	File high head — split cylinder allocation.
DL$XT1		105	Hexadecimal	Extent type indicator — Code — Meaning 00 — No valid extent described 20 — Sequential file (DTFSD) 40 — Direct access file (DTFDA) 60 — Nonindexed file (DTFNI) 80 — Indexed sequential file (DTFIS) 90 — IRAM (DTFIR) or MIRAM (DTFMC) 02 — SAT (DTFPF) FF — Job Control
DL$XS1		106	Binary	Extent sequence number — relative number of extents in multiple-extent volume.
DL$XL1		107-110	Discontinuous binary	Lower limit — the address specifying the start of the extent, in the form *CCHH.*

Table D—6. Contents of Disk Format 1 Label (Part 5 of 5)

Label	Initialized by	Bytes	Code	Description
DL$XU1		111–114	Discontinuous binary	Upper limit — the address specifying the end of the extent, in the form *CCHH.*
		115–124		Second extent — same format as described for bytes 105 through 114.
		125–134		Third extent — same format as second extent.
DL$CP1	Space mgmt	135–139	Discontinuous binary	Continuation pointer — the address of a format 2 label. The address is in the form *CCHHR.*

D.3.2. Disk Format 2 Label

The format 2 label is used as an extension to the format 1 label to further describe the file.

For nonindexed files (DTFSD, DTFDA, DTFNI), bytes 1 through 43 are used to carry partition information in a maximum of seven 6-byte entries. For indexed ISAM files, bytes 13 through 43 are used to carry index control information. For IRAM and MIRAM files, bytes 13 through 43 are used to carry index control and file characteristic information. For library files, bytes 32 through 47 are used to carry information on the library text and directory; bytes 13 through 31 contain binary zeros.

The format 2 label is initialized by space management and maintained by data management. The label is always present and is linked from the format 1 label. The link field in the format 2 label will point to a format 3 label, if used. This pointer will be present for all split-cylinder files and for nonsplit-cylinder files requiring more than three extents. If it is not present, the field is filled with binary zeros. Figure D—9 shows the format 2 label; Table D—7 summarizes its contents. The format of the ISAM file information area is shown in Figure D—10, and the contents of this area are listed in Table D—8. The format of the IRAM/MIRAM file information area is shown in Figure D—11, and the contents of this area are listed in Table D—9. The format of the library file information area is shown in Figure D—12, and the contents of this area are listed in Table D—10.

BYTES	0	1	2	3
0	key ID	nonindexed LBC	key length or lace factor	reserved
4	EOD ID			nonindexed LBC
8	key length or lace factor	reserved	EOD ID	
12	EOD ID			
	(up to five additional partition descriptors)			
40				
44	reserved		blocks/track, PCA1	
48	PCA1ID 0 2	relative track addr-1* 3 15	tracks 16 31	
52	PCA2ID 0 2	relative track addr-2* 3 15	tracks 16 31	
128	tracks per cylinder		file low head no.	
132	relative extent count		flags	
136	format 3 pointer			

*Thirteen bits can represent a maximum relative track address (RTA) of 8191_{10} ($1FFF_{16}$). To support the larger 8433 disc, the high-order bit of the tracks field (bit 16) of the logical extent is used to indicate that the RTA must be increased by a constant value of 8192_{10}. (See Table D–7.)

Figure D—9. Disk Format 2 Label, Nonindexed Files (DTFSD, DTFDA, DTFNI)

BYTES	13	14	15
	disc address of last prime . . .		
16	. . . data record, plus 1	number of cylinders having overflow space full	
20	current independent overflow address		
24	prime data load count	overflow record count	
28	reserved	disc address of last block level . . .	
32	. . . index record, plus 1	overflow retrieval count, except first of chain	
36	reserved	total count of prime data records	
40	bytes of main storage required for top index	number of records tagged for deletion	
44	number of blocks per cylinder		

Figure D—10. ISAM (DTFIS) File Information Area, Disk Format 2 Label

BYTES		13	14	15
		key location for key 1		key length for key 1
16	used for IRAM processing key characteristic flag byte (MIRAM)	descriptor for...		
20	key 2	descriptor for...		
24	...key 3	descriptor for...		
28	...key 4	descriptor for...		
32	...key 5	sector offset	number of records...	
36	...in file, plus 1	record size		fine-level index block size in sectors
40	number of index levels	last fine-level index block		

NOTE:

Descriptions that pertain to IRAM files also apply to MIRAM files with IRAM characteristics.

Figure D—11. IRAM/MIRAM File Information Area, Disk Format 2 Label

BYTES	28	29	30	31
28	directory partition lace factor			
32	directory partition lace adjustment factor			
36	text partition lace factor			
40	text partition lace adjustment factor			
44	number of library blocks per track			

NOTE:

In the format 2 label for library files, byte 3 and bytes 13—27 are reserved and contain binary zero.

Figure D—12. Library File Information Area, Disk Format 2 Label

Table D—7. Contents of Disk Format 2 Label (Part 1 of 3)

Label	Initialized by	Bytes	Code	Description
DL$SID2	Space mgmt	0	Hexadecimal	Key identification X'02'
DL$SPC2	Data mgmt	1		Nonindexed last block control — the number of logical records in the last block of the partition for fixed-length blocked files.
DL$SLF2		2		Key length or lace factor.
DL$SLA2		3		Reserved
DL$SEP2	Data mgmt	4—6	Binary	End of data ID — relative block address plus 1 of the last block written into the partition.
		7—12		A 6-byte partition descriptor entry in the same form as bytes 1—6.
	Data mgmt	13—43	Hexadecimal	For nonindexed files, up to 5 additional partition descriptors. For ISAM files, see index information area (Table D—8 and Figure D—10). For IRAM and MIRAM files, see IRAM/MIRAM information area (Table D—9 and Figure D—11). For library files, see library information area (Table D—10 and Figure D—12).
		44—45		Unused (binary zero) for all but indexed files; reserved for indexed files.

Table D—7. Contents of Disk Format 2 Label (Part 2 of 3)

Label	Initialized by	Bytes	Code	Description
DL$SBPT2		46—47		Blocks per track — the number of blocks per track in the first or only partition of the file.
DL$SXAR2	Data mgmt	48—127	Discontinuous binary	Logical extent table area — These entries are 4 bytes in length and specify PCA ID in 3 bits, starting relative track address in 13 bits, and number of tracks in that address. From one to twenty 4-byte logical extent entries may be placed in this 80-byte area. Each 4-byte entry has the following format: Bit — Meaning 0—2 — The high-order three bits of the logical extent identify the partition to which it is assigned. (This value may be from 1 to 7.) 3—15 — The next 13 bits indicate the relative track address of the logical extent. 16—31 — If the first bit (bit 16) of the track field is set on, a value of 8192 must be added to the relative track address to indicate the relative track address of the logical extent on the 8433 disc. The remaining 15 bits indicate the number of tracks contained in the extent.
DL$TPC2		128—129	Hexadecimal	Tracks per cylinder for this file.
DL$FLH2		130—131		File low head — the lowest head number in the assigned cylinders accessible for this file.
DL$SXCT2		132—133		Number of relative extents contained in this label.
DL$SFL2		134		Flags. Bit — Content — Meaning 0, 1, 2, 3, 4 — Reserved 5 — 1 — Library lace adjustment, type 2 6 — 1 — Library lace adjustment, type 3 7 — 1 — 9400 SAT compatible

Table D—7. Contents of Disk Format 2 Label (Part 3 of 3)

Label	Initialized by	Bytes	Code	Description
DL$SCID2	Space mgmt	135—139	Discontinuous binary	The disc address, in the form *CCHHR*, of the format 3 label (if required) associated with this file.

Table D—8. Contents of Indexed File Information Area, Disk Format 2 Label

Label	Initialized by	Bytes	Code	Description
DL$PID2	Data management	13—17	Discontinuous binary	Disc address of last prime data record (plus 1), in the form *rrrbb*, where *rrr* = relative block address and *bb* = displacement within the block
DL$NMA2		18—19	Hexadecimal	Count of cylinders having overflow space filled
DL$IOF2		20—23		Current address of independent overflow (*rrr*)
DL$PDLC2		24—25	Discontinous binary	Prime data load count
DL$NMO2	Data management	26—27	Hexadecimal	Count of the number of overflow records in the file
—		28		Reserved
DL$BID2		29—33	Discontinuous binary	Disc address of last block level index record (plus 1), in the same form as bytes 13—17
DL$NMR2		34—35		Overflow chain retrieval count, not first of chain
—		36		Reserved
DL$NMP2		37—39		Total count of number of prime data records
DL$NMS2	Data management	40—41	Hexadecimal	Number of bytes required to hold top index in main storage
DL$NMT2		42—43		Number of records user has tagged for deletion
—		44—45		Number of blocks per cylinder

Table D—9. Contents of IRAM/MIRAM File Information Area, Disk Format 2 Label

Label	Initialized by	Bytes	Code	Description
DL$XILOC	Data mgmt	13—14	Hexadecimal	Key location for key 1
		15		Key length for key 1
		16	Hexadecimal (IRAM)	Used for IRAM file index processing
			Binary (MIRAM)	For MIRAM files, byte 16 contains key 1 characteristics: Bit — Content — Meaning 0 — 1 — Duplicates allowed for this key 1 — 1 — Key change allowed for this key during during update 2—4 — Unused *5 — 1 — Index-only records permitted in this file *6 — 1 — Variable-length record format *7 — 1 — Record control byte (rcb) present
		17—20	Discontinuous binary	Descriptor for key 2 (binary zeros for IRAM)
		21—24		Descriptor for key 3 (binary zeros for IRAM)
		25—28		Descriptor for key 4 (binary zeros for IRAM)
		29—32		Descriptor for key 5 (binary zeros for IRAM)
DL$MARSO		33	Hexadecimal	Sector offset for files created with recovery
DL$COUTR		34—36		Number of records in file (plus 1)
DL$REC		37—38		Record size
DL$CSIZ		39		Fine-level index block size in sectors
DL$CLEV		40		Number of index levels
DL$FAB		41—43		Relative block number of last block of the fine-level index

*These bit positions are unused in the descriptors for keys 2 through 5.

NOTE:

Descriptions pertaining to IRAM files also apply to MIRAM files with IRAM characteristics.

Table D—10. Contents of Library Information Area, Disk Format 2 Label

Label	Initialized by	Bytes	Code	Description
DL$DIRL2	Data management	28–31		Hardware-adjusted lace factor for the directory partition
DL$DIRF2	Data management	32–35		Rotational adjustment for directory lace factor
DL$TXTL2	Data management	36–39		Hardware-adjusted lace factor for the library text partition
DL$TXTF2	Data management	40–43		Rotational adjustment factor for the library's text
—	Data management	44–47		Number of library blocks per track

D.3.3. Disk Format 3 Label

The format 3 label is used to maintain extent information for the file. For split-cylinder files, a format 3 label is always present. For files not using split-cylinder allocation, a format 3 label for the file will exist only when more than three extents are required.

The format 3 label is initialized and maintained by the disk space management routines. The format 3 label, when required, will always be linked from a format 2 label. Figure D—13 shows the format 3 label; Table D—11 summarizes its contents.

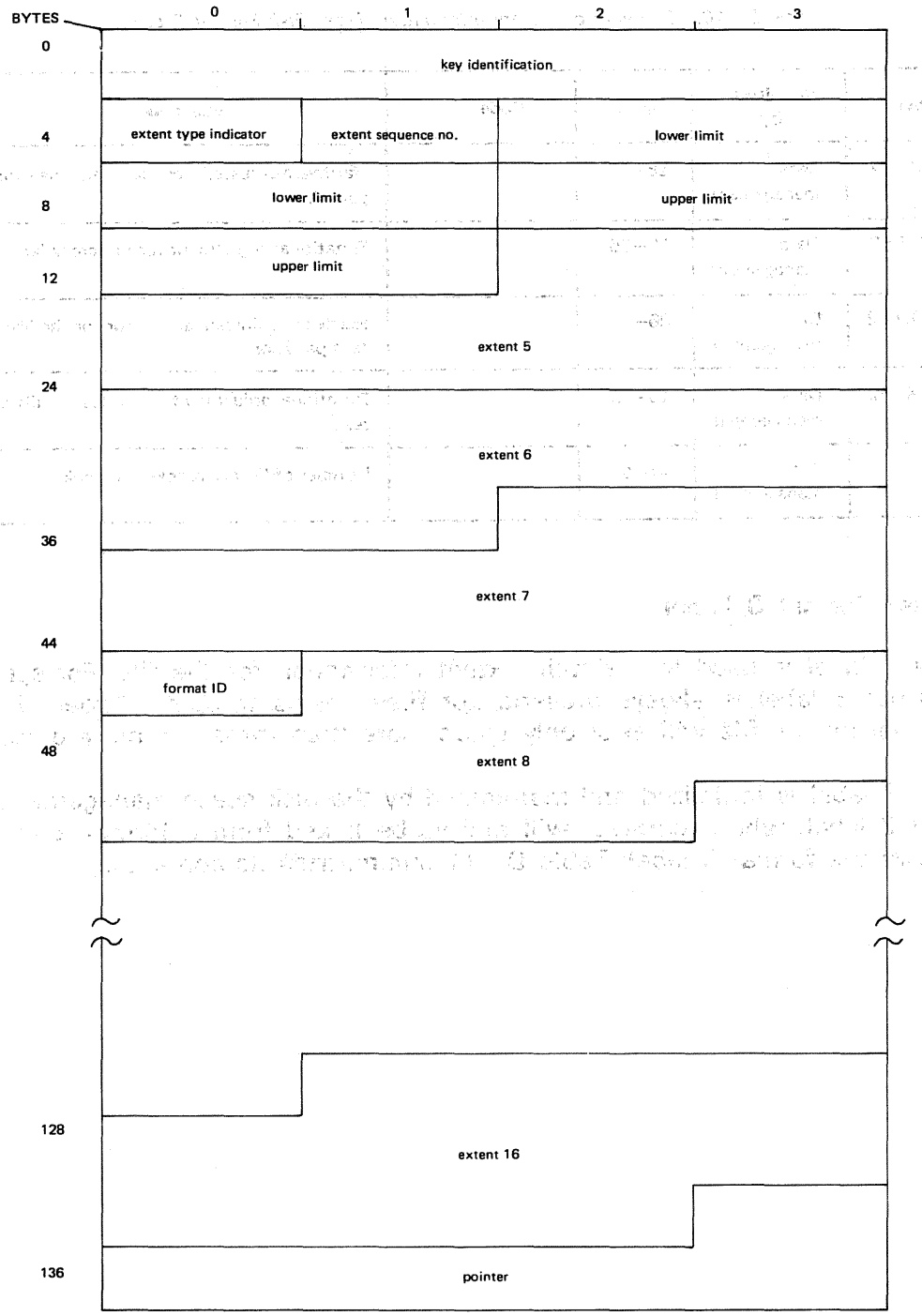


Figure D—13. Disk Format 3 Label

Table D—11. Contents of Disk Format 3 Label

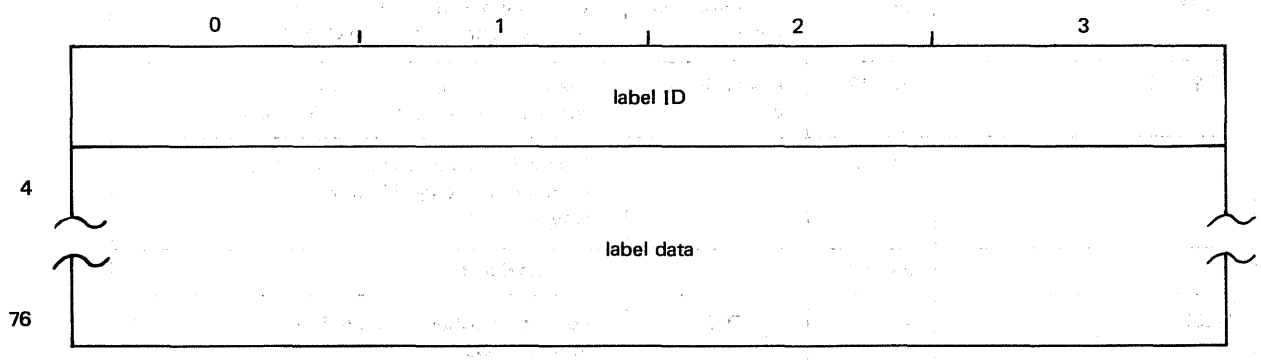
Label	Initialized by	Bytes	Code	Description
DL$ID3	Space mgmt	0—3	Hexadecimal	Key identification — each byte contains 03_{16}.
DL$XT3		4	Hexadecimal	Extent type indicator — Code — Meaning 00 — No valid extent described 01 — Prime data area
DL$SN3		5	Binary	Extent sequence number — relative number of extents in this volume of the file.
DL$XL3		6—9	Discontinuous binary	Lower limit — starting track address of the extent, in the form *CCHH*.
DL$XU3		10—13	Discontinuous binary	Upper limit — terminating track address of the extent, in the form *CCHH*.
		14—23		Extent 5 — same format as described for bytes 4 through 13 for this extent.
		24—43		Extents 6 and 7.
DL$FI3		44	EBCDIC	Format identifier — always 3, for format 3 label.
DL$XS3		45—134		Extents 8 through 16.
DL$CP3		135—139	Discontinuous binary	Pointer — address of next format 3 label, in the form *CCHHR*. Binary 0 if no further label.

D.4. OPTIONAL USER STANDARD LABELS

Optional user standard labels are records made available to you via your label processing routine (LABADDR) at the opening or closing of a disk volume. OS/3 data management supports user standard labels for your SAM and DAM disk files described by the DTFSD, DTFNI, and DTFDA declarative macroinstructions; it does not support them for your ISAM files, which are described by the DTFIS macro.

D.4.1. User Header Labels

If you require user header labels, these will be written on the first track of each volume of a DTFSD file and on the first track of the first volume of a DTFDA or DTFNI file. You may write a maximum of eight labels. Figure D—14 shows the format of the 80-byte user header label; its contents are explained in the table below Figure D—14.

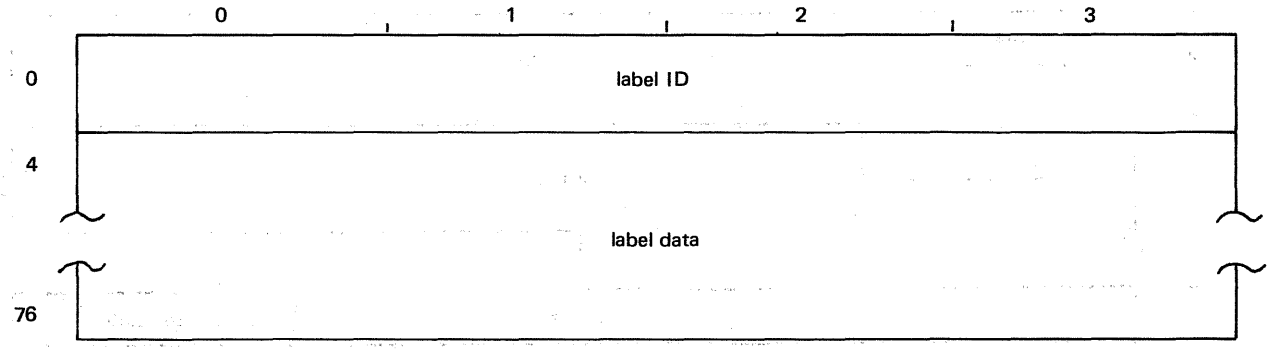


Field	Bytes	Code	Description
Label ID	0—3	EBCDIC	Contains 4-byte label identifier: UHL, followed by a label number which ranges from 1 through 8.
Label data	4—79	User option	Contains 76 bytes of user specified header label data.

Figure D—14. Optional User Standard Header Label

D.4.2. User Trailer Labels

If you need user trailer labels, these will be written on the first track of each volume of a DTFSD file and on the first track of the first volume of DTFDA or DTFNI files, following your user header labels. You may write a maximum of eight labels on DTFDA and DTFNI files, and eight labels per volume on DTFSD files. Figure D—15 shows the format of the 80-byte user trailer label; its contents are explained in the table below Figure D—15.



Field	Bytes	Code	Description
Label ID	0—3	EBCDIC	Contains 4-byte label identifier: UTL, followed by a label number which ranges from 1 through 8.
Label data	4—79	User option	Contains 76 bytes of user-specified trailer label data

Figure D—15. Optional User Standard Trailer Label

D.5. 8413 DISKETTE FILE LABEL

Figure D—16 illustrates the 8413 diskette file label format. Table D—12 explains the contents of each field in the diskette file label.

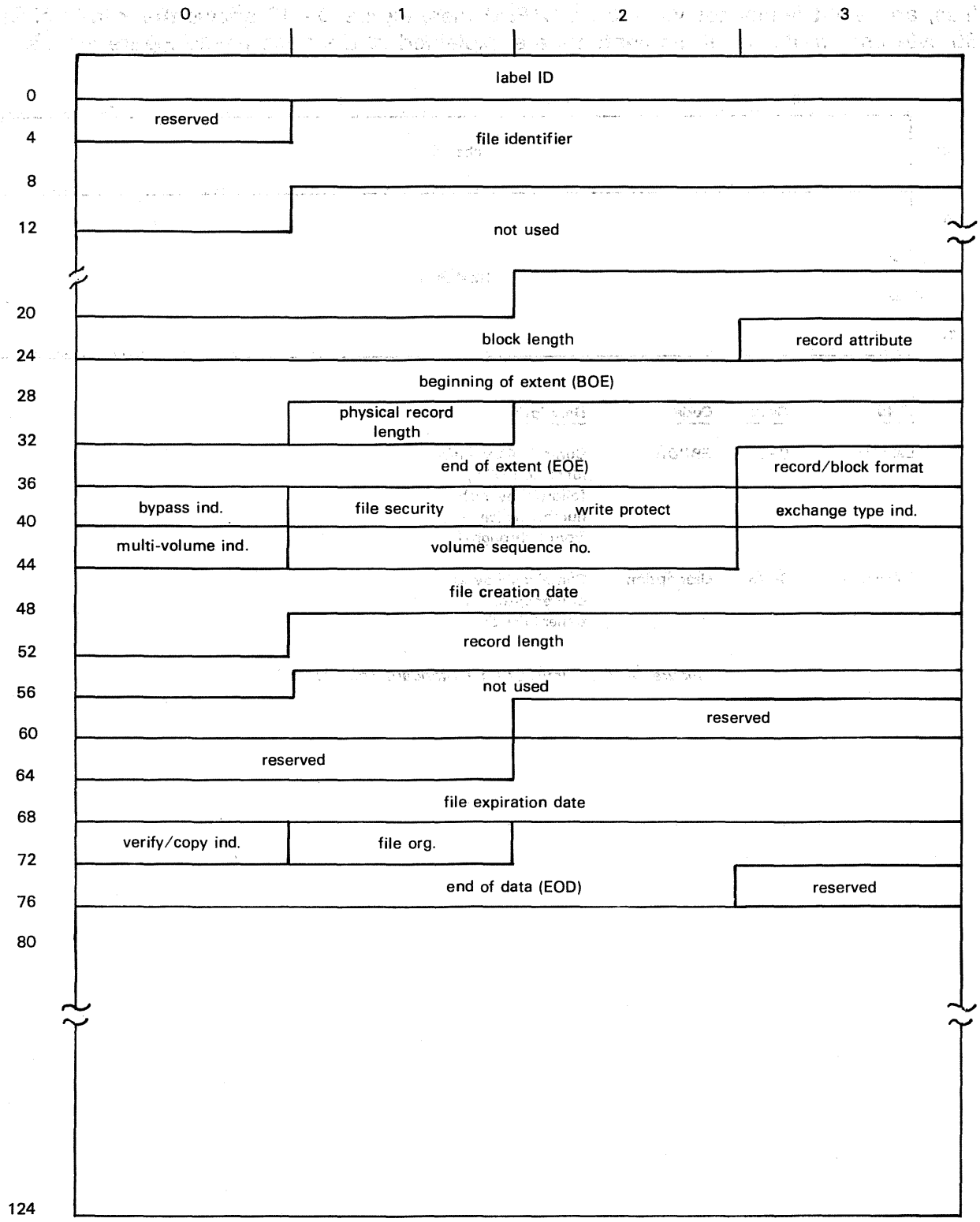


Figure D—16. 8413 Diskette File Label Format

Table D—12. Diskette File Label Description (Part 1 of 2)

Field	Byte Position	Description
label ID	0—3	Contains 4-byte label identifier HDR followed by the number 1
Reserved	4	Reserved
File identifier	5—12	Names user file and is from 1 to 8 characters. First character must be alphabetic and no blanks are allowed. Duplicate names on the same diskette are not allowed.
—	13—21	Not used
Block length	22—26	Indicates the record size as follows: Character — Meaning Δ — Record size will be obtained from the DTF 1—128 — Actual record size; for example, 80.
Record attribute	27	Blank; indicates unblocked records
Beginning of extent (BOE)	28—32	Identifies the address of the first sector of the file by cylinder number (pos. 28—29), head (pos. 30), and sector number (pos. 31—32).
Physical record length	33	Indicates physical record length and is blank meaning 128 bytes per record.
End of extent (EOE)	34—38	Indicates address of last sector reserved for this file and uses the same format as BOE.
Record/block format	39	This field must be blank.
Bypass indicator	40	Indicates a file to be skipped during exchange or copy operations when transmitting or transferring files on the volume. If position 40 is blank, the file is transferred; if B, the file is not transferred.
File security	41	Blank indicates the file can be accessed. Nonblank indicates restricted access. When nonblank, the volume accessibility indicator in the volume label (track 00, sector 07) must also be nonblank.
Write protect	42	Blank indicates both reading and writing allowed. P indicates only read activities allowed.
Exchange type indicator	43	Blank indicates file can be used for basic exchange. Nonblank indicates additional label checking is needed to exchange the file.
Multivolume indicator	44	Character — Meaning blank — File on 1 diskette C — File continued on next diskette L — Last diskette on which file resides
Volume sequence number	45—46	Indicates volume sequence number in multivolume set (01—99). Blanks indicate no volume sequence checking performed on this device.
File creation date	47—52	Indicates the creation date of the file by year (YY), month (MM), and day (DD). Blanks indicate nonsignificance of creation date.

Table D—12. Diskette File Label Description (Part 2 of 2)

Field	Byte Position	Description
Record length	53—56	Defines record length. ~~bbbb~~ = record length equals block length (position 22) (This field is ignored because blank in position 43 means the same.)
Offset to next record space	57—61	Not used
Reserved	62—65	Reserved
File expiration date	66—71	Contains date of deletion for a file. (Format is the same as creation date in position 47—52.) ~~bbbbbb~~ = file date expired 999999 = file date never expires
Verify/copy indicator	72	Character — Meaning blank — File is created V — File is verified C — File data successfully transferred to another medium
File organization	73	If position 43 contains blank, this field must also contain blanks.
End of data (EOD)	74—78	Identifies address of next unused sector within the file extent using BOE format. — If EOD equals BOE, the extent contains a null file. — If EOD equals address of next block beyond file extent (unblocked records), entire extent was used. — If blocked records are used, EOD must be used with offset to next record space (position 57—61) to find the actual EOD.
Reserved	79	Reserved
	80—127	Padded with binary zeros

NOTE:

ISAM does not support the 8413 diskette.

Appendix E. Magnetic Tape Labels

E.1. OS/3 SYSTEM STANDARD LABELS FOR MAGNETIC TAPE

This appendix describes the system standard labels for magnetic tape files and reels (or volumes) in OS/3. Section 8 describes magnetic tape SAM record formats, volume organization, and file conventions. Section 9 describes the functions and operations of magnetic tape SAM, including certain of the label-processing functions of transients entered by the OPEN and CLOSE imperative macros and the user's own special label processing routine (the address of which he provides to tape SAM by specifying the LABADDR keyword parameter in the DTFMT declarative macro defining any standard labeled magnetic tape file for which optional standard user header or trailer labels (UHL or UTL) are to be processed). The LBRET imperative macro (9.4.9) is issued in the user's label routine to control returns to and from tape SAM.

In OS/3 tape SAM, magnetic tapes may be labeled or unlabeled, and a labeled tape may contain either nonstandard or OS/3 system standard labels. Tape volumes have formats that may be specified as standard, nonstandard, or unlabeled, using the FILABL keyword of the DTFMT declarative macro (9.2.6.1); unlabeled format is assumed if this keyword is not specified.

E.2. SYSTEM STANDARD TAPE LABELS

All standard tape labels, including optional UHL and UTL, are in blocks of 80 bytes. There are five tape label groups: three required, two optional:

- Volume label group
- File header label group
- User header label group (optional)
- File trailer label group
- User trailer label group (optional)

E.2.1. Volume Label Group

The volume label group comprises a single volume label, VOL1. The VOL1 label identifies the tape reel and its owner, and it is used to check that the proper reel is mounted. (Refer to Table 9—3). When a tape is first used at an installation, its volume serial number (VSN) and other volume information, as shown in Figure E—1, are usually recorded on it magnetically with the TPREP utility routine, being specified to the routine with parameter cards. The volume serial number is also written on the exterior of the reel for visual identification to the operator. For a detailed description of TPREP, refer to the system service programs (SSP) user guide, UP-8062 (current version); note that you cannot use this routine to prep a magnetic tape volume dynamically, although you may prep as many as 36 volumes with it.

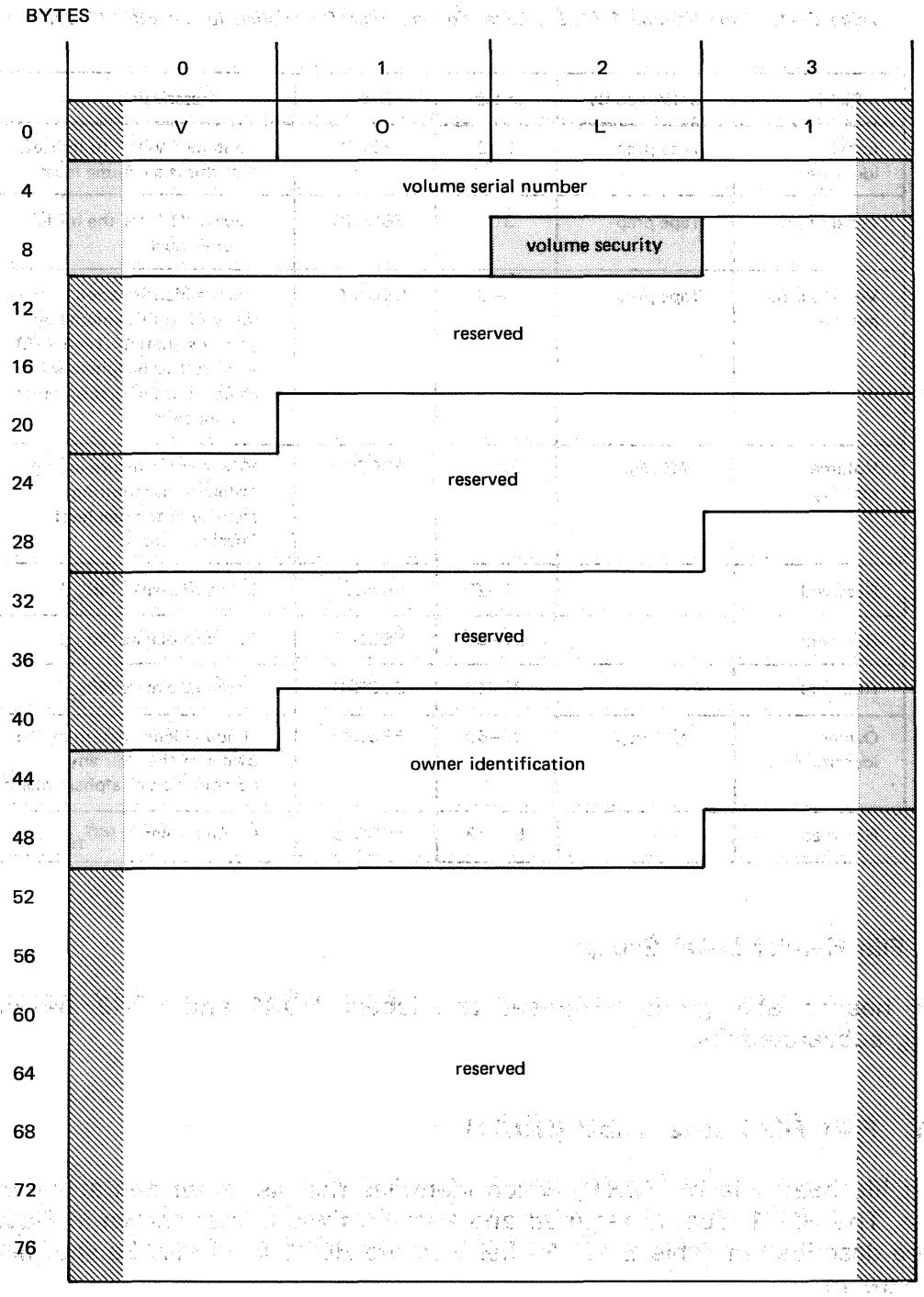
As an alternative to using TPREP, if you want tape SAM to prep the volumes of a standard labeled file dynamically, as a preliminary part of the job step in which you create the file, you specify the parameter (PREP) in the VOL job control statement of the file's device assignment set, also specifying a unique VSN. Data management then preps the volumes from information you supply on the associated VOL and LBL statements. This procedure is described in 9.3.3.3.

When you issue an OPEN macro to an output tape file, its open-and-rewind options are executed first (9.2.5.2 and 9.2.5.3), and then the tape is checked to see if it is at the load point. If it is at the load point, data management reads the VOL1 label (if it is not in the prep mode) and, checking the VSN, saves this for use in writing or reading the file header labels (HDR1 and HDR2). It then positions your tape 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 executed, tape SAM assumes that it 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 VSN 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 (9.3.4), and no further volume label processing is performed. If the input tape is not at the load point after your open-and-rewind options are executed, tape SAM 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 (9.2.5.4, 9.4.2), if you have specified READ=BACK (9.2.5.1), the label is bypassed without processing. Figure E—1 shows the format of the VOL1 label; its fields are described in Table E—1.



LEGEND:

Generated by data management or reserved for system expansion.

Written by data management from user-supplied data.

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

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

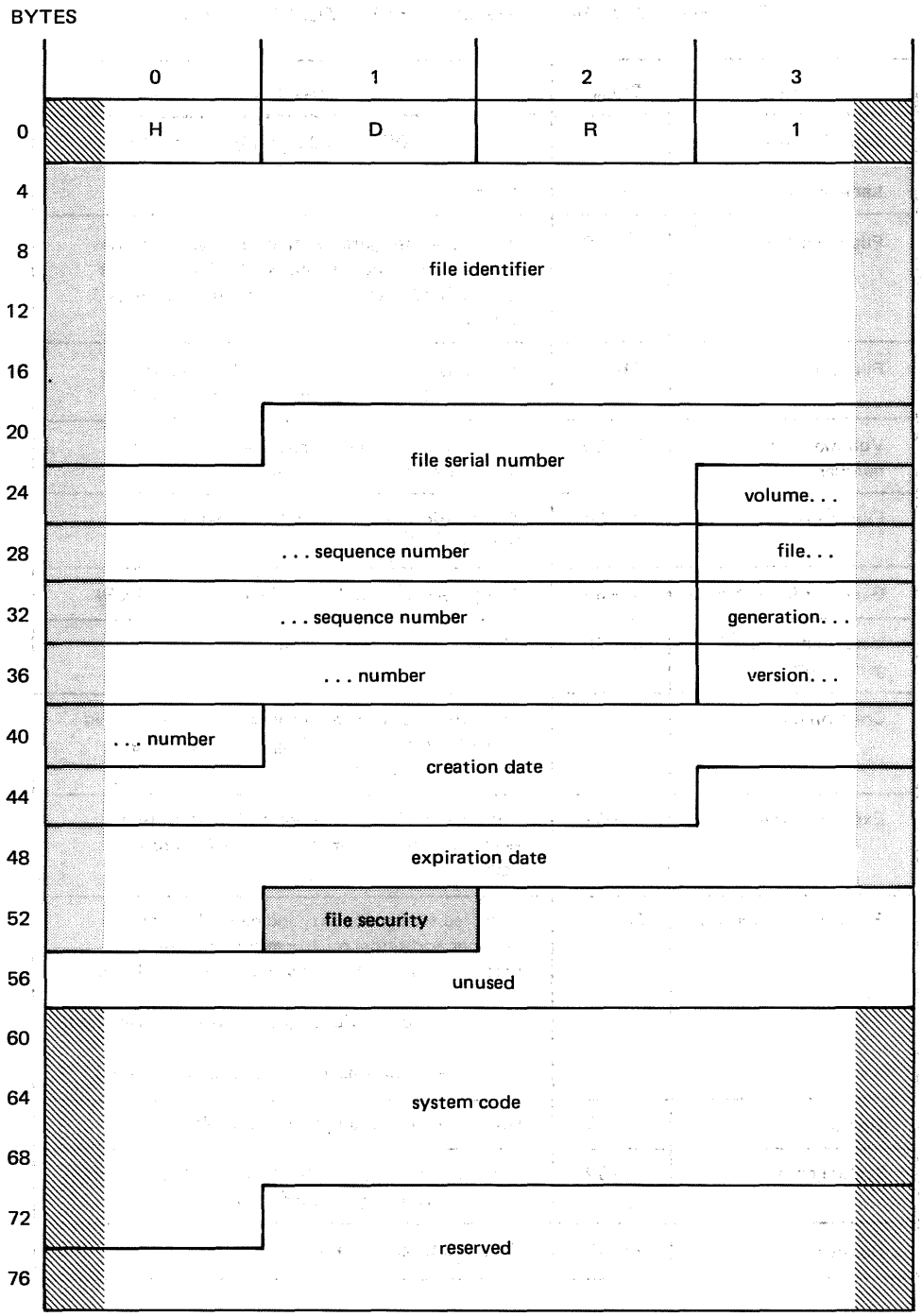
Field	Initialized By	Bytes	Code	Description
Label identifier	Tape prep	0–2	EBCDIC	Contains "VOL" to indicate that this is a volume label
Label number	Tape prep	3	EBCDIC	Always "1", for the initial volume label
Volume serial number	Tape prep	4–9	EBCDIC	Unique identification number assigned to this volume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

E.2.2. File Header Label Group

The file header label group comprises two labels, HDR1 and HDR2, described in the following subparagraphs.

E.2.2.1. 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 E—2; its fields are described in Table E—2. All fields in the HDR1 label may be specified in the job control stream.



LEGEND:

Generated by data management or reserved for system expansion.

Written by data management from user-supplied data.

Figure E—2. First File Header Label (HDR1) Format for an EBCDIC Tape Volume

Table E—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.
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}).

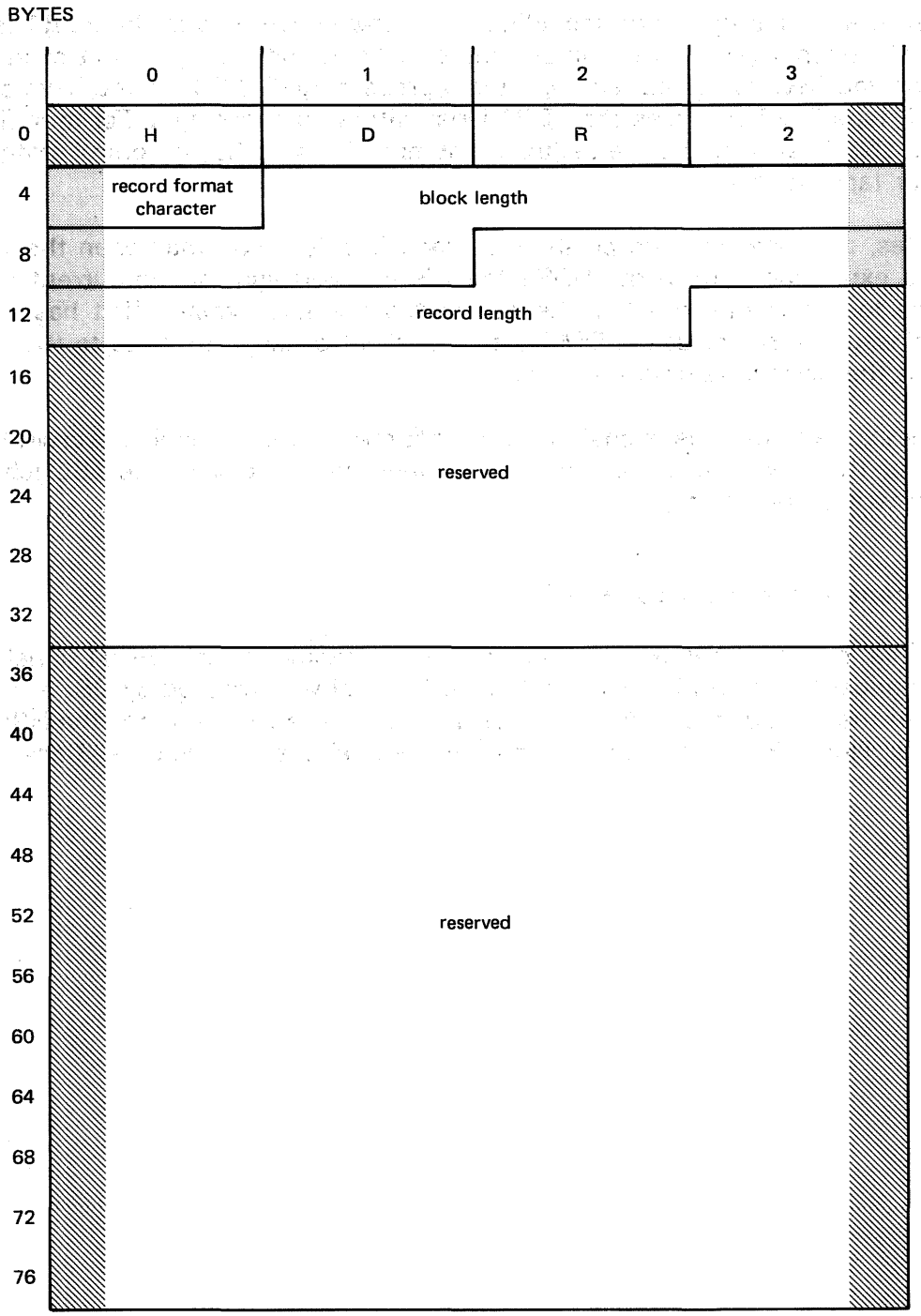
For input tapes, all fields up to and including the *system code* field in the HDR1 label are checked at file open against the values you specify in the LBL job control statement (9.3.4), unless you have specified read-backward processing (9.2.5.1). If you have specified READ=BACK, tape SAM bypasses the HDR1 label without processing it. 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 you may open the files. On file open, the expiration date in the HDR1 label is checked against the current or actual calendar date to determine whether the associated file has expired. If it has not, data management issues error message DM57, and it is not possible to write to the file. You should mount the correct volume and rerun.

If the file has expired, the tape is positioned so that the old HDR1 label is overwritten. The HDR1 label for the new file, set up from the values you specify in the LBL job control statement, is written on the tape.

E.2.2.2. Second File Header Label (HDR2)

The second file header label (HDR2) acts as an extension of the HDR1 label and is required in standard labeled files. Unless the HDR2 label was created by OS/3, however, as indicated in the *system code* field of the HDR1 label, tape SAM ignores the HDR2 label on input tapes. Figure E—3 shows the format of the HDR2 label; Table E—3 describes its fields.



LEGEND:

Generated by data management or reserved for system expansion.

Written by data management from user-supplied data.

Figure E–3. Second File Header Label (HDR2) Format for an EBCDIC Tape Volume

Table E—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, with length fields specified in decimal F — Fixed-length U — Undefined V — Variable-length, 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
Reserved	36—79	Reserved for future system use

E.2.3. 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.

When you issue an OPEN macroinstruction to an input tape file, with READ=BACK specified in the DTFMT macroinstruction, the OPEN transient checks the fields in an EOF1 and 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 E—4 illustrates the format of the EOF1 or EOV1 label; Table E—4 summarizes the contents of its fields. Similarly, Figure E—5 and Table E—5 describe the EOF2 or EOV2 label.

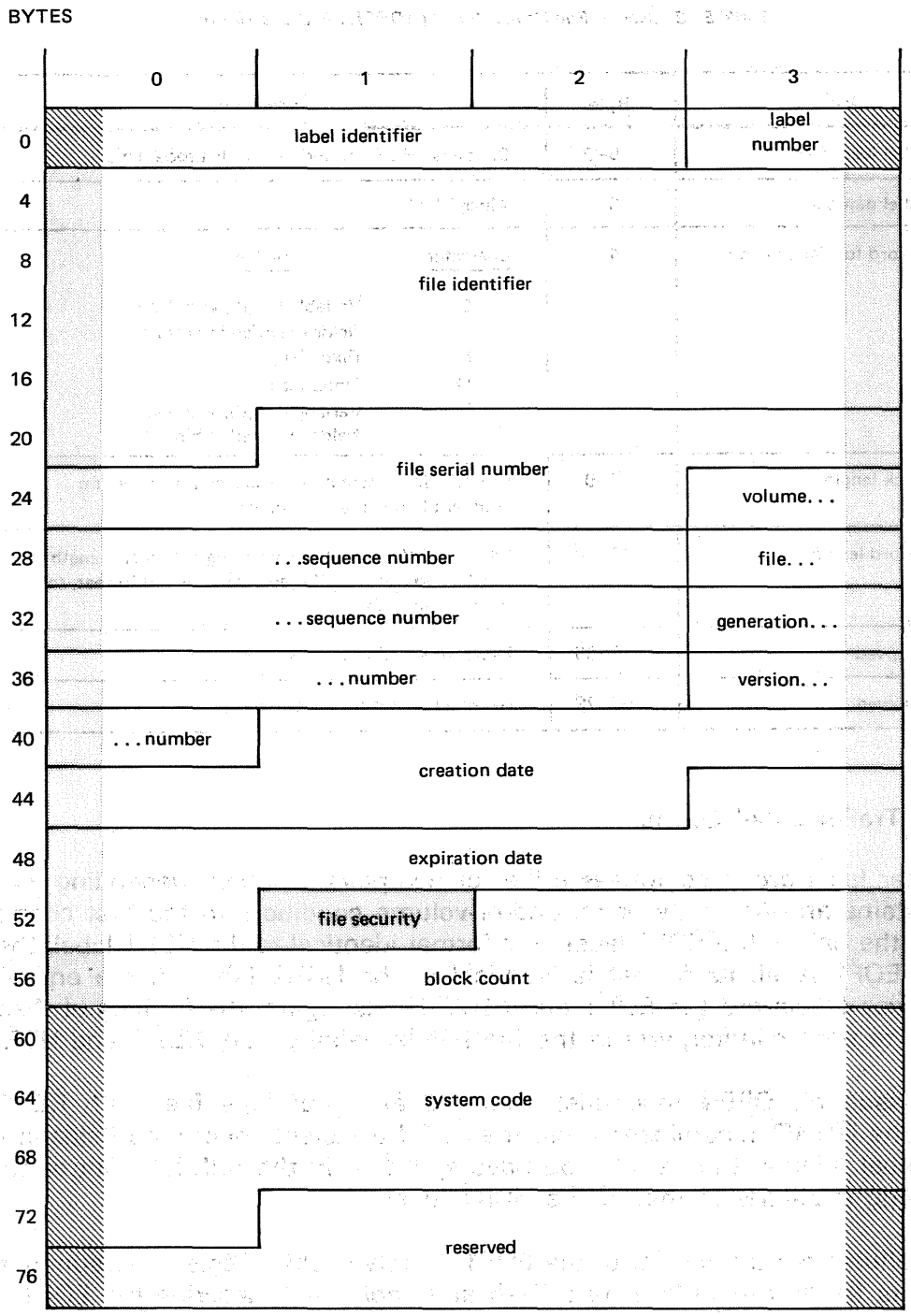
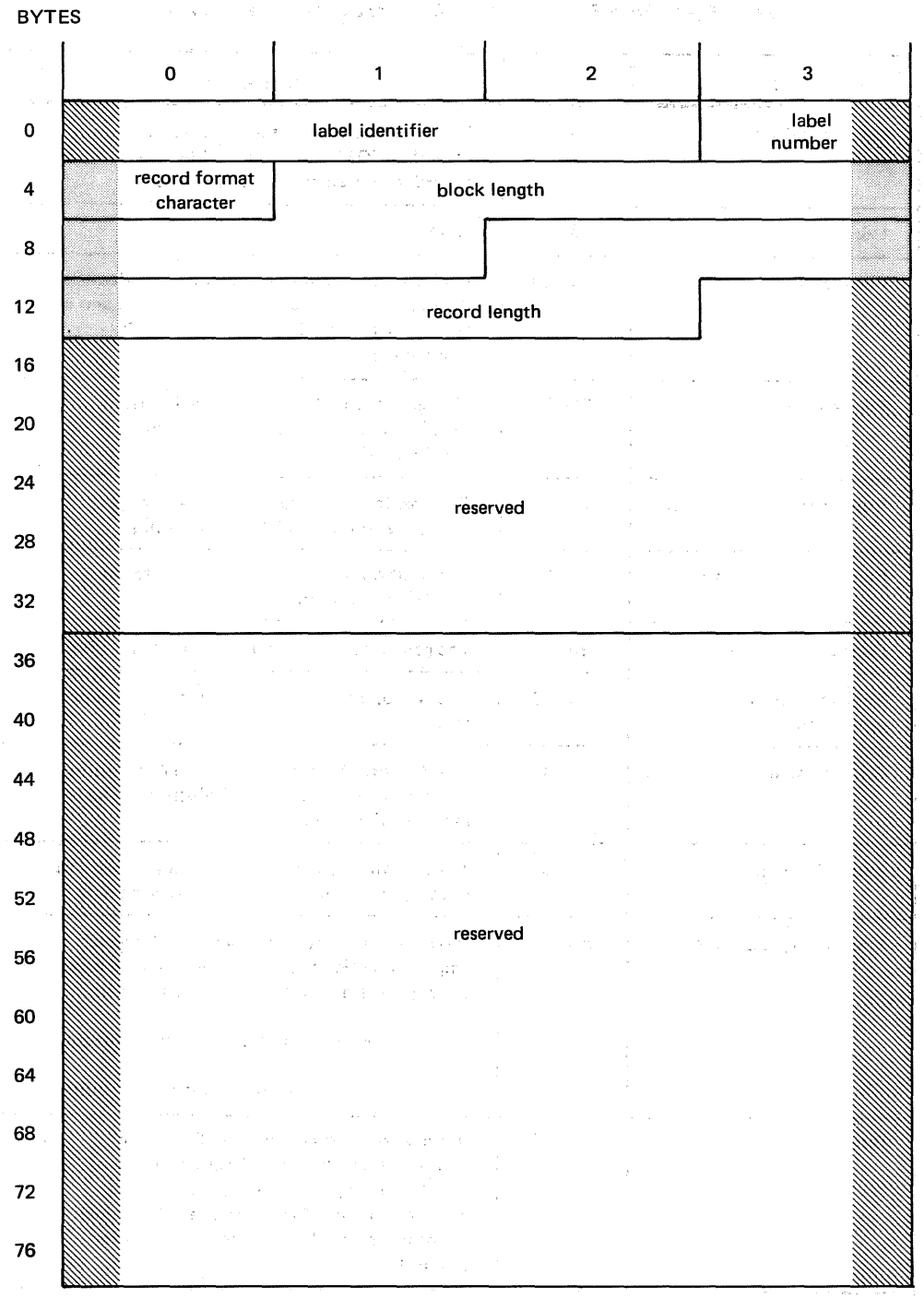


Figure E—4. Tape File EOF1 and EOV1 Label Formats

Table E—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.
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. Tape SAM 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 data management or reserved for system expansion.

Written by data management from user-supplied data.

Figure E—5. Tape File EOF2 and EOV2 Label Formats

Table E—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, with length fields specified in decimal F — Fixed-length U — Undefined V — Variable-length, 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
Reserved	36–79	Reserved for future system use

E.2.4. Standard User Header and Trailer Labels

In a standard labeled file, you may use only the standard UHL and UTL; within the OS/3 tape SAME conventions, their use is entirely optional. You may have one or as many as eight UHL, or none at all; the same applies to optional UTL. If you use them, they must be in the OS/3 standard 80-byte format and content as described in Figure E—6 and Table E—6. Their position in a standard labeled tape volume is as prescribed in Section 8.

When you have specified the block numbering option with the BKNO keyword parameter of your DTFMT declarative macro (9.2.3.5), the system labels and your optional user labels may not be the standard length. Optional user labels in an EBCDIC input file must then be 83 bytes long, and user labels in an ASCII input file 81 bytes long, to ensure correct processing.

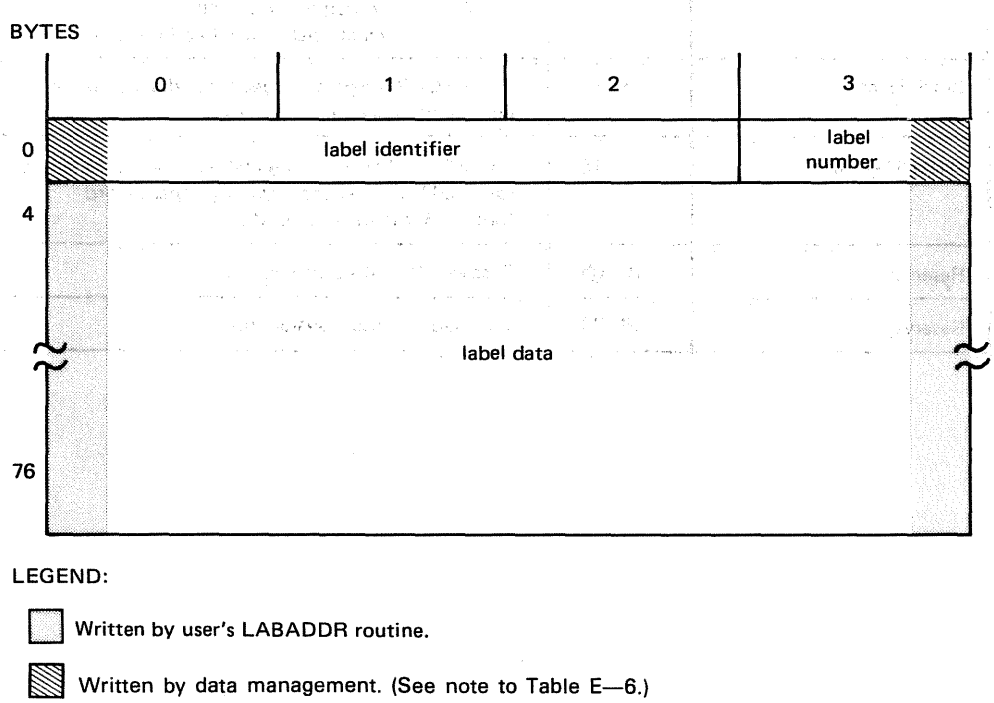


Figure E—6. Optional User Header and Trailer Label Format, ASCII and Standard Labeled EBCDIC Tape Files

Table E—6. Optional User Header and Trailer Labels, Field Description for Standard Labeled Tape Files

Field	Bytes	Description
Label identifier	0—2	Contains "UHL" for user header label; "UTL" for user trailer label
Label number	3	Ranges from 1 through 8 (See note.)
Label data	4—79	Contains 76 bytes of user-specified information

NOTE:

For ASCII files, the label number is not written by data management; it is the user's responsibility. Also there is no limit on the range; the user may have any number of user labels he wants.

E.3. ASCII STANDARD MAGNETIC TAPE LABELS

The figures and tables that follow describe the labels written (and expected to be read) by OS/3 magnetic tape SAM for ASCII files. Note the very small differences for foregoing EBCDIC labels and these ASCII labels, which conform to *American National Standard Magnetic Tape Labels for Information Interchange, X3.27 — 1969.*

OS/3 magnetic tape SAM writes and processes the following ASCII standard labels: VOL1, HDR1, HDR2, EOV1, EOV2, EOF1, and EOF2. Although data management also provides for input and output processing of optional UHLs and UTLs (their forms are identical in ASCII to those described in E.2.4), it does not provide for processing optional user volume labels (UVLs) on output. If present on ASCII input tapes, UVLs are accepted, but bypassed and ignored.

For ASCII record formats and reel organizations, refer to Section 8.

E.3.1. ASCII Character Code and Processing

During input and output processing of ASCII magnetic tape files, OS/3 data management uses the character code specified by *American National Standard Code for Information Interchange, X3.4 — 1968,* performing appropriate translations (EBCDIC to ASCII, or vice versa) so that your program always processes in EBCDIC. Refer to 9.2.7.1 for details on specific or automatic inclusion of the OS/3 ASCII translation table module during link time. Appendix C provides a useful cross-reference table depicting the correspondence between ASCII and EBCDIC.

E.3.1.1. Output Processing of Labels in ASCII Magnetic Tape Files

When you specify ASCII = YES in the DTFMT declarative macro defining an output magnetic tape file, OS/3 data management writes out all system labels in ASCII. Just as you must present your data to data management in EBCDIC (9.2.7.1) so also must you present your optional UHL and UTL label data in EBCDIC. Data management translates these into ASCII before writing them out to tape. It is your responsibility to write out the label number.

E.3.1.2. Input Processing of Labels in ASCII Magnetic Tape Files

When reading input magnetic tape files coded in ASCII, OS/3 data management assumes that these comply fully with *American National Standard X3.27 — 1969,* and that there is no mixture of character codes. Any exception may result in incorrect processing. Before passing your data and your optional UHLs or UTLs to you, data management translates these into the form it expects to receive before ouptut: your program receives data and labels in EBCDIC.

E.3.2. OS/3 Processing of Certain Fields in ASCII Tape Labels

The format and content of ASCII magnetic tape labels in OS/3 are depicted in Figures E—7 through E—11 and Table E—7 through E—11. These subparagraphs describe the way OS/3 processes certain of the label fields; further notes appear in the tables.

E.3.2.1. Accessibility Field

The *accessibility field* occurs in the VOL1, HDR1, EOF1, and EOV1 labels. During read-forward input, a "space" (2/0) encountered in this field of *both* the VOL1 and HDR1 labels allows processing to continue, whereas a nonspace in either field causes the issuance of error message DM12 to the operator's console; processing may not continue unless the operator responds with the override option in effect at your installation. (For backward-read input, the procedure is identical, the fields interrogated being those in the EOV1 and EOF1 labels.)

In OS/3, there is no option to write a nonspace character in the *accessibility field* of any system label on output ASCII tapes; data management always creates this field as "space".

E.3.2.2. Label Standard Level Field

The *label standard level* field occurs only in the VOL1 label. When writing output ASCII tapes, data management writes "1" in this field to indicate that the tape is in full compliance with *American National Standard X3.27 — 1969.* On input tapes, a "1" in this field ensures correct processing; tapes created under later versions of the standard may also be accepted, but you cannot be assured of correct processing.

E.3.2.3. Expiration Date Field

The *expiration date* field occurs in the HDR1, EOF1, and EOV1 labels. If you attempt to write over an unexpired file in an ASCII tape (one whose expiration date is later than today's date), data management issues error message DM57 to the operator console. You will not be able to continue processing unless the operator responds with the override option in effect at your installation.

You should note that, in a multifile ASCII volume, if you give a file a *later* expiration date than you have given to the files that you have already recorded on the tape, this additional protection you intend is not effective. The expiration date of the *first* file on a volume is the latest date on which any part of the volume is protected from being overwritten.

E.3.2.4. System Code

In writing output tapes, data management writes "OS3", left-justified in the 13-byte system code field. The remainder of the field is filled with "spaces". The field is ignored on input.

E.4. PADDING

In other systems, provision has sometimes been made to allow label blocks to be extended in length to the nearest multiple of the computer's word length, any desired characters being used for padding. OS/3 does not provide for padding of labels (nor of data blocks: see Section 8). If the labels in your ASCII input files are not exactly 80 bytes in length, they cannot be processed properly by OS/3. (If BKNO=YES is specified, however, label length should be exactly 83 bytes.)

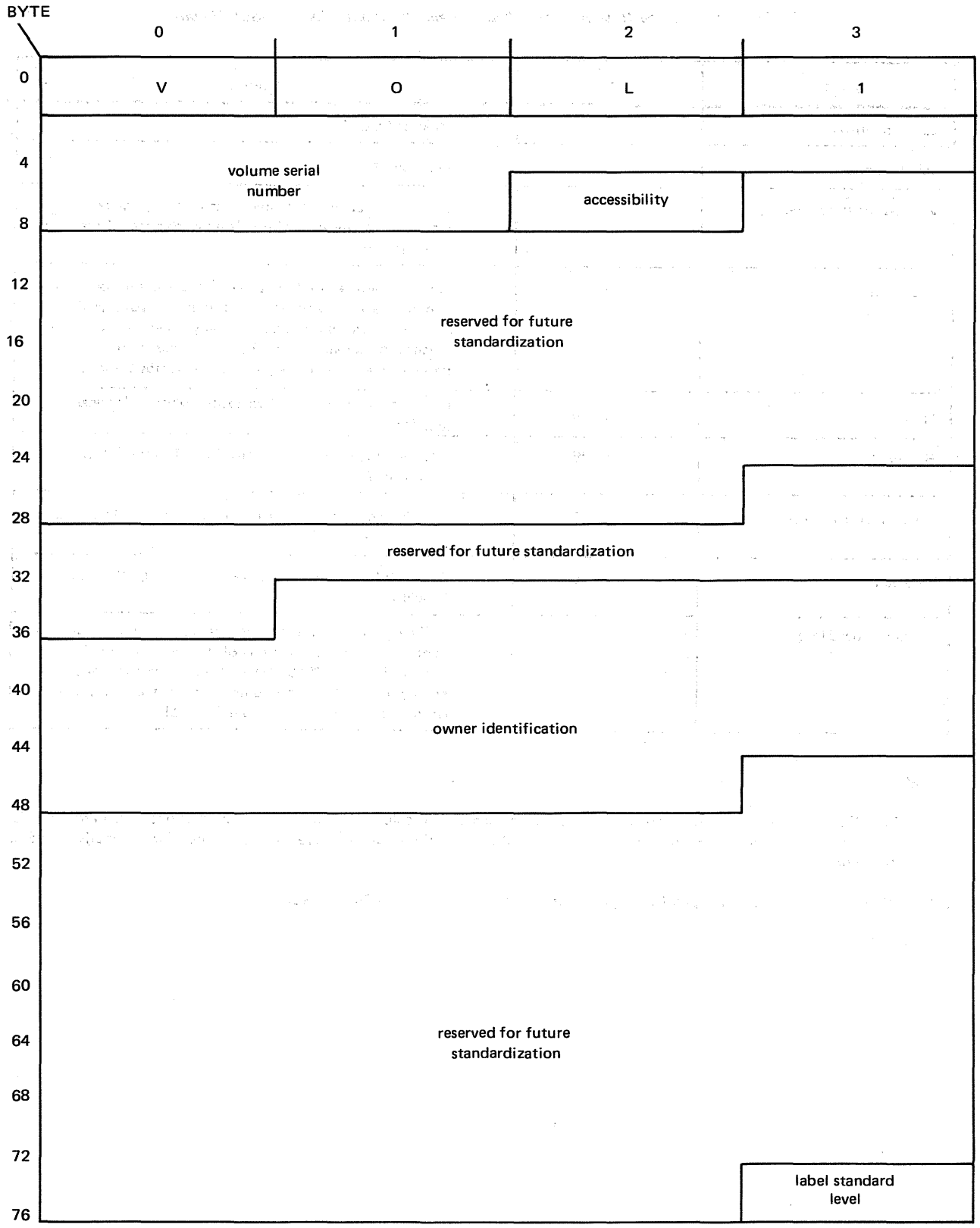


Figure E—7. Volume Header Label (VOL1) for an ASCII Magnetic Tape Volume

Table E—7. Volume Header Label (VOL1), Field Description for an ASCII Volume

Field	Bytes	Description
Label identifier	0–2	Must be "VOL"
Label number	3	Must be "1"
Volume serial number	4–9	Six "a" characters permanently assigned by the owner to identify this physical volume (that is, this reel of magnetic tape). (See Note 1.)
Accessibility	10	An "a" character that indicates any restrictions on who may have access to the information in this volume. A "space" means unlimited access; any other character means special handling, in the manner agreed upon between the interchange parties. (See Notes 1 and 2.)
Reserved	11–30	Reserved for future standardization. Must be "spaces". (See Note 2.)
Reserved	31–36	Reserved for future standardization. Must be "spaces". (See Note 2.)
Owner identification	37–50	Any "a" characters, identifying the owner of the physical volume (See Note 1.)
Reserved	51–78	Reserved for future standardization. Must be "spaces" (See Note 2.)
Label standard level	79	"1" means that the labels and data formats on this volume conform to the requirements of American National Standard X3.27–1969. "Space" means that the labels and data formats on this volume require the agreement of the interchange parties. (See Note 2.)

NOTES:

1. An "a" character is any of the characters occupying the center four columns of ASCII (American National Standard Code for Information Interchange) except position 5/15 and those positions where there is a provision for alternative graphic representation.

2. "Space" is the normally nonprinting graphic character occupying position 2/0 of ASCII.

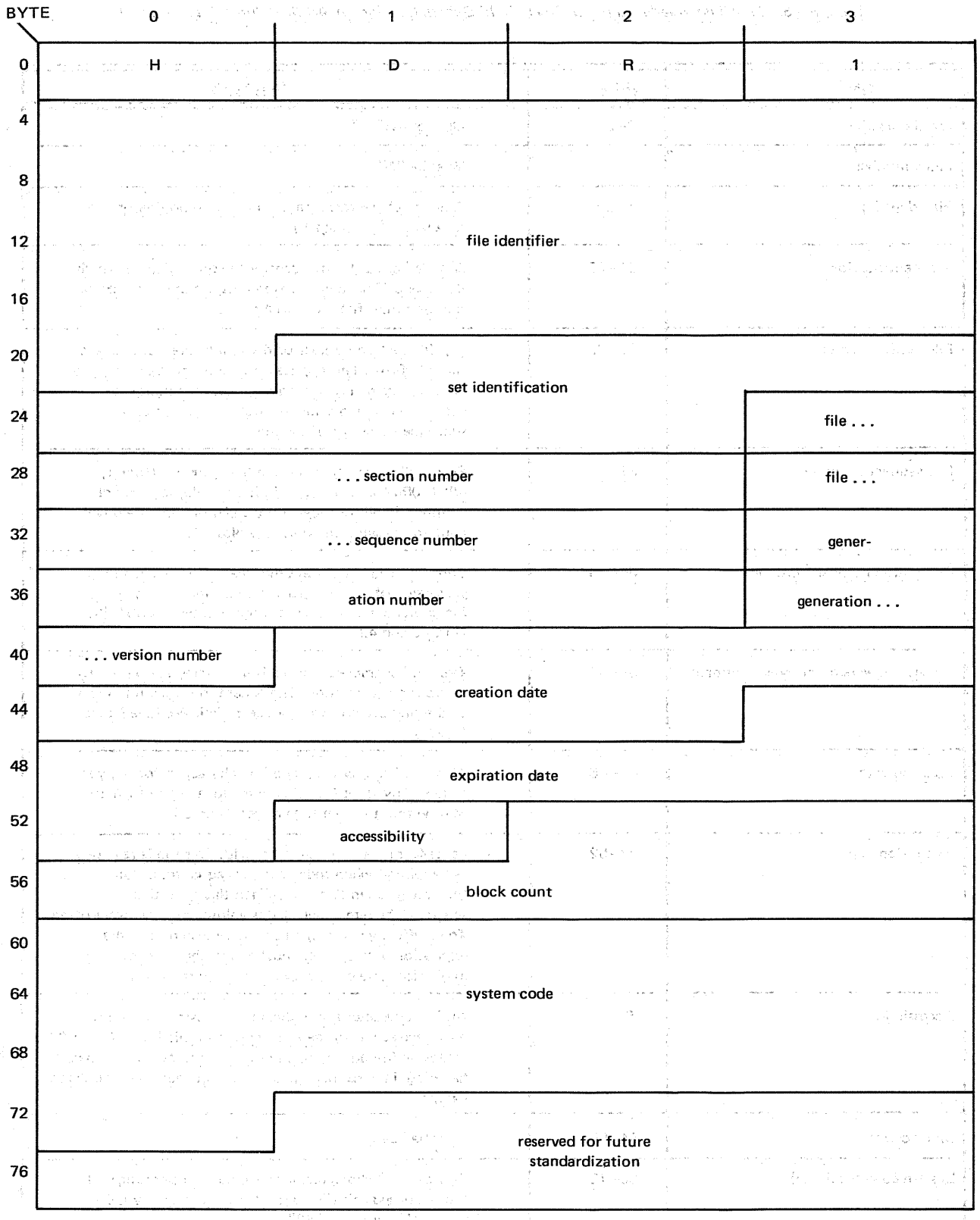


Figure E—8. First File Header Label (HDR1) for an ASCII Magnetic Tape Volume

Table E—8. First File Header Label (HDR1), Field Description for an ASCII Volume (Part 1 of 2)

Field	Bytes	Description
Label identifier	0–2	Must be "HDR"
Label number	3	Must be "1"
File identifier	4–20	Any "a" characters agreed on between originator and recipient . (See Note 1.)
Set identification	21–26	Any "a" characters to identify the set of files of which this is one. This identification must be the same for all files of a multifile set. (See Note 1.)
File section number	27–30	The file section number of the first header label of each file is "0001". This applies both to the first or only file on a volume and to subsequent files on a multifile volume. This field is incremented by one on each subsequent volume of the file.
File sequence number	31–34	Four "n" characters denoting the sequence (that is, 0001, 0002, etc) of files within the volume or set of volumes. In all the labels for a given file, this field will contain the same number. (See Note 3.)
Generation number (optional)	35–38	Four "n" characters denoting the current stage in the succession of one file generation by the next. When a file is first created, its generation number is 0001. (See Notes 3 and 4.)
Generation version number (optional)	39–40	Two "n" characters distinguishing successive iterations of the same generation. The generation version number of the first attempt to produce a file is 00. (See Notes 3 and 4.)
Creation date	41–46	A "space" followed by two "n" characters for the year, followed by three "n" characters for the day (001 to 366) within the year (See Notes 2 and 3.)
Expiration date	47–52	Same format as creation date field. This file is regarded as "expired" when today's date is equal to, or later than, the date given in this field. When this condition is satisfied, the remainder of this volume may be overwritten. To be effective on multifile volumes, therefore, the expiration date of a file must be less than, or equal to, the expiration date of all previous files on the volume.
Accessibility	53	An "a" character that indicates any restrictions on who may have access to the information in this file. A "space" means unlimited access; any other character means special handling, in a manner agreed upon between the interchange parties.
Block count	54–59	Must be "zeros"
System code (optional)	60–72	Thirteen "a" characters identifying the operating system that recorded this file. Output tapes written by OS/3 tape SAM contain "OS3".
Reserved	73–79	Reserved for future standardization; must contain "spaces"

NOTES:

1. An "a" character is any of the characters occupying the center four columns of ASCII (depicted in *American National Standard X3.4—1968*), except for position 5/15 and those positions where there is provision for alternative graphic representation.

Table E—8. First File Header Label (HDR1), Field Description for an ASCII Volume (Part 2 of 2)

2. A "space" is the normally nonprinting graphic character occupying position 2/0 in ASCII.

3. An "n" character is any ASCII numeric digit, from 0 through 9.

4. "Optional," when used to describe a field in these ASCII labels, means that the field may, but need not, contain the information described. If an optional field does not contain the designated information, it must contain "spaces".

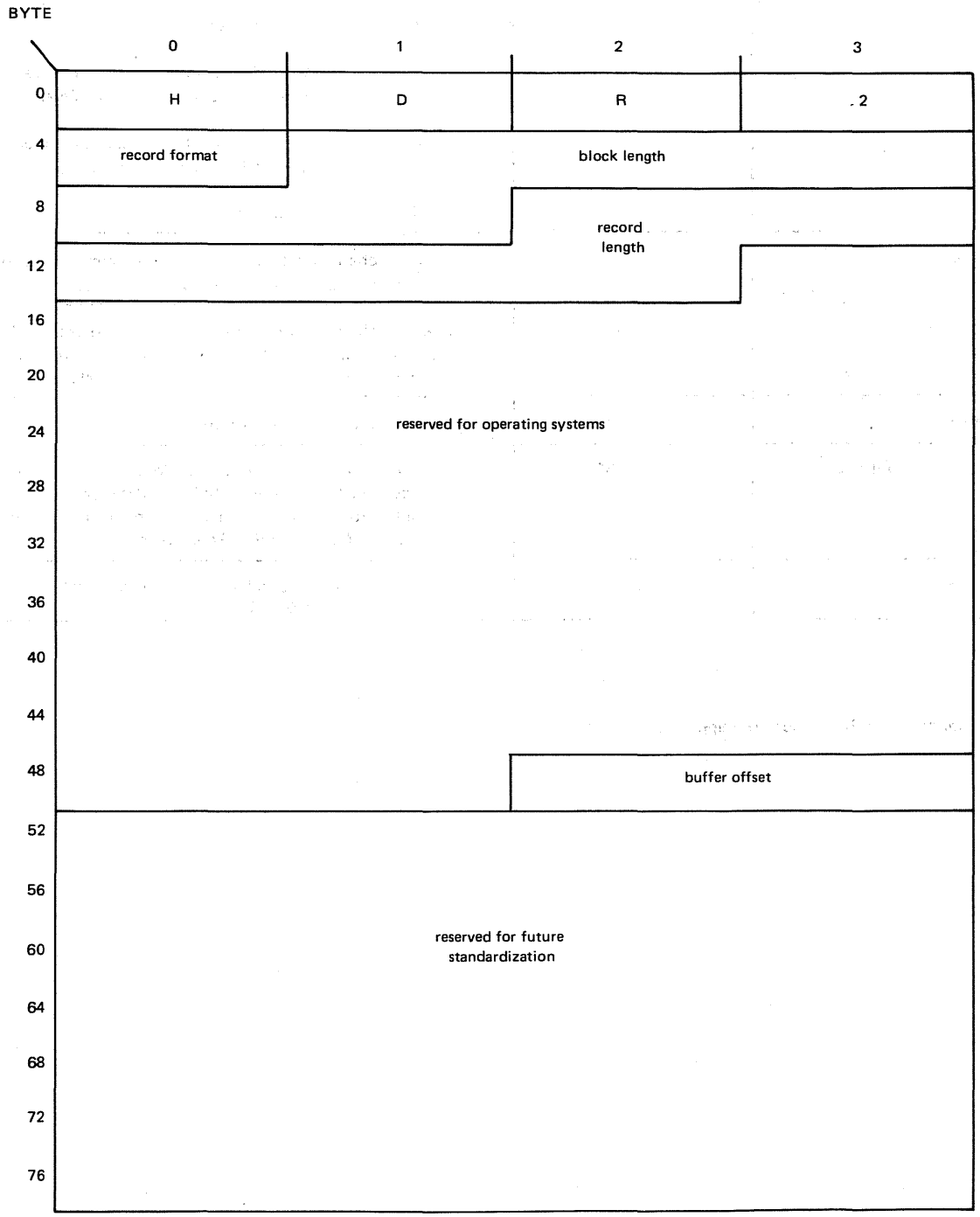


Figure E—9. Second File Header Label (HDR2) for an ASCII Volume

Table E—9. Second File Header Label (HDR2), Field Description for an ASCII Volume

Field	Bytes	Description
Label identifier	0—2	Must be "HDR"
Label number	3	Must be "2"
Record format	4	Character — Meaning F — Fixed length D — Variable, with the number of characters in the record specified in decimal V — Variable, with the number of characters specified in binary. (See Note.) U — Undefined
Block length	5—9	Five "n" characters specifying the maximum number of characters per block. (See Note 3, Table E—8.)
Record length	10—14	Five "n" characters specifying: if "record format" is F, record length; if D or V, maximum record length including any count fields; if U, then undefined. (See Note.)
Reserved	15—49	Reserved for OS/3 use; currently "spaces"
Buffer offset (optional)	50—51	Two "n" characters specifying the length in characters of any additional field inserted before a data block—for example, block length. This length is included in the block length. (See Notes 3 and 4, Table E—8.)
Reserved	52—79	Reserved for future standardization; must contain "spaces". (See Note 2, Table E—8.)

NOTE:

OS/3 magnetic tape SAM does not support the ASCII "V-format" record.

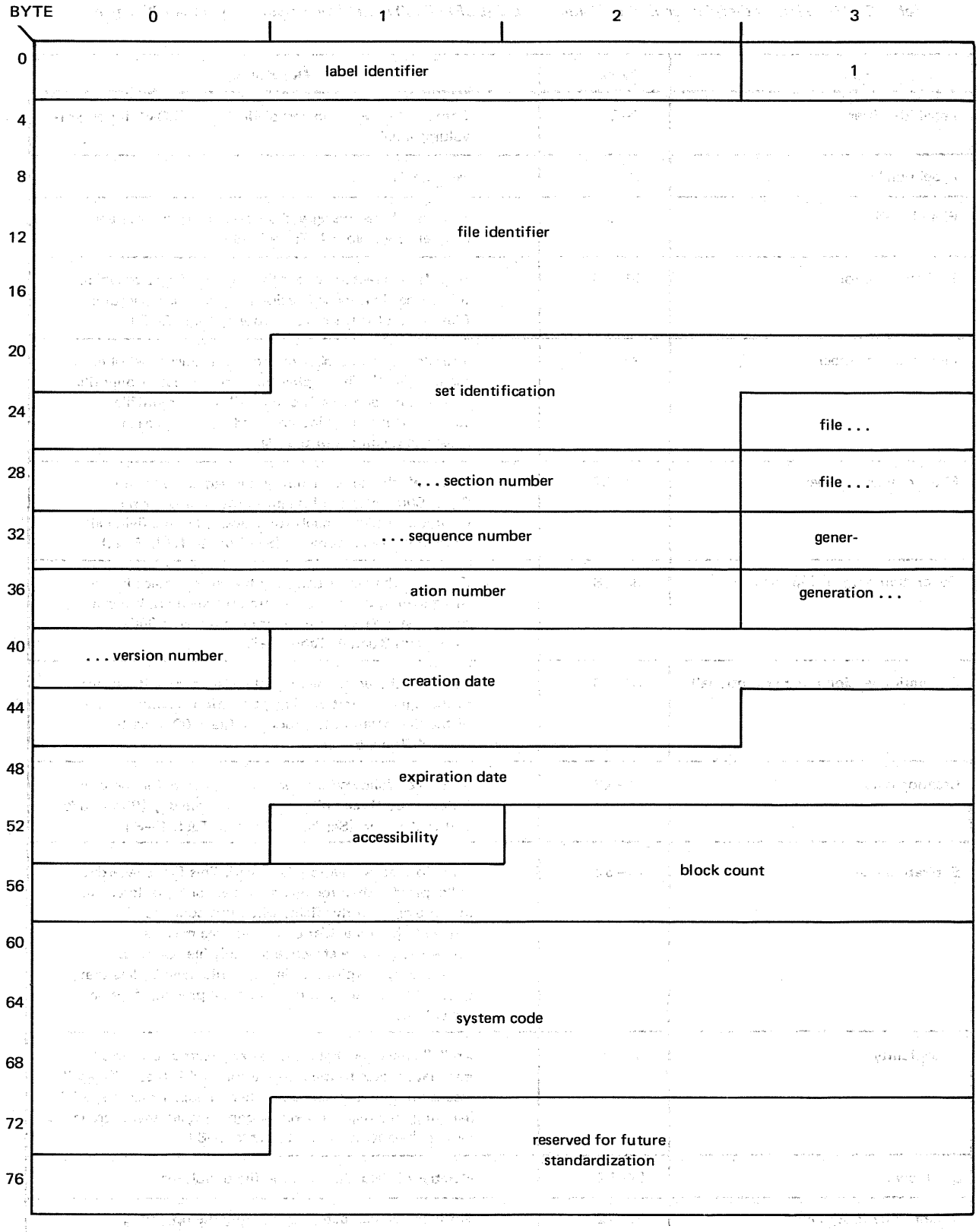


Figure E—10. First End-of-File or End-of-Volume Label (EOF1/EOV1) for an ASCII Volume

Table E—10. First End-of-File or End-of-Volume Label (EOF1/EOV1), Field Description for an ASCII Volume

Field	Bytes	Description
Label identifier	0—2	Contains "EOF" if an end-of-file label; "EOV" for end-of-volume label
Label number	3	Must be "1"
File identifier	4—20	Any "a" characters agreed on between originator and recipient (See Note 1, Table E—8.)
Set identification	21—26	Any "a" characters to identify the set of files of which this is one. This identification must be the same for all files of a multifile set. (See Note 1, Table E—8.)
File section number	27—30	The file section number of the first header label of each file is "0001". This applies both to the first or only file on a volume and to subsequent files on a multifile volume. This field is incremented by one on each subsequent volume of the file.
File sequence number	31—34	Four "n" characters denoting the sequence (that is, 0001, 0002, etc) of files within the volume or set of volumes. In all the labels for a given file, this field will contain the same number. (See Note 3, Table E—8.)
Generation number (optional)	35—38	Four "n" characters denoting the current stage in the succession of one file generation by the next. When a file is first created, its generation number is 0001. (See Notes 3 and 4, Table E—8.)
Generation version number (optional)	39—40	Two "n" characters distinguishing successive iterations of the same generation. The generation version number of the first attempt to produce a file is 00. (See Notes 3 and 4, Table E—8.)
Creation date	41—46	A "space" followed by two "n" characters for the year, followed by three "n" characters for the day (001 to 366) within the year. (See Notes 2 and 3, Table E—8.)
Expiration date	47—52	Same format as creation date field. This file is regarded as "expired" when today's date is equal to, or later than, the date given in this field. When this condition is satisfied, the remainder of this volume may be overwritten. To be effective on multifile volumes, therefore, the expiration date of a file must be less than, or equal to, the expiration date of all previous files on the volume.
Accessibility	53	An "a" character that indicates any restrictions on who may have access to the information in this file. A "space" means unlimited access; any other character means special handling, in a manner agreed upon between the interchange parties. (See Notes 1 and 2, Table E—8.)
Block count	54—59	Number of data blocks in the file or volume
System code (optional)	60—72	Thirteen "a" characters identifying the operating system that recorded this file. Output tapes written by OS/3 tape SAM contain "OS3". (See Note 1, Table E—8.)
Reserved	73—79	Reserved for future standardization; must be "spaces". (See Note 3, Table E—8.)

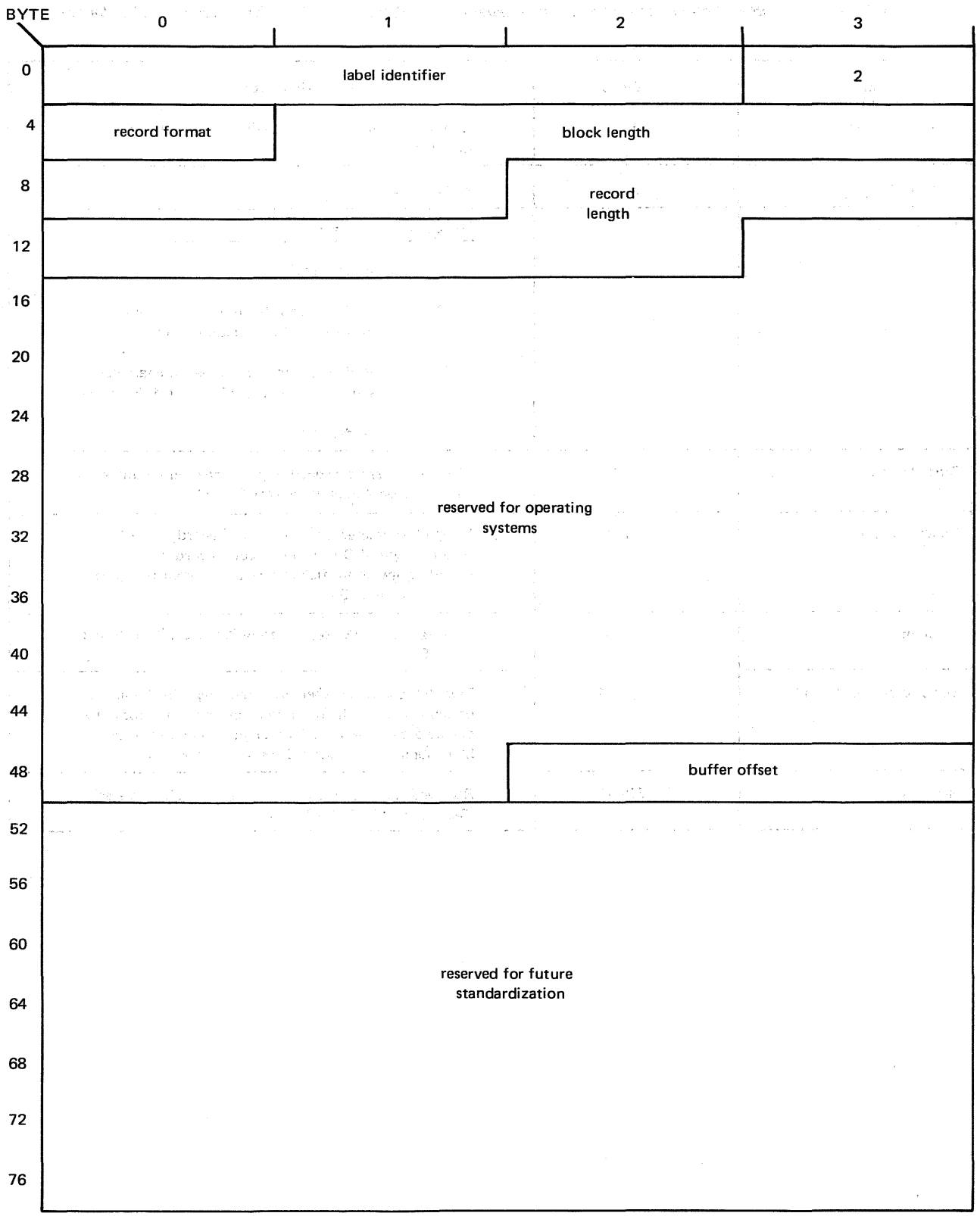


Figure E—11. Second End-of-File or End-of-Volume Label (EOF2/EOV2) for an ASCII Volume

Table E—11. Second End-of-File or End-of-Volume Label (EOF2/EOV2), Field Description for an ASCII Volume

Field	Bytes	Description
Label identifier	0–2	Contains "EOF" if an end-of-file label; "EOV" for end-of-volume
Label number	3	Must be "2"
Record format	4	Character — Meaning F — Fixed length D — Variable, with the number of characters in the record specified in decimal V — Variable, with the number of characters specified in binary. (See Note 1, Table E—9.) U — Undefined
Block length	5–9	Five "n" characters specifying the maximum number of characters per block. (See Note 3, Table E—8.)
Record length	10–14	Five "n" characters specifying: if "record format" is F, record length; if D or V, maximum record length including any count fields; if U, then undefined. (See Note 1, Table E—9.)
Reserved	15–49	Reserved for OS/3 use; currently "spaces". (See Note 2, Table E—8.)
Buffer offset (optional)	50–51	Two "n" characters specifying the length in characters of any additional field inserted before a data block—for example, block length. This length is included in the block length. (See Notes 3 and 4, Table E—8.)
Reserved	52–79	Reserved for future standardization; must be "spaces". (See Note 2, Table E—8.)

Appendix F. Consolidated Data Management Migration Considerations

F.1. WHAT DO I HAVE TO DO TO MIGRATE TO CONSOLIDATED DATA MANAGEMENT?

The answer is that the amount of effort varies with language of the program you want to migrate: that is, BAL, RPG II, 1968 American National Standard COBOL, 1974 American National Standard COBOL, or FORTRAN.

F.2. MIGRATION REQUIREMENTS

The migration requirements for programs written in the various programming languages are discussed in the paragraphs that follow.

F.2.1. BAL Programs

If your program is written in BAL, you must redefine your files. This means that you must replace all basic data management DTF declarative macroinstructions with the consolidated data management CDIB and RIB declarative macroinstructions.

If you use disk files with your program, these must be converted (using data utilities) to MIRAM files before they can be used.

You must replace all basic data management imperative macroinstructions with the appropriate consolidated data management imperative macroinstructions. These new imperative macroinstructions must be immediately followed by inline coding that checks the status of the operation being performed. This is necessary because consolidated data management always returns control inline. There is no provision made for contingency routine addresses such as EOFADDR and ERROR in the RIB macroinstruction.

After you have modified your program, it must be reassembled and relinked before it can be executed.

Note that if your program creates disk files, these files must be specified as MIRAM files in the job control stream used to execute the program.

F.2.1.1. OS/3 Sequential DTF Mode To CDI Macro Converter (DTFCDI301)

You can reduce the amount of effort required to migrate a basic data management BAL program that processes sequential files (card, tape, printer, or SAM disk) to consolidated data management by using DTFCDI301. This converter processes basic data management BAL source program modules that you have previously written to a library file. It produces a consolidated data management source program module and a listing that shows the actions taken by the converter. The consolidated data management source program module is automatically written to the original library file unless you specify otherwise. Those statements that cannot be converted or that require your attention are indicated by diagnostic messages on the listing. As you can see, the converter can save you considerable time because you only have to change your program where indicated.

After you have modified your program, it must be reassembled and relinked before it can be executed.

The DTFCDI301 user guide, UA-0455 (current version) contains the detailed information you need to successfully use the converter.

F.2.2. RPG II Programs

An RPG II program does not require any modifications unless you want to want to use workstations. It must, however, be recompiled and relinked before you can execute it even if you did not modify it.

If your program uses disk files, these must be converted (using data utilities) to MIRAM files.

Note that if your program creates disk files, they must be specified as MIRAM files in the job control stream used to execute the program.

F.2.3. 1968 American National Standard COBOL Programs

Programs that are written in this language cannot be migrated to consolidated data management. They must first be converted to 1974 American National Standard COBOL.

F.2.4. 1974 American National Standard COBOL Programs

A program written in this language does not require any modification unless it uses disk files. It must, however, be recompiled and relinked before you execute it even if you did not modify it.

If your program uses disk files, these must be converted (using data utilities) to MIRAM files before you can use them.

Note that if your program creates disk files, they must be specified as MIRAM files in the job control stream used to execute the program.

F.2.5. FORTRAN Programs

Your FORTRAN program does not have to be recompiled; however, you must update your unit definitions, reassemble them, and relink them with your program before you can execute it.

If you want to use workstations, you must modify your FORTRAN source program and recompile it. You must also update your unit definitions and include unit definitions for the workstations. These unit definitions must be reassembled and relinked with your program before you can execute it.

If your program uses disk files, these must be converted (using data utilities) to MIRAM files.

Note that if your program creates disk files, they must be specified as MIRAM files in the job control stream used to execute the program.

E

M

O

P

R

S

T

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:

Cut along line.

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

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