GC33-5372-2
File No. S370-30

Systems

DOS/VS Data Management Guide

Release 29

IBM

GC33-5372-2
File No. S370-30

Systems

DOS/VS Data Management Guide

Release 29

IBM

Summary of Amendments

This edition documents

- Virtual Storage enhancements and
- Support of the following new devices:

System/370 Model 115
3203 and 5203 Printers
3340 Disk Storage
3420 Tape Unit
3540 Diskette I/O Unit
3780 Data Communication Terminal
5425 Multifunction Card Unit

In addition, minor technical changes and editorial corrections have been made throughout the book.

Changes in content are indicated by a vertical bar to the left of the change.

Third Edition (Nov. 1973)

This edition applies to Version 5 of the IBM Disk Operating System/Virtual Storage DOS/VS, and to all subsequent versions and releases until otherwise indicated in new editions or Technical Newsletters. Changes are continually made to the information herein. Before using this publication in connection with the operation of IBM systems, consult the latest IBM System/360 and System/370 Bibliography, GA22-6822, for the editions that are applicable and current.

This is a major revision of, and obsoletes, GC33-5372-1.

Note: For the availability dates of features and programming support described in this manual, please contact your IBM representative or the IBM branch office serving your locality. Similarly, requests for copies of IBM publications should be made to your IBM representative or to your local IBM branch office.

If the functions listed below are available as a separate component and your system does not include this component, the information about these functions is included for planning purposes only. The functions are:

- Chained RPL support
- Concurrent request processing
- Overlapping I/O operations by means of the EXCPAD exit.

A form for readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Laboratory, Publications Department, 703 Boeblingen, Schoenaicher Strasse 220, Postfach 210, Germany. Comments become the property of IBM.

PREFACE

This book is a guide to data management as it is provided by Version 5 of the IBM Disk Operating System: DOS/VS. All of the data management facilities provided are discussed on conceptual and functional levels.

Data management refers to the way data is organized and accessed. The book describes how DOS/VS accomplishes this and, in addition, provides information for choosing data management strategies for different types of data files and processing requirements.

This book is not a guide to the operating system; instead, a separate manual is provided for this purpose: the *DOS/VS System Management Guide*, GC33-5371. A general overview of the system is presented in the *Introduction to DOS/VS*, GC33-5370.

After reading the above mentioned manuals, you should be able to turn directly to the DOS/VS library of reference manuals in order to work with your system. From these manuals you can easily retrieve details about the formats of the control statements, macros, labels, and messages, which you must deal with from day to day.

About this manual ....

The organization of this manual reflects four levels of information; these are presented in four subsequent sections:

1. Concepts of data management.

   The first section introduces data management without regard to any specific operating system. It is mainly addressed to readers newly entering the world of data processing.

2. DOS/VS data management facilities.

   The second section describes all of the components that make up the DOS/VS data management programming system.

3. Data management support by DOS/VS.

   This section focusses on the access methods provided under DOS/VS, and the various techniques a programmer can use in managing data files.

4. Access methods.

   The last section describes the four access methods in detail. It acts as an introduction to the publication *DOS/VS Supervisor and I/O Macros*, GC33-5373. After reading this section, the reader should be familiar with the access methods, and be able to implement the facilities offered in his problem programs, using the referenced manual to find the detailed formats of the macros.

In addition to those sections, a set of appendixes is included. A glossary containing terms and their definitions, as they are used in this manual, and an index, are present at the end.

TABLE OF CONTENTS

Section 1: CONCEPTS OF DATA MANAGEMENT

This first section introduces the basic principles of data management. It is addressed to users newly entering the world of data processing who may need a general overview of the subject. For more experienced users, reading this section may help to refresh their memories.

The following items are introduced:

- the functions of data management
- the meaning of the term 'information'
- data fields, records, and files
- data representations
- data organization
- attributes of storage devices
- record blocking
- file labeling
- input/output control
- data structures

These are explained in very general terms, without regard to any specific operating system. The same items will be discussed in the following sections, in regard to DOS/VS. The last chapter in this section is a list of references to other sections in this manual.

Introduction

Data management is the control, storage, and retrieval of information to be processed by a computer. Each of these three areas of data management is an essential function of any information system.

Control is the internal supervision of the data management process. It establishes the user's right to access or modify the information in the system. And it monitors the location of information, insuring data integrity (no data loss), and insuring that the information in the system is current.

Storage is the technique for representing the information on a storage device such as disk, tape, or punched card. It includes the order in which the information is stored, the way it may be accessed or addressed, and the method of representing the data itself.

Retrieval is the process of locating, formatting, and sequencing information for the user of the system. Locating information means determining what data is required and where it may be found. If this data is not in a usable form, it must be re-structured and perhaps ordered in a different sequence, to meet the needs of the user.

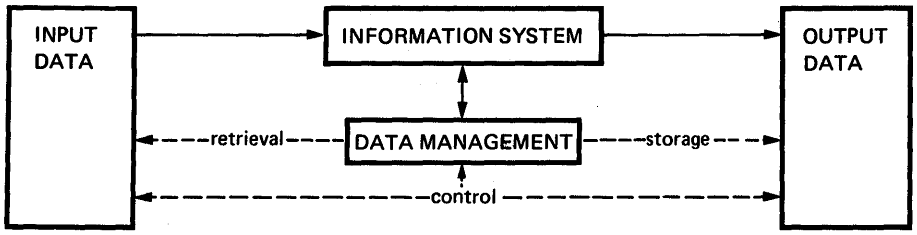


Figure 1.1 Concepts of data management
Data management handles three essential functions of an information system: control, retrieval, and storage of information.

Information - what it means

Information includes facts about entities such as things, people, and machines. We record information about entities for use by the data processing system. For example, as an entity, a person could be described as follows:

> "John Jones is an accountant who works for the ABC Company in department 5A. He is 40 years old, and he is married. His salary is $250 per week. He lives at 801 Main Street, Grovers Corners, N.Y. His social security number is 999-99-9999."

Much of this information is implicit and we interpret it by reason of our experience. An attempt to make the information explicit results in the following:

> "*John Jones* is the NAME OF AN EMPLOYEE whose OCCUPATION is *Accountant.* He works for EMPLOYER *ABC Company* in DEPARTMENT *5A.* His AGE is *40* years, his MARITAL STATUS is *married.* His SALARY is *$250/wk.* His ADDRESS is *801 Main Street, Grovers Corners, N.Y.*. His SOCIAL SECURITY number is *999-99-9999.*"

This example shows that information about entities is composed of at least two parts: the CONTEXT and the *data*. The context is the same for all similar entities, the data may vary for each entity.

In addition to context and data, it is necessary to know how the data is represented. We must know, for example, that the data in the field NAME of AN EMPLOYEE is represented as follows: blanks separate the first name from the last and that the first name appears first. Information about entities therefore consists of three parts:

INFORMATION = CONTEXT + DATA + REPRESENTATION

In conventional data processing the data is stored separately from the context and representation. The reason is, as was shown, that both context and representation are common to all like entities. The data is stored on devices, such as tape, disk, or cards, and the context and representation are implemented as part of the computer program that processes the data.

It is the function of data management to provide organization schemes for data to make data available within a program so that it may be properly related to its context and finally displayed in a meaningful representation. Figure 1.2 shows how the data and its context and representation are related to one another.

'KNOWN' BY THE APPLICATION PROGRAM		STORED ON EXTERNAL DEVICE
CONTEXT	REPRESENTATION	DATA
NAME	20 Alpha characters maximum First name first	John Jones
S.S. #	11 Alpha characters	999-99-9999
DEPT. #	3 Alpha characters maximum	5A
SALARY	5 Decimal digits, assumed decimal point between second and third digit from the right (example = $ 250.00)	250.00

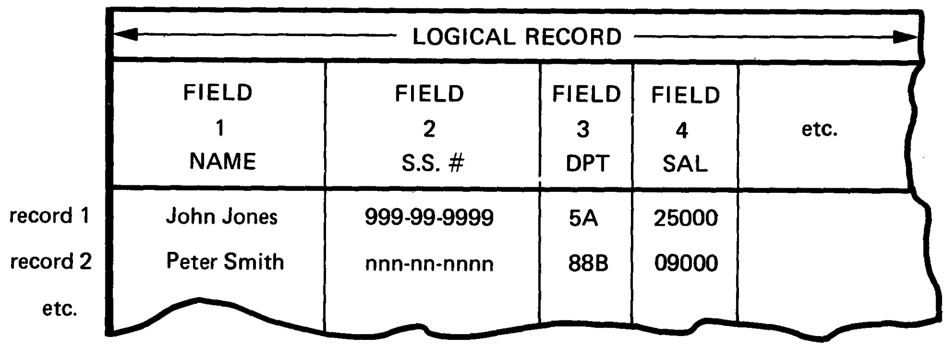
Figure 1.2 Information = Context + Representation + Data
The program that uses data must specify its context and representation in order to have it interpreted as meaningful information.

Each entity has certain characteristics that define the context (for the entity 'Person' they are: Name, Address, Employer, Marital Status, etc.). A user specifies the context of the information he will use by listing the characteristics for which corresponding data values are recorded. Such a list of characteristics is called the *description of a logical record*. It is the description of a logical record which gives the data values a meaning: without specifying both the context and the representation, data has no meaning to anyone.

Fields and Records

A field is the smallest unit of information of interest to a user. A field has a meaning and a name (context), a value (data), and a representation. Examples of fields were shown in Figure 1.2; a collection of fields relating to the same entity is a logical record, which is shown in Figure 1.3

The collection of logical records that is presented to a user is called a *file*.



	LOGICAL RECORD				
	FIELD 1 NAME	FIELD 2 S.S. #	FIELD 3 DPT	FIELD 4 SAL	etc.
record 1	John Jones	999-99-9999	5A	25000	
record 2	Peter Smith	nnn-nn-nnnn	88B	09000	
etc.					

Figure 1.3 Fields and records
The program knows the context of each field of data represented on a record.

In a computer system, there are several ways of representing data both internally (main storage) and externally (tape, disk, punched card, etc.). The following types of representation lend themselves to processing by System/370:

- Arithmetic: represents numeric data.
  - decimal fixed point
  - decimal floating point
  - binary fixed point
  - binary floating point

- String: represents a sequence of characters or binary digits.
  - bit string (string of binary digits)
  - character string (string of alphameric characters)

In general, data to be used in calculations is represented arithmetically, whereas other data is represented by strings.

Later in this chapter, more will be said about data representations.

Data Encoding / Data Compression

In addition to the types listed above, data can be represented within character and bit strings in encoded or compressed form.

Encoding means translating data from one form of representation to another, and is used to conserve either external or internal storage, or to allow easier processing.

Compression means condensing repetitive information. Like encoding, it is also used to conserve storage space.

Examples of encoding and compression of data are shown in Figure 1.4 and Figure 1.5

CHARACTERISTIC	POSSIBLE DATA	MAX. FIELD SIZE	TRANSLATION	ENCODED RESULT	RESULT LENGTH
Language	GERMAN ENGLISH FRENCH SPANISH	7 char.	GERMAN $\rightarrow$ G ENGLISH $\rightarrow$ E FRENCH $\rightarrow$ F SPANISH $\rightarrow$ S	G E F S	1 char.
Color	RED BLUE ORANGE YELLOW GRAY	6 char.	RED $\rightarrow$ 001 BLUE $\rightarrow$ 010 ORANGE $\rightarrow$ 011 YELLOW $\rightarrow$ 100 GRAY $\rightarrow$ 101	001 010 011 100 101	3 bits

Figure 1.4 Examples of data encoding
Encoding saves space by using symbols to represent data.

POSSIBLE DATA	LENGTH	COMPRESSION ALGORITHM	COMPRESSED RESULT	RESULT LENGTH
. . .JOHN . . . . .JONES	25 char.	Remove blanks and replace by a count number of blanks removed.	3JOHN9JONES4	12 char.
0015700 0016300 0019560 0187370	7 char.	Remove first and last digit since neither changes (always 0).	01570 01630 01956 18737	5 char.

Figure 1.5 Examples of data compression
Compression saves space by leaving out unnecessary data.

Note: in the first example, blanks are represented by a period (.).

Virtual Data

Another type of data that can be useful in information systems is *virtual data.* Virtual data, unlike all other types of data mentioned so far, does not really exist on a storage device, but is calculated from other values present. For example, when the fields PRICE-PER-UNIT and AMOUNT-OF-UNITS are stored in a logical record, then TOTAL-VALUE is virtual data, since it may be calculated by multiplying the contents of the fields PRICE-PER-UNIT and AMOUNT-OF-UNITS.

Record Design

When designing logical records it is important to consider all of the data representations mentioned in the preceding text. The following is a list of considerations to bear in mind when choosing which type of data representation to use.

1. How meaningful is the information in this field?

Since a logical record is designed for a user of the information system, only those fields that have a meaning for the application should be included. It should also be noted, however, that the addition of a field to records in an existing file may require a reorganization of the entire file. It is therefore usually advisable to incorporate an amount of 'empty' space in a record for later addition of new fields.

2. How will the field be used?

The data representation of a field should be consistent with the predominant usage of that field. For example, if a field is used 80% of the time for arithmetic calculations and 20% of the time for display on a report, it should be stored in arithmetic form (decimal or binary, fixed-point or floating-point).

3. Are there language restrictions on a particular data representation?

Not all data representations are supported in all high-level programming langauges (COBOL, FORTRAN, PL/I, and RPG). Thus, the data representation should be chosen from those available in the programming language that will be used for the particular application.

4. Is it necessary to be machine independent?

Certain data representations (fixed-decimal, fixed-binary) when implemented for a particular computer system, may make the data 'machine-dependent', that is, this data cannot be processed by a computer of different design. Character string data is the most compatible data representation. In data interchange between two computer systems the data is generally regarded as character string data, any other type of data representation normally requiring prior agreement between the interchange parties.

5. Is the field easy to process?

The data in a field should be in its most usable form. For example, a field SEX can be encoded *1* for *male* and *2* for *female*. To display this information on a report would probably require translation (*1* to *male* and *2* to *female*). If SEX were encoded *M* for *male* and *F* for *female* translation is not required.

6. How will the data representation of a field affect the overall efficiency of the process?

Certain data types in combination with encoding and compression conserve internal and external storage, while others save processing time. These tradeoffs should be considered in the design of a field.

7. Will changes in the data values of a field affect the representation?

In any encoding or compression scheme, prior knowledge of all possible data values is required. Later changes in values may require extensive changes in programs.

Fixed-Length Records and Variable-Length Records

A logical record is a collection of information fields relating to the same entity. The user specifies the size of each field to hold the largest item of data that may be encountered. The sum of the lengths of all fields in a logical record is the length of that logical record.

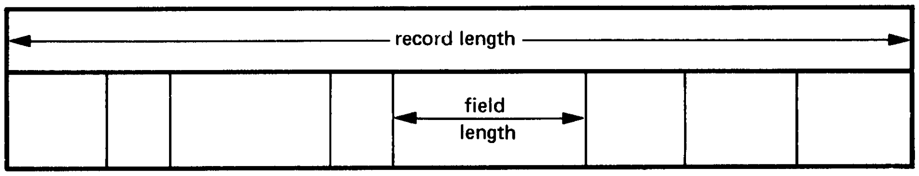


Figure 1.6 Field length and record length
Fields and records may be of fixed or variable length, depending on the application.

A fixed-length logical record is a record in which all fields are of unchanging length and in which the location of all fields is defined for each record within the file. Figure 1.3 is an example of a file containing fixed-length logical records. The fixed-length format is the most commonly used format in conventional data processing, because it is easy to process and control. Not all representations of data, however, conform to this format.

A record becomes a variable-length record if:

- it contains one or more variable-length fields
- it contains a variable number of fixed-length fields.

Variable-Length Fields

For applications that use very large data files it may be important to save external storage in order to reduce the total size of the files. One way to do so is to make use of variable-length data. An example of data that might be variable in length is a set of names, as is shown in Figure 1.7.

Fixed-length design	Fixed size	Variable-length design	Variable size
John Jones	20	John Jones	10
Stanley R. Kowalski	20	Stanley R. Kowalski	20

Figure 1.7. Variable-length fields
Using variable-length fields reduces the size of a file.

Variable Number of Fixed-Length Fields

To illustrate this case, assume that a factory keeps records of its production process by customer-order. Each record in the file describes the product that has to be made as well as the plant activities required to make the product. The number of activities to be performed will differ for each type of product. The records will be designed so that they contain a fixed part at the beginning and a variable part at the end, as is shown in Figure 1.8.

Fixed part		Variable part				
Customer order no.	Description ordered product	Variable number of activities to be performed in the factory				
		1	2	3	4	----------n

Figure 1.8. Variable number of fixed-length fields
Each manufacturing step is represented by a fixed-length field. Some products require less steps than others and will therefore use less fields in the variable part of the record.

As we have said, the fixed length-format is most commonly used. Users tend to choose this format, even when a variable-length format seems to be more appropriate. The reason is that variable-length records require more processing (more program steps, and therefore more time) than fixed-length records. In addition, the procedures are more complex, which means that programming (testing included) will take more time. The programs will consume more time when they are executed, since the procedures require more steps. A user will therefore tend to accept a certain loss of space in external storage, if the profit in saved storage space does not balance the extra processing time.

Record Identification

Logical records normally include an identification called a *key*. This key is usually unique within the set of records, thus making each record unique within the set.

The record identification which uniquely identifies a logical record in a set is called the *primary key*. Examples are:

- Employee number in payroll records
- Order number in order records
- Part number in inventory records.

The order of sequence among the records in a collection is usually chosen according to the primary key, but may be changed by selecting another field as a key and by re-sorting the records according to the new key.

Fields that are selected for re-sorting are called *secondary keys*. Examples are:

- Department in payroll records
- Delivery date in order records
- Inventory level in inventory records.

A secondary key is usually not unique within a group of records. Figure 1.9 illustrates a set of punched cards containing personnel information. The primary key is a personnel number which is unique within the set of cards. As a secondary key, any other field or even group of fields may be used. The second example in Figure 1.9 shows the same set, re-sorted according to the secondary key *'Department'*.

As will be shown in the next chapter, the primary key is an important item when organizing data in external storage.

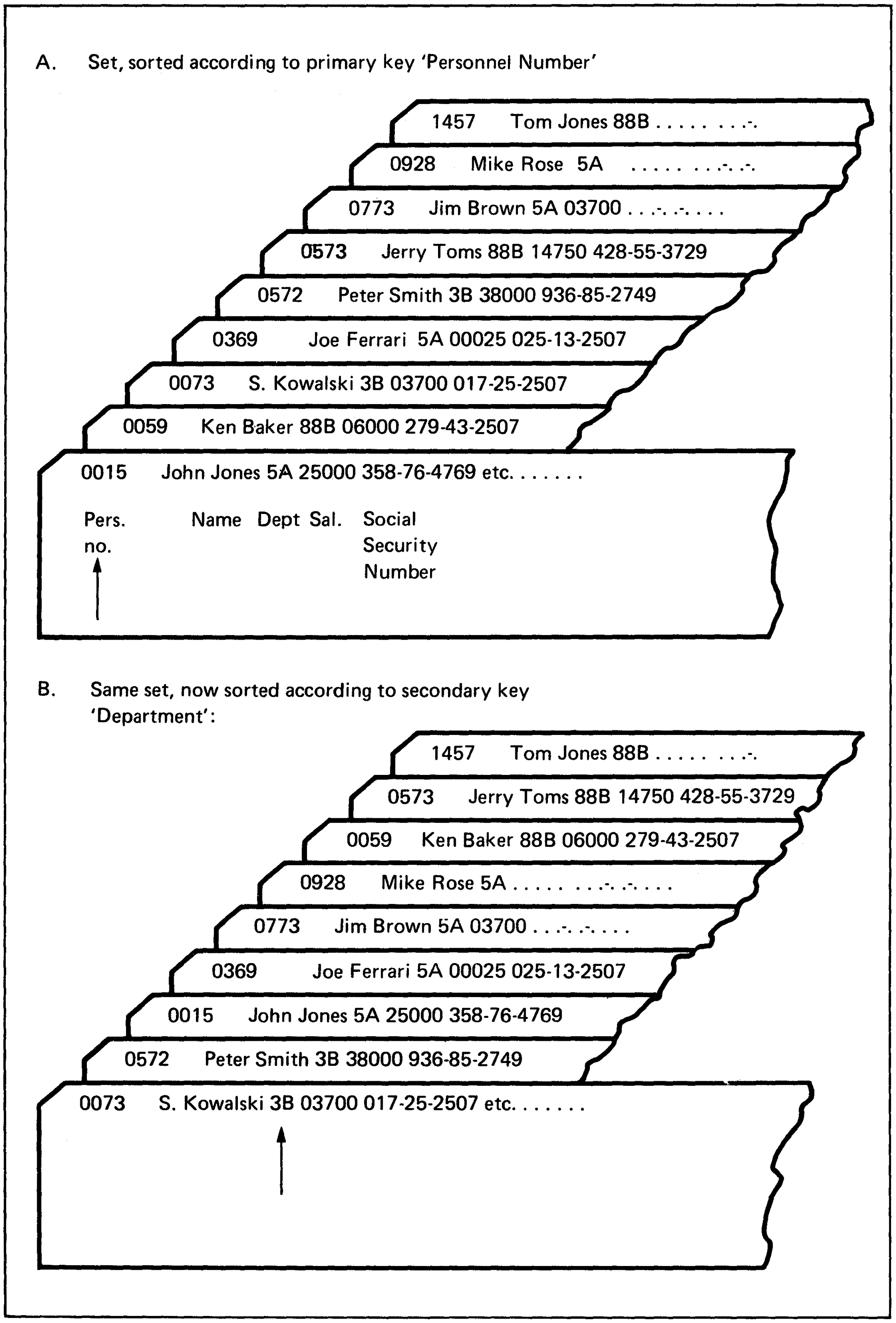

A. Set, sorted according to primary key 'Personnel Number'
1457 Tom Jones 88B . . . .-.
0928 Mike Rose 5A . . . .-. .-.
0773 Jim Brown 5A 03700 . . .-. .-. . . .
0573 Jerry Toms 88B 14750 428-55-3729
0572 Peter Smith 3B 38000 936-85-2749
0369 Joe Ferrari 5A 00025 025-13-2507
0073 S. Kowalski 3B 03700 017-25-2507
0059 Ken Baker 88B 06000 279-43-2507
0015 John Jones 5A 25000 358-76-4769 etc. . . . . . .
Pers. no.
Name
Dept
Sal.
Social Security Number
B. Same set, now sorted according to secondary key 'Department':
1457 Tom Jones 88B . . . .-.
0573 Jerry Toms 88B 14750 428-55-3729
0059 Ken Baker 88B 06000 279-43-2507
0928 Mike Rose 5A . . . .-. .-. . . .
0773 Jim Brown 5A 03700 . . .-. .-. . . .
0369 Joe Ferrari 5A 00025 025-13-2507
0015 John Jones 5A 25000 358-76-4769
0572 Peter Smith 3B 38000 936-85-2749
0073 S. Kowalski 3B 03700 017-25-2507 etc. . . . . . .

Figure 1.9. Primary key and secondary keys
Its primary key distinguishes a record from other records in a set.

Data Organization

In order to make use of data that is stored on an external device it is necessary to identify and retrieve that data. The techniques for identifying, storing, and retrieving data are collectively referred to as *Data Organization.* Data organization techniques are an important part of the services of any operating system.

Basically there are two forms of data organization:

- *Sequential* data organization
- *Direct* or *random* data organization.

Up to now, the most common form of data organization has been sequential (Figure 1.10A). Records are stored and placed adjacent to one another, and to retrieve any record, all preceding records must be scanned. The advantage of a sequential organization is rapid access to the next record in sequence (Figure 1.10A). Typical examples of sequentially organized data occur on punched card equipment, printers, paper tape, and magnetic tape.

Direct data organization ignores the physical sequence of records stored, and accesses records on the basis of their physical location in the storage device. The advantage of a direct organization is that any record in a collection can be reached without scanning all preceding records. A direct data organization is applicable to direct access devices such as disks, drums, and data cells.

For both types of data organization, the primary key is used to identify records. For sequential organization the records are normally ordered in key sequence; in a direct organization they are usually not, although the primary key is still used for identification. There are two ways of identifying records in a direct data organization:

- Through an *index* (Figure 1.10B)
- Through an *algorithm* (Figure 1.10C).

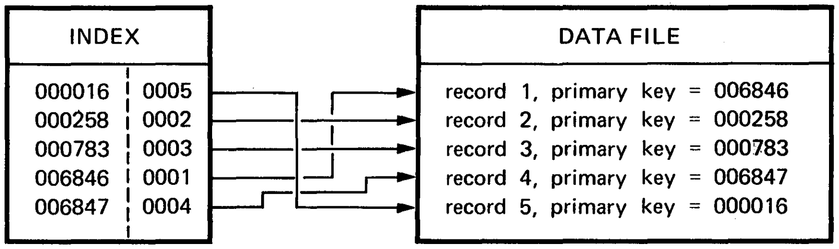
Direct organization through an index means that there is a separate list which contains the primary keys of records in the file, each key accompanied by a reference to the actual data in external storage. This list (index) may be a separate part of the file, or it may be a separate file itself. The index is maintained in a sequential organization, according to the values of the keys. Therefore, direct organization through an index is often called *indexed sequential* organization.

Direct organization through an algorithm means that a formula is used to establish a relationship between the primary key of a record and the address of that record in a direct access storage device (DASD). This technique is often called *'randomizing'* or *'hashing'*.
The formula (transformation algorithm) is usually arrived at by trial: an algorithm is chosen and all primary keys are processed. Then an analysis of the resulting storage addresses is made and, if the transformation algorithm proves to be inadequate, a new one is developed and tested. An algorithm is considered inadequate if it transforms many different primary keys into the same storage address, or if many addresses are never used.

A. Sequential data organization.

record 1 prim.key = 000016	record 2 prim.key = 000258	record 3 prim.key = 000783	record 4 prim.key = 006846	record 5 prim.key = 006847	etc.

B. Direct data organization through an index.
The actual sequence of the data records in the data file is usually the same as the sequence in which records are entered when the file is created or updated.



INDEX	
000016	0005
000258	0002
000783	0003
006846	0001
006847	0004

DATA FILE
record 1, primary key = 006846
record 2, primary key = 000258
record 3, primary key = 000783
record 4, primary key = 006847
record 5, primary key = 000016

C. Two examples of direct data organization through an algorithm.
The first example shows that the contents of the primary key is also the address of the data. Each value has a unique record address. This is the simplest type of transformation algorithm.
DASD addresses marked '*' are empty locations since no records are present with that primary key (= address).
The second example shows an algorithm that gives non-unique addresses. This might happen when a user wants only the track address to be calculated and wants to organize the records within the track himself, assuming of course that a track has space for more than one record. In this example 10 records can be stored on a track. (In certain types of disk storages devices, manufactured by IBM, this type of track organization is now done by hardware, making the transformation problem much easier than in the past.)

Randomizing example 1:

PRIMARY KEY	DASD ADDR
21320	021320
21321	021321
	021322*
21323	021323
	021324*
	021325*
21326	021326
21327	021327

Randomizing example 2:

PRIMARY KEY	DASD ADDR
01861	0000186
01868	0000186
02478	0000247
02500	0000250
02503	0000250
02509	0000250

TRANSFORMATION ALGORITHM

$$\frac{\text{(Primary Key)}}{10}$$

Figure 1.10. Examples of sequential and direct data organization
In sequential organization, data records with successively higher keys are found in physically sequential locations. In random organization, either an index or an algorithm is used for relating keys to addresses of data records that need not be in physically sequential locations.

Physical and Logical Data Organization

Data organization has been discussed thus far in terms of data in external storage. This is known as the physical organization of data. Physical data organization refers to the techniques used for storing and maintaining data in external storage as well as to the physical format of that data in external storage.

The user may view the data in a way that is different from its physical organization. If direct organization through an index is used to organize the data physically, for example, the user may still view his data as if it were in logical sequence (retrievable in key sequence) although physically the data was stored at random. In addition, a user may establish relationships between parts of his data: there may be relationships between data within one record, or relationships between different records (see: *Data base concepts* later in this section). The user's view of the data is called the logical data organization. As will be explained later, a user may specify that records are grouped together into *blocks* of records in order to improve processing speed or to reduce external storage space requirements. In such a case, the physical data organization operates on blocks of data records while the user may still use a logical data organization which operates on single logical records.

The Role of the Operating System in Data Organization

For both the logical and the physical data organization, an operating system provides sets of input/output (I/O) routines for storing, retrieving, and updating data records. There are different sets of I/O routines available, each set supporting a certain type of data organization: sequential, indexed sequential, or direct. In some cases, a mixture is possible. Such a set of I/O routines is called an *access method*. A user may decide to maintain his data by means of a direct access method (e.g. through an algorithm) and, for certain programs, to retrieve this data in physical sequential order by means of a sequential access method.

In some cases it may be necessary to give data records a special format in order to make them acceptable to a specific access method. In other cases, it may be necessary to adapt data records in order to be able to use different access methods on the same file. For example, it may be necessary to add additional data to allow an access method to identify a record. This type of data is called *control information*.

A user's problem program normally operates on single records which are called *logical records*. In his logical data organization, a user provides an interface between his logical records and the access method. This includes *preformatting* by which a logical record is 'mapped' into a format acceptable to a specific access method and device type. This record is then 'mapped' by the access method into what is called a *stored record*, which is then written to external storage as a *physical record* or *physical block*. In some cases the logical, stored, and physical formats are identical, as in punched card devices; in other cases they may be different.

We have described what happens when data is written (stored); when it is read (retrieved), the mapping is done the other way around, as is shown in Figure 1.11.

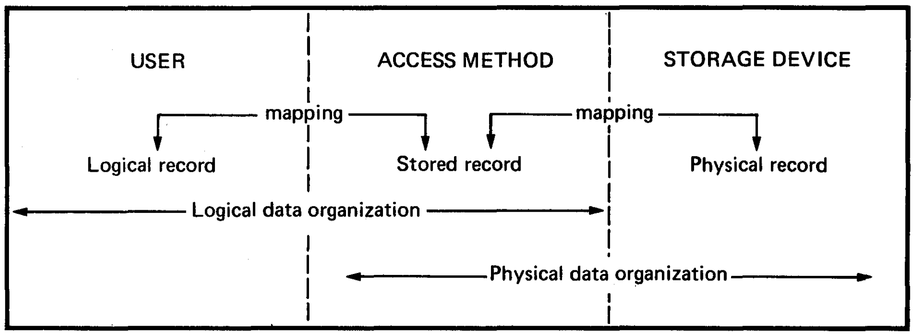


Figure 1.11. Data organization
Records are mapped to formats acceptable to the user (logical record), the access method (stored record), and the storage device (physical record).

Storage Devices

Input/output devices include:

- Card readers and card punches
- Printers
- Typewriter-keyboards
- Magnetic tape units
- Direct access storages (disk, drum, data cell)
- Paper tape readers and punches
- Teleprocessing equipment
- Optical readers and magnetic readers
- Process control equipment
- Visual display units
- Audio response units
- Diskette units

It is the main function of the I/O devices to provide an interface between main storage and data stored in external storage media such as punched cards, magnetic tape, diskettes, or disk packs.

There are several characteristics that classify storage devices. These include data capacity, addressability (how data is located), access time to data, data transfer rate, physical advantages and limitations, and cost. Devices are selected by comparing these characteristics with the requirements of applications.

The capacity of a storage device is the amount of data that can be stored on it. In some types of devices, part of this capacity is used by the device itself for error checking and for synchronizing information. The remaining capacity, accessible by programs, is usually measured in bytes.

Accessibility can be considered under two aspects:

- addressing - how data is located
- resolution - how much data (bits, bytes, words) is referred by one data address.

Addressing may simply be implied. For example, cards are read from a card reader in the order in which they are placed in the hopper. A particular card cannot be selected for reading. The opposite extreme is direct access or direct addressing, where data is referred to explicitly by location.

Access time is the time needed to locate the beginning of the data desired. For magnetic tape this may be the time necessary to move the tape to the beginning of the next record. Access time for direct access devices such as disk consists of 'seek time', which is the time required to position the read/write head to a particular track, plus the 'latency', or time necessary to reach the beginning of the desired record on that track. Once the beginning of a desired record has been reached, transfer of data between main storage and the device occurs at a specific transfer rate (bytes per second).

Record Blocking

Physical records, for instance, on a magnetic tape are separated by *interrecord gaps*. These 'empty spaces' between the physical records are needed to allow the magnetic tape device to gain speed before reading or writing and to come to a complete standstill after reading or writing. This start-stop time is an important part of the access time for magnetic tape devices.

A technique to reduce the number of interrecord gaps is to group more than one logical record into a *physical block*. For magnetic tape this may result in faster processing; for disk it may result in more efficient use of space and faster processing.

The 'blocking' of records before writing and the 'de-blocking' after reading is normally done by the operating system, without intervention by the user. The user only supplies the system the size of physical blocks he wants written, and the system then calculates the number of logical records that should be put in a physical block.

A. Unblocked records. Each logical record is a physical block.

B. Blocked records. More than one logical record in a physical block.

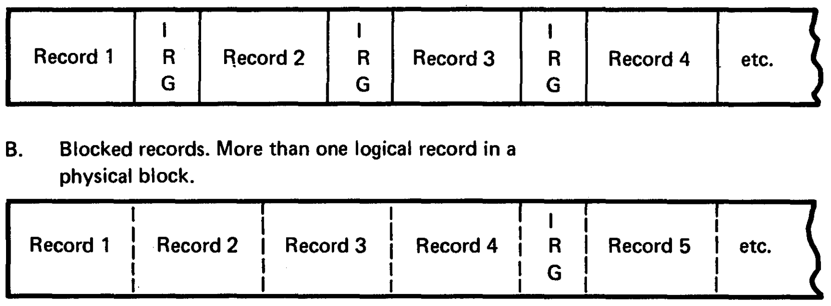


Figure 1.12. Unblocked and blocked records
Blocking is a technique for saving space and processing time.

When blocked, records need more space in main storage for input and output areas than when unblocked. On the other hand, time is saved because actual reading and writing takes place only when a new block is needed (input) or when a current block has been completely filled (output). In choosing blocksize, therefore, the user must balance the space he has available for input/output areas against the average access time required to read or write a new block. In some cases the maximum blocksize is limited not only because of space available in main storage, but also because of the physical characteristics of the external storage device. The maximum blocksize of a punched card is obvious; but data blocks on direct access storage devices are also limited in size, since the size of a track limits the maximum size of a block. And, last but not least, an operating system usually also specifies a maximum size of blocks that can be transferred, without regard to any type of device.

File Labeling

The same data file is often used by more than one program, in more than one application, and even by more than one computer system. A company may, for example, send the output file of its payroll program to a bank for processing the payments. In this case the company and the bank must agree on the format of the data: the arrangement of fields within records, the record size, the number of logical records in a block (blocking factor), and any required control information. But it is equally important that the file be identified as a payroll file. This is done by labeling.

It is conceivable that a computer operator might put a stock dividend file onto the computer, believing that it is a payroll file. The payroll application will start treating stock data as payroll data. In many cases this will be detected by the system because the blocksize of the wrong file will not be the same as the expected blocksize specified by the program. But in many other cases the blocksizes may be the same because of a standard blocksize. For instance, when the contents of a set of punched cards is written on magnetic tape, a blocksize of a multiple of 80 bytes will frequently be used.

File labeling makes sure that a file can always be recognized and identified. Generally, there are three different types of labels:

- labels for magnetic tape files (tape labels)
- labels for direct access files (DASD labels)
- labels for diskette files.

All three types are used to identify and recognize data files although the techniques are different. Thus, they are conceptually the same, although physically different.

Tape Labels

Tape labels are frequently used for data interchange between different computer systems. Therefore it is necessary that a tape file be recognized in the same way by all systems. Magnetic tape labeling has been standardized by industry-wide agreements. This means that a data file on magnetic tape is delimited by one or more special records, called *labels*, at the beginning and at the end of this file.

Labels that precede the data file are called *header labels*. They contain information about the data, such as record format and blocking factor. They also contain information about the physical medium (reel of tape), such as the reel identification number.

Labels following the data file are called *trailer labels*. They contain additional information about the file, such as a block count for checking against a count of the number of blocks processed, in order to make sure that no blocks have been skipped due to some error.

Labels are separated from data by a special single-character record called a *tapemark*. It is written as a separate block and allows a system to distinguish between labels and data. Figure 1.13 shows the basic labeling scheme for one magnetic tape file. If a file is written over more than one reel of tape, this scheme applies also to each section of the file, on each volume. If, on a reel of tape, more than one file is written, this scheme applies to each subsequent file on that volume.

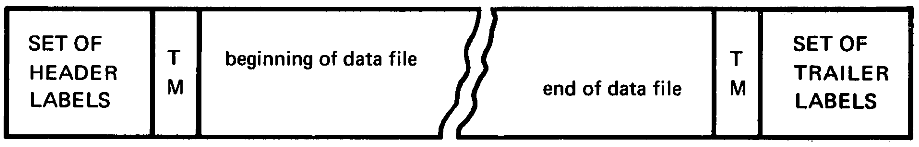


Figure 1.13. The principles of tape labeling
Two sets of labels, separated from the data by tapemarks, are used for identifying files on magnetic tape.

DASD Labels

A DASD volume (for example, a disk pack), like a reel of tape, may contain more than one file, and each file may be anywhere on the volume. In order to locate any particular file, there is a table on each volume called the *Volume Table Of Contents* (VTOC). In fact the VTOC is a set of labels that identify each file on the volume, and through which each file must be located.

DASD files are not usually interchanged the way tapes are. Therefore there is at present no need for an international standard for disk labeling. DASD labels are used only to locate, identify, and recognize data files within one system, and every computer manufacturer has his own standard label formats and labeling techniques for DASD.

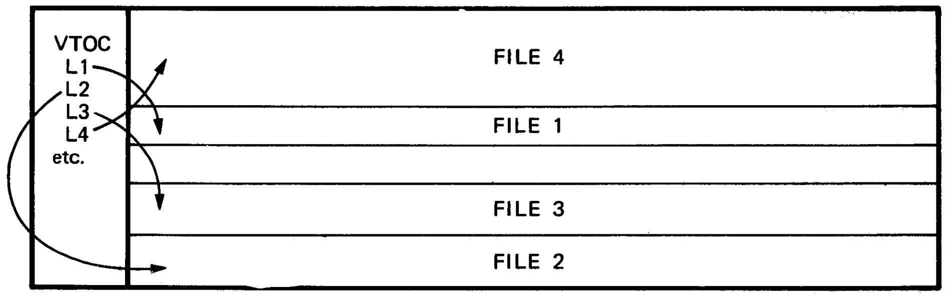


Figure 1.14. The principles of disk or diskette labeling per volume
Each DASD or diskette volume has a table of contents consisting of a set of labels, one label for each file on the volume.

Diskette Labels

A diskette volume may contain more than one file, and each file may be anywhere on the volume. In order to locate any particular file, there is a table on each volume called the *Volume Table of Contents* (VTOC). In fact the VTOC is a set of labels that identify each file on the volume, and through which each file must be located (see Figure 1.14.).

Label Processing

Tape labels, diskette labels, and disk labels are normally processed by a set of label handling routines which are part of the operating system supplied by the manufacturer.

Input/Output Control

The actual reading and writing of data blocks is normally done by the operating system. The user has a set of instructions at his disposal to inform the system what input or output action is desired. These instructions are then interpreted by the operating system and translated into actual input and output functions. The routines that perform those functions act as an interface between the user's problem program and the external devices. They are usually collectively referred to as the *Input/Output Control System* (IOCS). Examples of functions are:

- actual reading and writing
- blocking/deblocking of logical records
- label processing
- checking for error conditions.

In addition, the IOCS provides a means of processing data from input/output buffers. A buffer is an intermediate storage area between an external device and the user's I/O area, and allows programs to run faster.

Actual reading and writing takes a considerable amount of time because of mechanical movements in the I/O device. On the other hand, a move of a data record between locations in main storage is very fast. When data is buffered, we can make use of the speed of this internal move.

An extra I/O area is placed between the external device and the I/O area in the user's problem program. This extra area is called a buffer. As shown in Figure 1.15, before the first logical record of a file is read by the program of the user, the system reads it first and places it into the buffer. When the user's problem program asks for this record, it is moved from the buffer to the user's I/O area where it can be processed. IOCS then already reads the next subsequent record into the buffer while the problem program is still processing the first record, etc. As a result there is an overlap between reading and processing, and the program runs much faster.

Data Base Concepts

Most of the files in today's types of applications continue to use separate records as entities which are unrelated to other records. That is to say, many current implementations show the same data used in many different files, each of them specially designed for a different purpose. For example, a customer number may be used in applications for the commercial department (in order files and invoice files), in applications for the

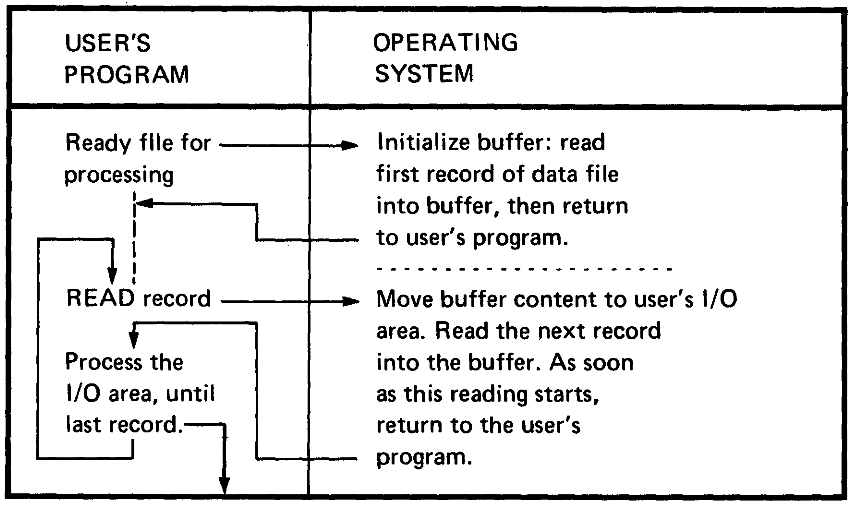

USER'S PROGRAM
OPERATING SYSTEM
Ready file for processing
Initialize buffer: read first record of data file into buffer, then return to user's program.
READ record
Move buffer content to user's I/O area. Read the next record into the buffer. As soon as this reading starts, return to the user's program.
Process the I/O area, until last record.

Figure 1.15. Buffered data
Without buffering, the user's program, after processing a record, has to wait while the external device transfers a new record to the I/O area.

production planning department (in different files for loading and scheduling), in applications for the production control department (as an identification of products being produced), and so on. The same holds for other data items such as amount of products to be or being produced, and delivery dates. Since the same data is present in different files for different departments there is a possibility that each application interprets data differently, from its own point of view, and since each application may update its own data independently from the other applications there is also a possible danger of ambiguities among the different files.

However, the advanced storage and processing capabilities of a modern computer system enable a user to implement more complex applications than he was used to in the past. New applications may be designed to share the same data with other applications. This removes ambiguities, since a data item is recorded only once, and it also minimizes data redundancy for the same reason. Such applications are usually more complex than they used to be in the past, mainly because of the more complex data structure.

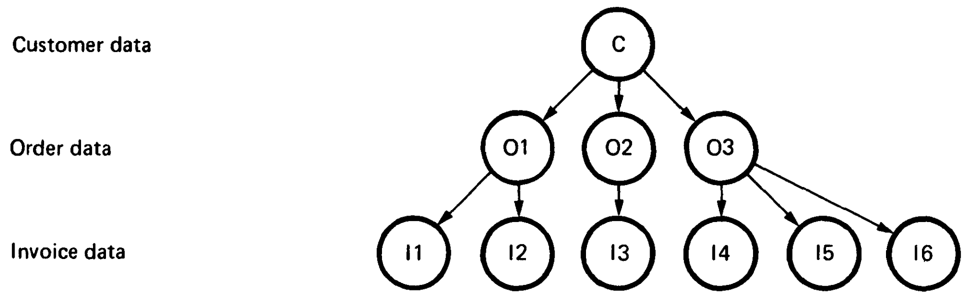
The collection of data which is shared by multiple applications or which has a more complex structure is called a *data base*. It usually contains much data, and usually this data is fundamental to the enterprise because it is present in the data base only, making it one of the most valuable resources for the enterprise. In many cases, an information system in a data base environment shows several separate applications of the past integrated into one, focussed on one specific area of an enterprise such as production planning/control together with sales and transports, or one system for the entire personnel department.

A discussion of data base concepts is beyond the scope of this manual. A brief introduction to the concepts of data structuring is felt to be very useful, however, because the design of a data base application presents one of the greatest challenges to system designers of today.

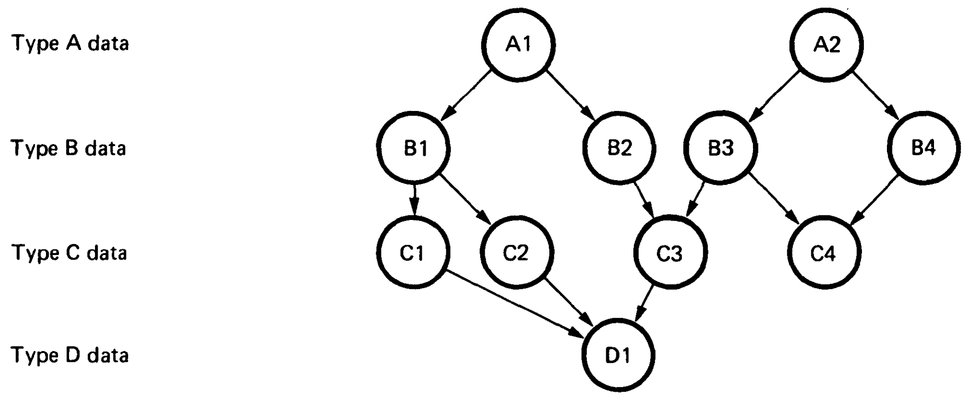
Data Structures

When structuring data, the implementor of an information system incorporates physical relationships between various types of data in the design of the data itself. Data may be structured in many ways. For example, the data structure may be represented as a 'tree' or as a 'network', as is shown in the following diagrams.

Tree-structured data example:



Network-structured data example:



Implementation of Data Structures

The way of implementing a data structure depends on the type of structure to be implemented. Some of the methods followed are typical, however, and can be used in several types of data structures. This chapter will introduce some possibilities for implementing a tree structure which happens to be the most commonly used structure, applicable for many different types of applications.

In implementing a data structure, the implementor usually provides for additional information to accompany the data. This may be in additional fields in the records, or it may be separate. Whether in the records or separately, in most cases one makes use of *pointers*: address fields that refer to other records. Pointers can be used to refer from one record to another in order to establish a relationship between those two records, for example their sequence.

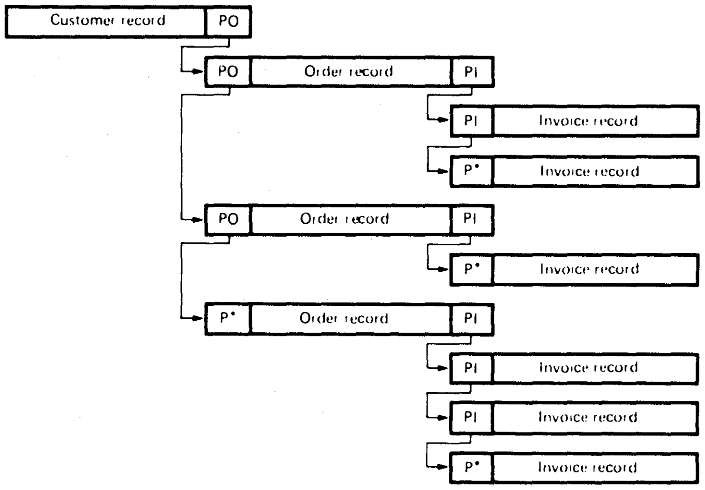
Assume a tree-structured collection of data as was shown above. At the top of the structure there is a collection of customer records each of which may have been connected to one or more order records. Each order record may, in turn, be connected to one or more invoice records. From bottom to top, this structure represents a set of invoices that belong to a set of orders, and the set of orders belongs to a set of customers. It is the task of an implementor of an information system to provide a tool by means of which these relationships can be identified and used.

If it is possible to assign keys that are logically related to one another, it may be possible to use an index which reflects the structure of the data. For example, suppose that customer records can be identified by a key within the range *C001* ... *C999*, and that the keys of the order records include the key of the customer record: *CnnnO001* ... *CnnnO999*, and that the keys of the invoice records include the key of the order record: *CnnnOnnnI01* ... *CnnnOnnnI99*. In this case, the index could possibly have the following format:

....		
....		
C286	P001	(customer)
C286O001	P002	(order 1)
C286O001I01	P003	(invoice 1)
C286O001I02	P004	(invoice 2)
C286O002	P005	(order 2)
C286O002I01	P006	(invoice 3)
C286O003	P007	(order 3)
C286O003I01	P008	(invoice 4)
C286O003I02	P009	(invoice 5)
C286O003I03	P010	(invoice 6)
....		
....		

In the example above, *P001* ... *P010* are pointers which contain the addresses of the associated records. For example, for key *C286O003I02*, pointer *P009* represents the address of the invoice record with that key (the second invoice for the third order for customer C286).

In this structure of keys, the data structure is implied in the sequence of the keys in the index. If the keys cannot be structured as indicated above, as is often the case, other methods must be used. One possible technique is the *list structure.* Here, pointers are put in the data records themselves, for example as follows:

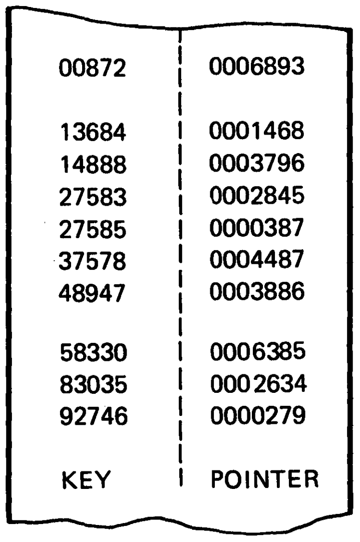


In this scheme, *PO* represents a pointer to an order record, *PI* represents a pointer to an invoice record. Each list or sublist ends with a *P** pointer that indicates the end of a list. The entry to a customer record (the 'root' of a tree) can be found through an index, and any order record or invoice record then can be found by following the pointer chains.

One of the most important problems of structured data is that any change in the structure must be maintained in the pointer strings as well. For example, the addition of a new order record to the structure above must cause the *P** pointer in the last old order record to be changed into a valid pointer to the new order record which is inserted. Also care must be taken that no order record is deleted as long as it still contains a valid pointer to an invoice record.

In many cases, both an index and pointers in the records are used. The index then is used to locate individual records and the pointers in the records are used to find the proper relationships with other records. Also (not shown in the preceding diagram), each record often contains a pointer to the 'top' of the hierarchy to which it belongs (above: order records belong to a customer record, and invoice records belong to an order record). This allows an implementation to scan pointer chains both upward and downward.

Assume that for customer *00872* the following orders are present: *92746, 83035*, and *58330*. Further assume that the following invoices exist: *27583* and *27585* for order *92746, 48947* for order *83035*, and *13684, 14888, 37578* for order *58330*. The keys of all records (customer numbers, order numbers, and invoice numbers) are listed in an index in ascending sequence, each key accompanied by a pointer to the record:



KEY	POINTER	
00872	0006893	(customer)
13684	0001468	(invoice)
14888	0003796	(invoice)
27583	0002845	(invoice)
27585	0000387	(invoice)
37578	0004487	(invoice)
48947	0003886	(invoice)
58330	0006385	(order)
83035	0002634	(order)
92746	0000279	(order)

From the example above the reader may conclude that the customer numbers, order numbers, and invoice numbers always fall within a certain range so that the keys of customer records, order records, and invoice records are automatically separate in the index. Furthermore, the addresses (pointers) of the records are obtained through some kind of randomizing algorithm, and the different types of records occur randomly throughout the entire collection.

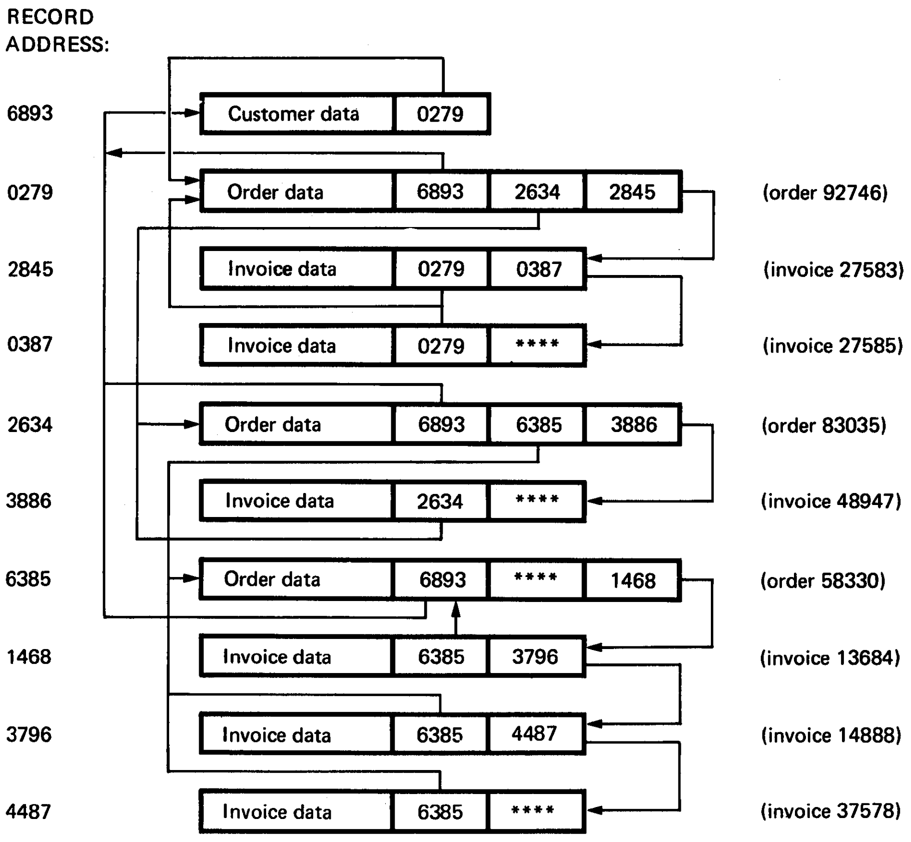
The index above can be used only to locate individual records; no interrelationship between any customer record and its order records and invoice records can be found here. The interrelationships between records are established in the data itself, as is shown in the last diagram.

In this example, the format of the different types of records is as follows:

- Customer records: data, plus a pointer to an order record
- order records: data, plus three pointers:
  1. an 'upward' pointer to the customer record
  2. a 'horizontal' pointer to the 'next' order record for the same customer
  3. a 'downward' pointer to an invoice record
- invoice records: data, plus two pointers:
  1. an 'upward' pointer to the order record
  2. a 'horizontal' pointer to the 'next' invoice record for the same order

If no 'next' record can be pointed to, the pointer is filled with a value that can easily be recognized. In the example above, this is the value ****; any other value such as, for example, *9999* can be used as well.

Any application that retrieves an invoice record can now retrieve the associated order record as well. And having retrieved an order record, it can also retrieve the customer record. In addition, the tree can be followed down from customer record through all associated order records and invoice records.


RECORD ADDRESS:
6893
0279
2845
0387
2634
3886
6385
1468
3796
4487
Customer data | 0279
Order data | 6893 | 2634 | 2845
Invoice data | 0279 | 0387
Invoice data | 0279 | ****
Order data | 6893 | 6385 | 3886
Invoice data | 2634 | ****
Order data | 6893 | **** | 1468
Invoice data | 6385 | 3796
Invoice data | 6385 | 4487
Invoice data | 6385 | ****
(order 92746)
(invoice 27583)
(invoice 27585)
(order 83035)
(invoice 48947)
(order 58330)
(invoice 13684)
(invoice 14888)
(invoice 37578)

References

This first section of this manual has introduced the major concepts of data management, without regard to any specific operating system. The following sections will discuss these items again (except data base) as they appear in DOS/VS for IBM System/370.

Section I made references to the following topics, which appear in other sections of this manual:

Fields and Records	Section 2: *Record Formats and Record Structures*
Data Organization	Section 3: *Access Methods and File Organization*
Storage Devices	Section 2: *Record Structures for the Various Devices*; Appendix 1: *Devices Supported by DOS/VS*
Record Blocking	Section 2: *Record Formats and Record Structures*; Section 3: *Input/Output Control System*
Labeling	Section 2: *File Labeling*
Input/Output Control	Section 3: *Input/Output Control System*; Section 4: *Entire*

Section 2: DOS/VS DATA MANAGEMENT FACILITIES

This section describes the separate components that constitute DOS/VS in terms of data management.

The first chapter discusses the various kinds of record and block formats that can be handled by the system. The next chapter relates those formats to the various device types. The different record and block structures for different device types are discussed.

A following chapter then relates the data files to the devices on which they are stored. The relationships between files and volumes are explained.

A chapter on file labeling discusses methods that are used to locate and identify data files.

The concluding chapters deal with special tools in DOS/VS such as logical relationship between data and devices, input/output control, the checkpoint/restart facility, and the means of maintaining data security and data integrity.

Record Formats and Record Structures

In data processing, a distinction is made between three different types of records:

- logical records
- stored records
- physical records (usually called physical blocks).

The examples in Figure 2.1 show how these three types of records may be represented.

A *logical record* is a grouping of related information, identified in a unique way, and treated as a unit by the application programmer. The concept of logical record is not restricted to external data, but is carried over into the definition of virtual storage items, including work areas in the problem program. A logical record is seen from the standpoint of its content rather than from its physical attributes.

For DOS/VS it is always assumed that a logical record is stored and retrieved as one unit of information. Examples are: a payroll record for each employee, an order record for each order, an inventory record for each item.

A *stored record* is the group of data as it is manipulated by the data management routines of DOS/VS. It contains the same information as a logical record, and it may be expanded by additional control information required by the operating system.

A *physical block* is the amount of data, written to or read from an external device as a unit. A physical block may contain one or more stored records, or part of one or more stored records (see *Spanned records*). In some cases a physical block is expanded by an additional block prefix.

The term *record format* refers to the choice a user must make between:

- fixed-length records, blocked or unblocked
- variable-length records, blocked or unblocked
- spanned records, blocked or unblocked
- undefined records.

The term *record structure* refers to the special requirements that may be specified by the physical characteristics of some device types. For example, an IBM direct access storage device requires that a block of data be preceded by a count area. This is not a requirement for magnetic tape. Thus, even if the format of the data is the same, its structure may be different when stored on different device types.

The following chapters discuss the record formats available in DOS/VS first, and then the record structures applicable to different device types. It should be noted that these discussions apply to general DOS/VS data management facilities, some of which may not be supported by a particular high-level programming language.

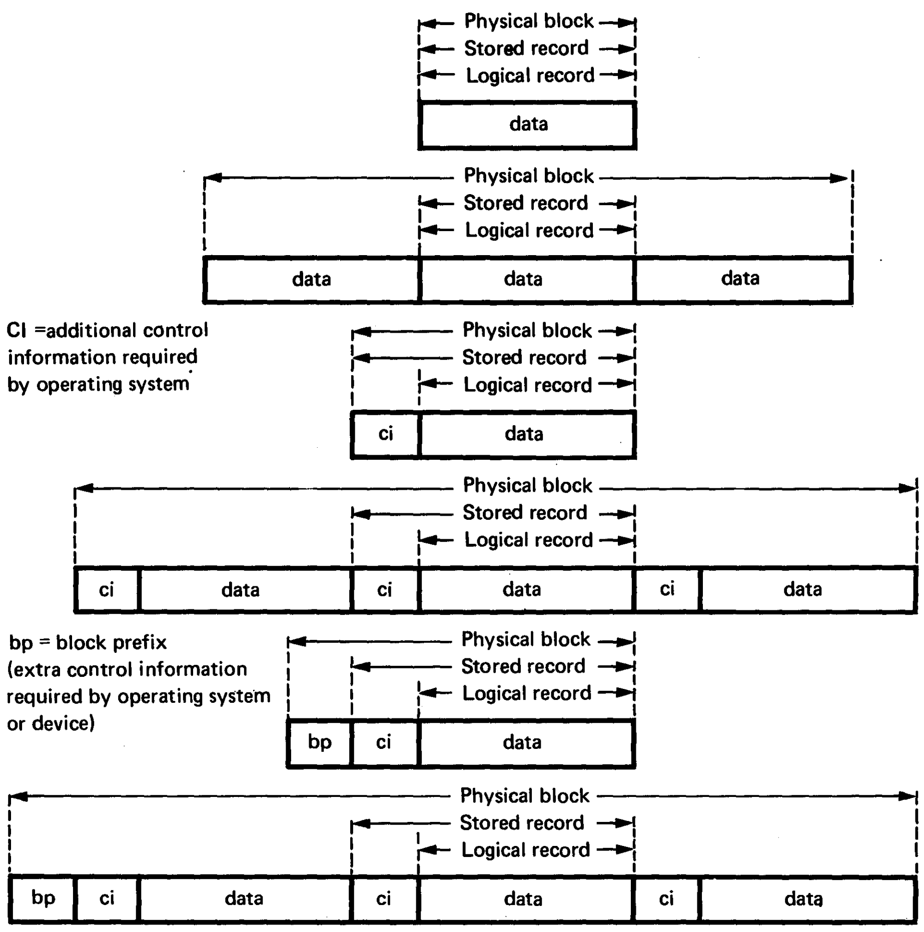


Figure 2.1. Examples of the differences between physical blocks, stored records, and logical records
The first example in Figure 2.1 shows a design in which the three types of records look the same. The second example indicates a design in which records are blocked; here the stored record looks the same as the logical record. The third and fourth examples indicate the use of additional control information; here a stored record is formed by a logical record plus the control information. The last two examples show a design in which a block prefix is placed at the beginning of a physical block. There are other possibilities which will be explained when the various record formats are discussed in detail.

Record Formats in DOS/VS

The application programmer is normally involved with logical records only. He may be expected to supply additional information so that stored records can be built by the DOS/VS data management routines. Physical blocks are handled by both the DOS/VS data management routines and the hardware systems.

Logical records are specified by the programmer as having one of the following formats:

- fixed length, blocked or unblocked
- variable length, blocked or unblocked, spanned or unspanned
- undefined.

In addition, provision is made for the processing of ASCII magnetic tape files, that have a special ASCII (Format D) format. Information about ASCII tape files is presented in Appendix 5 of this manual.

The prime consideration in the selection of a record format is the nature of the information itself. The programmer knows the type of input his program will accept, and the type of output it will produce. His selection of a record format is based on this knowledge, as well as on an understanding of the type of input/output devices on which the files are written and of the access method within DOS/VS that reads or writes the files.

Fixed-Length Records (Format F)

A logical record is considered as a fixed-length record (format F) when all records in the file are of the same length. Format F records may be blocked or unblocked, as is shown in Figure 2.2. The number of logical records in a block is normally constant.

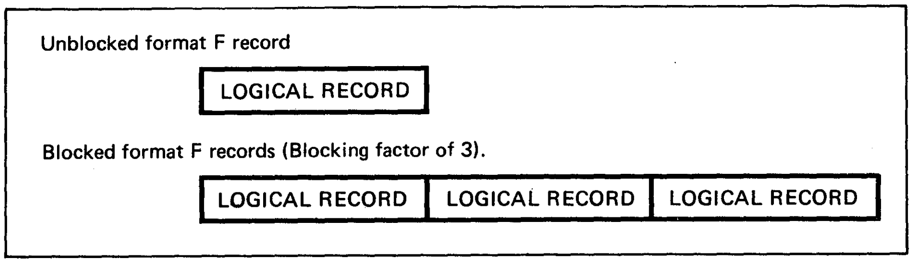

Unblocked format F record
LOGICAL RECORD
Blocked format F records (Blocking factor of 3).
LOGICAL RECORD
LOGICAL RECORD
LOGICAL RECORD

Figure 2.2. Fixed-length record format (format F)
Each logical record has the same length.

The system provides for the reading and writing of *truncated blocks* (short blocks) which may be found at the end of a file containing blocked records. This can happen when the last block is not completely filled with the number of logical records specified for a block (blocking factor).

Variable-Length Records (Format V)

A logical record is considered as a variable-length record (format V) when there are differences between the lengths of the records in a file. Format V records may be blocked or unblocked, as is shown in Figure 2.3. Logical records are expanded with additional control information specifying the record length *RL* of each individual logical record, including the length of the control information itself. Physical blocks are expanded with additional control information specifying the block length *BL* including the length of the control information itself. Programmers must supply the length of each logical record on output; the length of a logical record is communicated to a program by DOS/VS on input. The size of the I/O areas must be large enough to accommodate the maximum block length that may be expected.

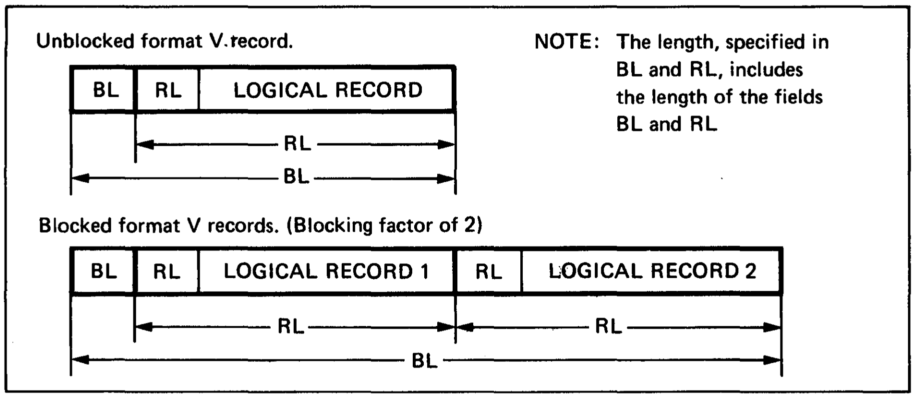

Unblocked format V record.
BL
RL
LOGICAL RECORD
RL
BL
NOTE: The length, specified in BL and RL, includes the length of the fields BL and RL
Blocked format V records. (Blocking factor of 2)
BL
RL
LOGICAL RECORD 1
RL
LOGICAL RECORD 2
RL
RL
BL

Figure 2.3. Variable-length record format (format V)
Each logical record and physical block must be accompanied by length specification.

Spanned Records (Format V)

Spanned records are records of varying length that may be written in one or more continuous blocks. When in spanned record format, logical records are broken into record *segments* before being written, and reassembled after having been read. This dividing into segments and reassembling is all done by the DOS/VS data management routines without intervention by the user.

Spanned records may be useful when a file is to be moved between device types with different characteristics, for instance, when the receiving device type imposes a physical limitation on the maximum block size of the sending device type. Another example when spanned records might be useful is in text processing applications where very long strings of text must be written. Figure 2.4 shows the fundamental concepts of spanned records.

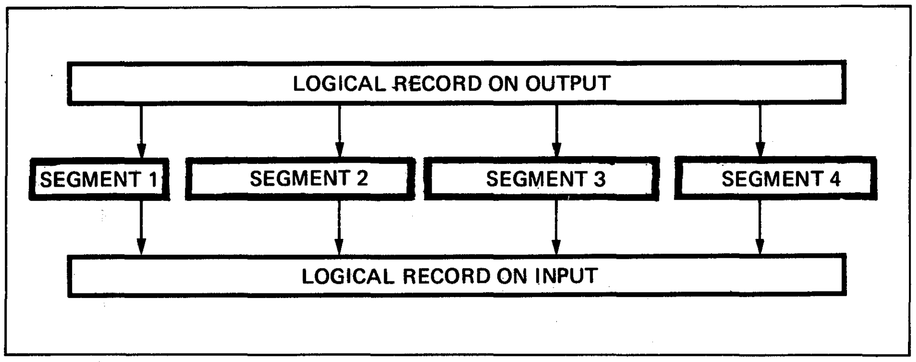


Figure 2.4 Record spanning
When useful for the application, the operating system splits logical records into segments before writing, and reassembles them after reading.

The processing of spanned records is an extension of variable-length processing. The main difference between variable-length processing and the processing of spanned records is that the programmer needs to be aware of the maximum data capacity of the I/O areas when processing variable-length records, but not when processing spanned records. The DOS/VS data management routines relieve him of this concern by dividing the logical records into segments that never exceed the size of the output area.

The structure of a spanned record is exactly the same as that of a variable-length record. The distinction between the processing of spanned records and variable-length records is made by the DOS/VS data management routines entirely. Specifying in a program that spanned records are to be processed, causes the logical records to be divided into segments (output), or to be constructed from one or more segments (input), whenever necessary. A complete logical record may be wholly contained in one physical block, or it may be contained in consecutive physical blocks. In the first case, one segment constitutes the logical record; in the latter case, more than one segment constitutes the logical record.

Spanned records may be stored in a blocked or unblocked format. Thus, a physical block may contain all or part of one or more logical records. A physical block will never contain more than one segment of a logical record; a logical record may start anywhere in any one physical block when blocked spanned records are processed.

Figure 2.5 shows possible arrangements of spanned records. The field *SL* has the same meaning for spanned records as the field *RL* for variable-length records (see Figure 2.3): it specifies the length of the data portion that follows including the length of the field RL itself. The only difference from variable length records is the meaning of the content of that field: RL in Figure 2.3 specifies the length of a complete record, whereas SL in Figure 2.5 specifies the length of part of a record (one segment). In both cases the programmer must supply the length of the logical record to the system when writing it, and in both cases the system will communicate the length of the complete logical record to the problem program when reading it. In both cases the system will compute the content of the field BL, according to the maximum allowable blocksize, as specified in the program.

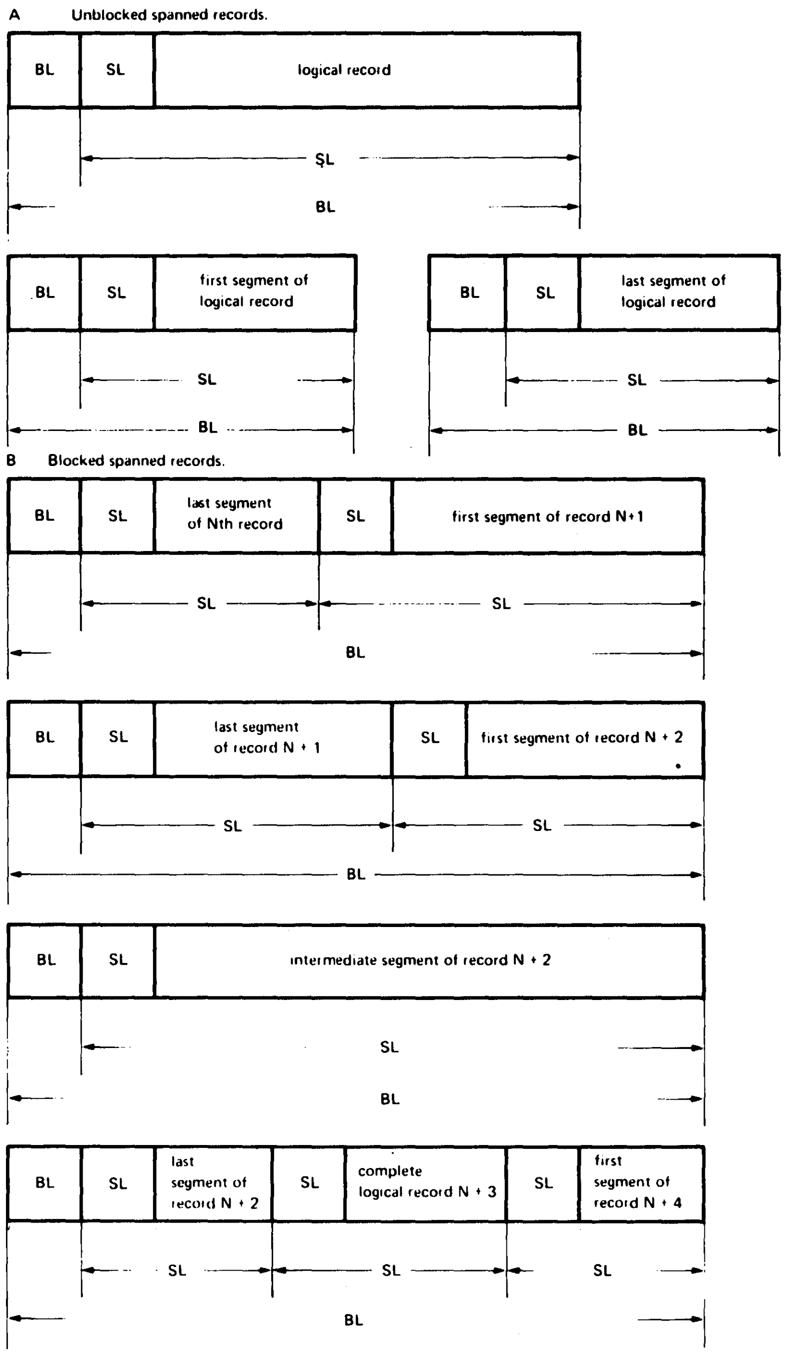

A Unblocked spanned records.
BL
SL
logical record
SL
BL
first segment of logical record
last segment of logical record
B Blocked spanned records.
last segment of Nth record
first segment of record N+1
last segment of record N + 1
first segment of record N + 2
intermediate segment of record N + 2
last segment of record N + 2
complete logical record N + 3
first segment of record N + 4

Figure 2.5 Spanned record format (format V)
Field SL specifies either the length of an entire record that is not divided into segments, or the length of part of a logical record that is divided into segments.

Control Information

The preceding text introduced the control information that is needed for the processing of variable-length records and spanned records:

- field BL (Block Length)
- field RL (Record Length)
- field SL (Segment Length)

These fields all have a size of four bytes, and are included in the physical blocks whenever necessary; the I/O areas involved must be large enough to accommodate the data plus the control information.

Block Descriptor

Field BL is called the block descriptor. It specifies the length of the data portion of any physical block, including the lengths of field BL itself and of any other control information fields that may be present in the physical block after field BL. The block descriptor is maintained by the data management routines of DOS/VS and is not furnished to the user; it is present in all structures of variable-length or spanned records, whether blocked or unblocked.

Record Descriptor (for Variable-Length Records)

Field RL is called the record descriptor. It specifies the length of a logical record in a physical block, including the length of field RL itself. When a problem program writes variable-length records it is expected to supply the length of each logical record to the data management routines of DOS/VS which construct field RL from the information supplied. When a problem program reads variable-length records the size of each logical record read is furnished to the problem program by the DOS/VS data management routines, from the field RL. The format of this information may differ, depending on the programming language used. More specific information can be found in the appropriate language reference manuals.

Segment Descriptor (for Spanned Records)

Field SL is called the segment descriptor. It specifies the length of a segment of a spanned logical record, including the length of field SL itself. Field SL may specify the length of one complete logical record if this record is not divided into segments (it is then contained completely in one physical block), or it may specify the length of part of a logical record if this logical record is divided into segments (it is then contained in more than one physical block). The content of field SL is computed by the DOS/VS data management routines when spanned records are written, from data that specifies the lengths of complete logical records as supplied by the problem program. When spanned records are read, the DOS/VS data management routines furnish the length of each complete and re-assembled logical record, computed from the contents of all fields SL for that logical record, to the problem program.

In addition to the length of a segment, field SL also specifies the *segment type*: whether it is the first, last, only, or an intermediate segment of a logical record. A segment may consist of a segment descriptor only, without any data following. It is then called a *null segment.* A special indicator in the segment descriptor specifies whether a segment is a null segment or not.

Undefined Records (Format U)

Any logical record that does not conform to either format F or format V is considered an undefined record (format U). The DOS/VS data management routines permit the processing of such records.

When undefined records are used, each logical record constitutes a physical block. Therefore, any blocking or deblocking must be performed by the user's problem program itself, and that program must locate individual records and fields.

Programs that write undefined records, must communicate the size of each individual undefined record to the DOS/VS data management routines; the size of each undefined record is communicated to problem programs that read such records. For more detailed information about this subject, the reader is referred to the manuals for the language processor he is using.

Undefined records are useful for paper tape, console printer-keyboards, optical character readers, and magnetic ink character readers. They may also be used on printers, magnetic tape, and direct access storage devices. In all cases, undefined records are processed as variable-length blocks without control information.

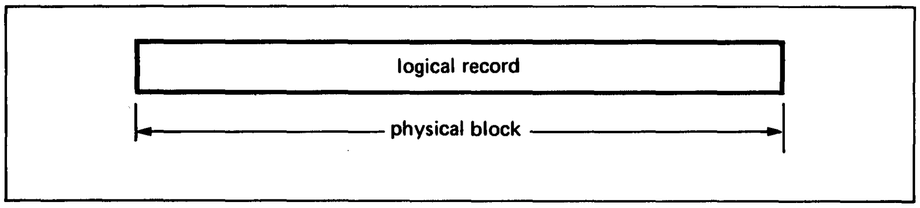


Figure 2.6 Undefined record format (format U)
Undefined records are processed as variable-length blocks without control information.

Control Character (optional)

The programmer has the option of specifying a control character to precede each logical record in a file. The control character is used for carriage control on printers, or for stacker selection on card punches. If specified, the control character must be the first character of each logical record; it is, however, not printed or punched. If the control character option is not used, the first character of a logical record is treated as the first data character; otherwise, the second character of each logical record will be treated as the first data character.

If the immediate destination of a record is a device that does not recognize the control character (for example, disk), the data management routines treat this control character as really the first data character of the record. This enables a user to store data with control characters on disk, for example, and later move the data from disk to a printer by another program. The control character will then be active for the printer, provided that the use of the control character option is specified.

Record Structures for the Various Device Types

The physical characteristics of certain device types require certain record structures, and may even limit the choice between record formats. This chapter summarizes the device types that are supported under DOS/VS and gives the record formats and record structures applicable to those types.

Note: It should be kept in mind that a specific programming language may have its own requirements for, and restrictions on, record structure and format. For detailed information about this subject the reader is referred to the manuals for the language processor he is using.

Card Readers and Card Punches

All card input must consist of fixed length, unblocked records (format F) records with a size of 96 characters or less for 96-column devices, or 80 characters or less for 80-column devices.

Card output may be any *unblocked* format F, V, or U. When format V records are punched, the descriptor bytes preceding the logical records (fields BL and RL) are not punched. If the control character option is used, it is used for stacker selection only and is not punched.

Note: LIOCS (see "Input/Output Control System") accommodates only 8-bit character codes. Column binary cards cannot be processed.

Printers

Printed output may be of any *unblocked* format F,V, or U. The maximum size of a record is defined by the maximum length of a print line. When format V records are printed, the descriptor bytes preceding the logical record (fields BL and RL) are not printed. If the optional control character is specified, it is used for carriage control only and is not printed.

Console

Records may be entered or displayed on the consoles in format F or U; the blocksize must not exceed 256 characters. The control character option cannot be used.

Magnetic Tapes

All standard record formats F, V, or U, blocked or unblocked, are acceptable to magnetic tape. All control bytes are transmitted. Magnetic tapes can be read forward or backward. Depending on the direction of reading, the data will appear in the I/O area either left-justified or right-justified. Spanned records may span volumes.

ASCII Tape Files.
If a DOS/VS system is required to process ASCII magnetic tape files, the system must be prepared for this processing by specifying *ASCII=YES* during system generation. Information about the format and structure of ASCII tape files and data is presented in Appendix 5 of this manual.

Paper Tape Readers

Paper tape input may consist of fixed length, unblocked records (format F) or undefined records (format U) terminated by an end-of-record character. Considerations for paper tape are presented in *Appendix 4.*

Paper Tape Punches

Paper tape output may consist of fixed length, unblocked records (format F) or undefined records (format U). End-of-record characters are added to undefined records by the DOS/VS data management routines. Considerations for paper tape are presented in *Appendix 4.*

Optical Character Readers (OCR)

Records can be read from an OCR device in format F or U. Format F is used when reading journal tapes containing an equal number of characters per line, or when reading documents from the 3886 optical reader; format U is used for 1287 files when the line length is variable.

When documents are processed, each field to be read can be treated as format F or U.

Optical Mark Readers (3881)

Records can be read only in fixed-length, unblocked format (F). Records can be up to 900 bytes long. The first six bytes of the record are used for record descriptor information. The remaining bytes of the record are used for the mark read data. BCD (binary coded decimal) data follows the mark read data if the BCD read feature is installed. If the serial number feature is installed, the serial number data follows the BCD data.

Magnetic Ink Character Readers (MICR)

Records can be read from MICR devices in format U. Record characteristics are determined by the settings of the field selection switches on the reader.

Diskette Storage Devices

Records can be read from diskette devices only in format F. Because a diskette is a pre-formatted medium, each track has the same fixed format.

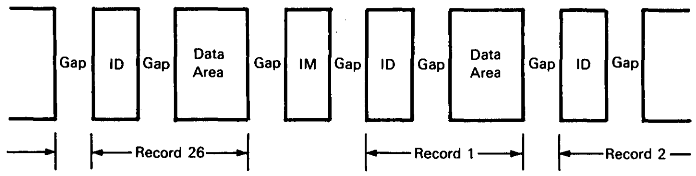


Figure 2.7 Schematic representation of the diskette track format. Index Mark (IM) and Record Identifier (ID) are control fields for the device.

Control Information on a Diskette Track
Control information is used to indicate the beginning of a track and the status of that track. The control information consists of:

- index marker (IM), and
- record identifier (ID)

An index marker indicates the physical beginning of a track. The record identification field identifies the location of each of the 26 record positions on a track. Included in the record identification field are bytes used to verify the validity of reading and writing. They are a function of the record verification circuits of the system and are automatically generated for each physical position. Each physical record position is formatted to allow data records ranging from 1 to 128 bytes in length.

Direct Access Storage Devices (DASD)

All standard record formats F, V, or U, blocked or unblocked, are acceptable to direct access storage devices. All control bytes are transmitted.
All direct access devices have the same track format: control information, a track descriptor record (called *record zero*), and data records.

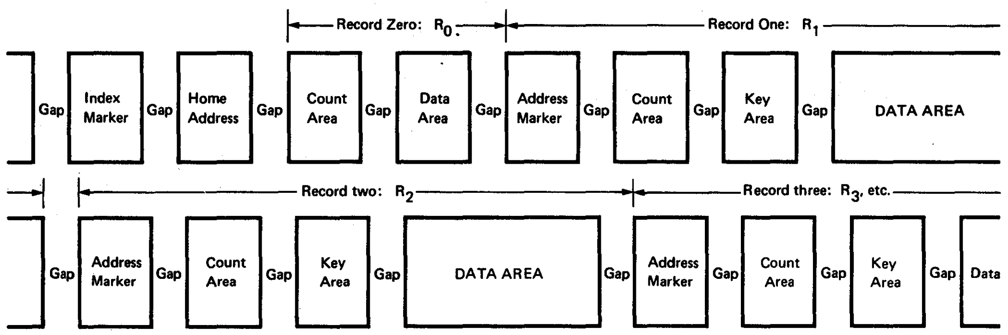


Figure 2.8 Schematic representation of the DASD track format
The key area is optional. Data areas that are preceded by a key area can be located by the device, by means of a search on key. Data areas that are not preceded by a key area must be identified by the user.

Control Information on a DASD Track
Control information is used to indicate the beginning of a track and the status of that track. It is normally of no concern to users.
The control information consists of:

- index marker, and
- home address.

An index marker indicates the physical beginning of a track. The home address defines the location of a track in terms of the physical parameters of the files; it is recorded in binary format and is seven bytes long. Index markers are written before the DASD packs are delivered to a user; home addresses are written by an IBM-supplied utility program (see *DASD initialization and maintenance*).

The flag byte is recorded on a track during a write-home-address operation. It indicates the condition of the track and is automatically propagated to all physical blocks as they are recorded on the track.

The four-byte address contains the cylinder number (*CC*) and the head number (*HH*), giving the physical location of a track.

The check bytes verify the validity of reading and writing. They are a function of the record verification circuits of the system, and are automatically appended to each separate area written on the same track.

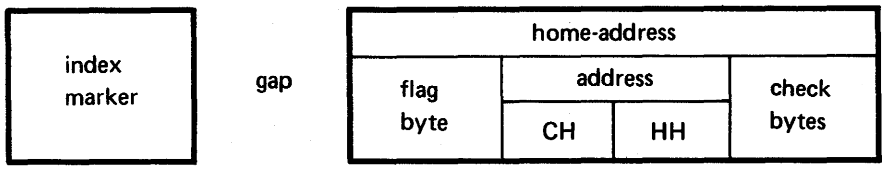


Figure 2.9 Schematic representation of the control information on a DASD track

Record Zero (R0)
The first block on every track facilitates the use of an alternate track, when

the original track is found to be defective (see *DASD initialization and maintenance*). Referred to as the *track descriptor record* or *record zero (R0)*, this record is unique in that it is not preceded by an address marker (see *DASD data records*, below). Figure 2.10 is a schematic representation of R0; the count area is similar to that used for normal data blocks described in the following paragraphs.

The data area can hold updated information about the data blocks on the track following R0. A discussion of the capacity record portion of the R0 data area will be found in the section *Access Methods* (Section 4, under *Direct Access Method*).

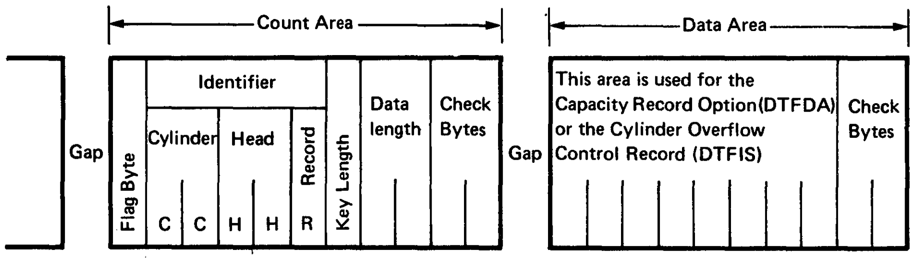


Figure 2.10 Schematic representation of record zero (R0)
R0 is called the track descriptor record because it can hold information about the data blocks following on the same track.

DASD Data Records

There are three sections in each physical DASD data block:

- the count area
- the key area (optional)
- the data area

Figure 2.11 is a schematic representation of blocks on DASD.

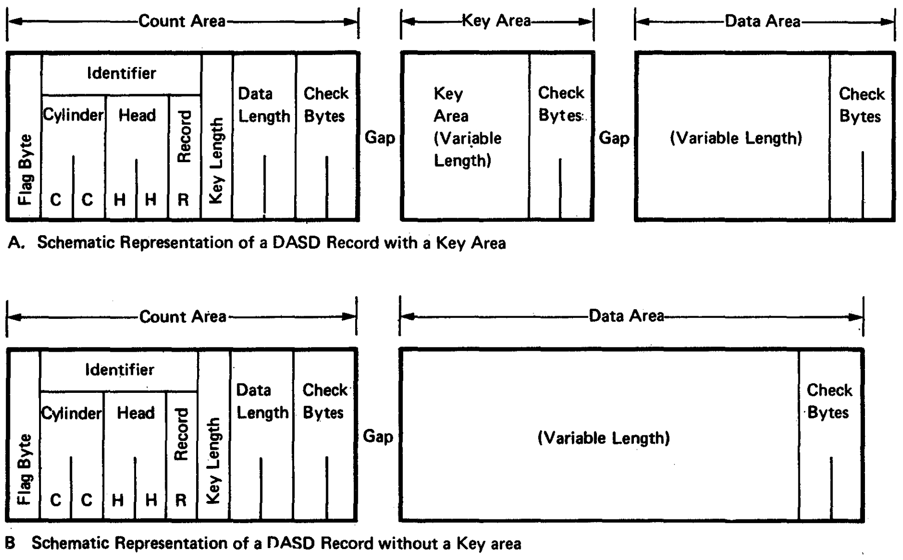


Figure 2.11 Data representation on DASD

The Count Area.
The count area consists of the flag byte (copied from the home address flag byte), the identifier field, the key length, the data length, and check bytes. The count area is recorded automatically, in binary form.

The identifier field (record ID) of five bytes contains the cylinder number (CC), the read-write head number (HH), and the record number (R) to define the physical location of the record on a volume. The record number is the sequential position of the record on the track. (Record zero, see above, is the first record on a track and the following records are numbered in ascending order.)

The key length is one byte and specifies the number of bytes in the key area of the block, not including the two check bytes in the key area. If the key area is not present, the key length is recorded as zero.

The data length is two bytes and specifies the number of bytes in the data portion of the block, not including the two check bytes in the data area.

The Key Area.
The key area consists of a key field and two check bytes. The key field is the external record identifier such as a part number or an employee number, that (mostly uniquely) identifies a logical record. It is usually the major control field of the data to which it is appended. The length of a key can vary from zero to a maximum of 255 bytes.

The Data Area.
The data area contains the actual data and two check bytes. The data area may contain one logical record, or a number of logical records grouped together into a blocked format. If spanned records are processed, the data area may contain all or part of one or more logical records.

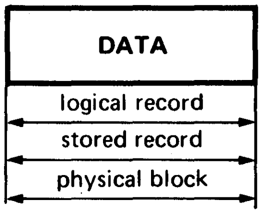
On DASD, records may be written in format F, V, or U, with or without keys. Spanned records (format V), however, can have a key with the first segment only; spanned records do not span volumes as they may do on magnetic tape.

Summary

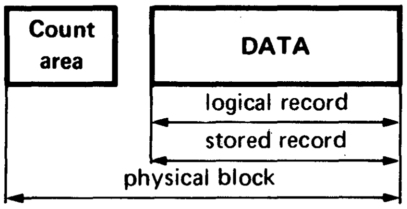
A summary of all record and block structures as they are applicable to the various types of I/O devices is presented in Figure 2.12.

UNBLOCKED, FIXED LENGTH RECORDS:

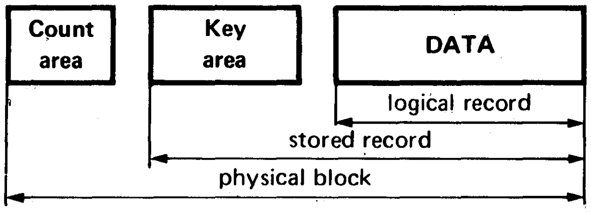
Unit record devices, magnetic tape, diskette, etc.



Direct access storage devices, without a key area.

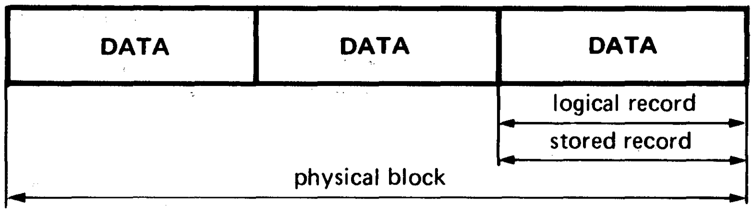


Direct access storage devices, with a key area.

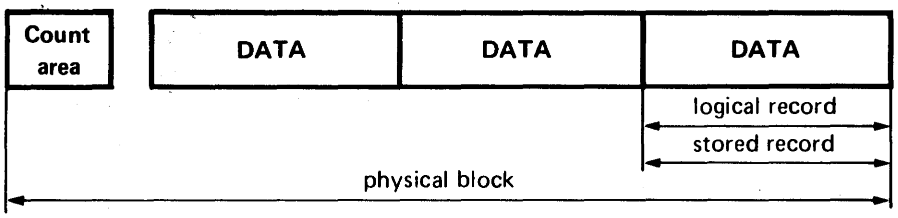


BLOCKED, FIXED LENGTH RECORDS:

Magnetic tape



Direct access storage devices, without a key area.



Direct access devices, with a key area.

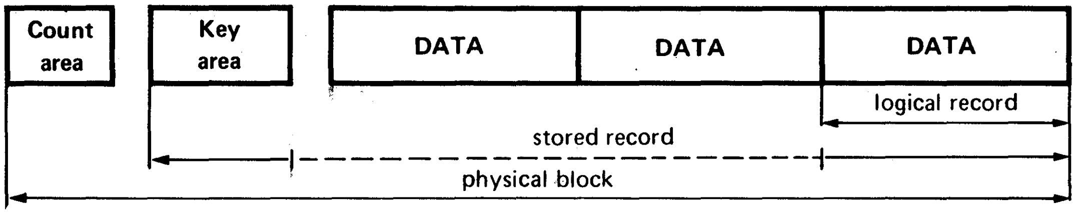
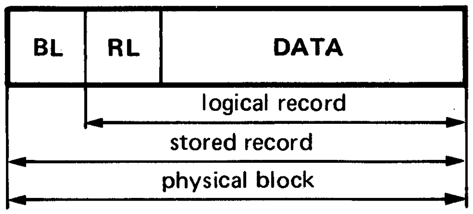


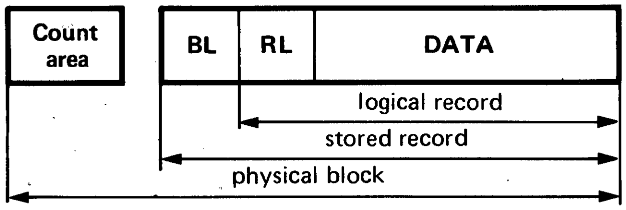
Figure 2.12. Record and block structures for the various devices (1 of 4)

UNBLOCKED, VARIABLE LENGTH RECORDS:

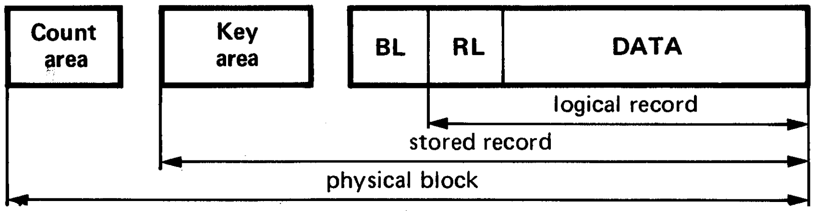
Magnetic tape



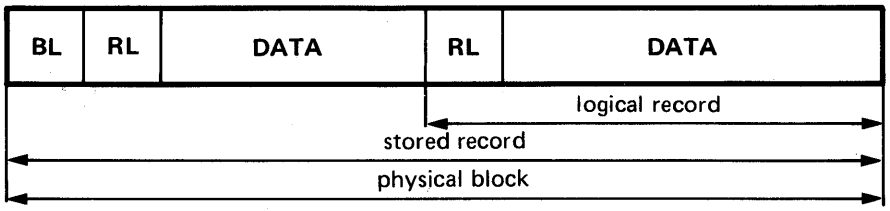
Direct access devices, without a key area



Direct access devices, with a key area



BLOCKED, VARIABLE LENGTH RECORDS:



Direct access storage devices, without a key area

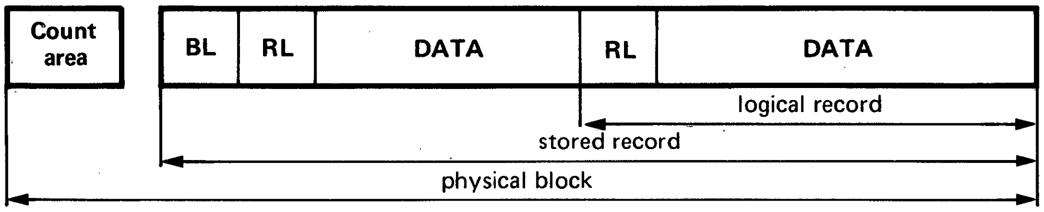
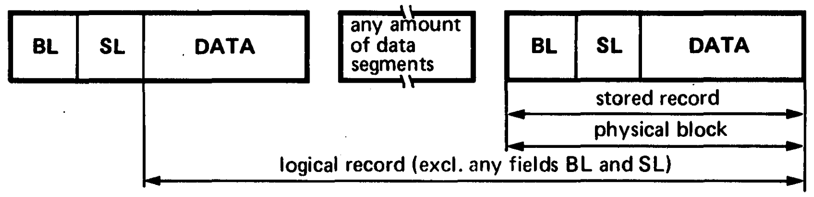


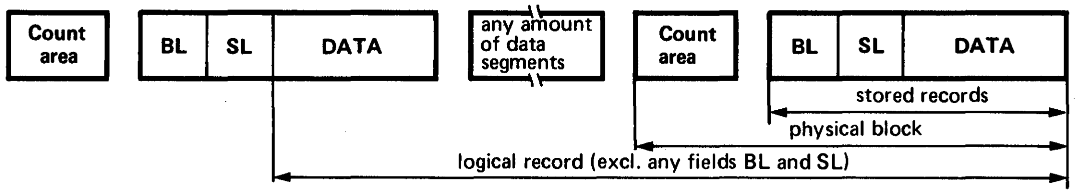
Figure 2.12. Record and block structures for the various devices (2 of 4)

UNBLOCKED, SPANNED RECORDS:

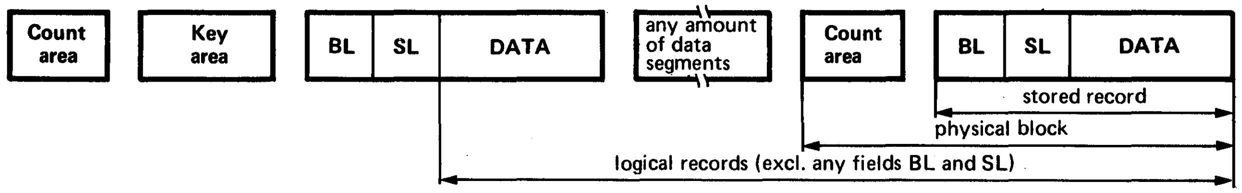
Magnetic tape


BL
SL
DATA
any amount of data segments
BL
SL
DATA
stored record
physical block
logical record (excl. any fields BL and SL)

Direct access storage devices, without a key area

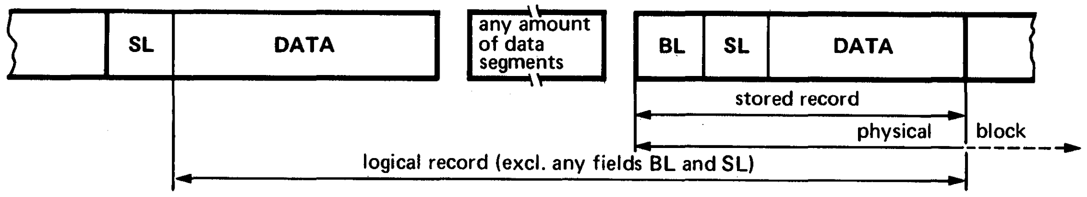

Count area
BL
SL
DATA
any amount of data segments
Count area
BL
SL
DATA
stored records
physical block
logical record (excl. any fields BL and SL)

Direct access storage devices, with a key area


Count area
Key area
BL
SL
DATA
any amount of data segments
Count area
BL
SL
DATA
stored record
physical block
logical records (excl. any fields BL and SL)

BLOCKED, SPANNED RECORDS:

Magnetic tape


SL
DATA
any amount of data segments
BL
SL
DATA
stored record
physical
block
logical record (excl. any fields BL and SL)

Direct access storage devices, without a key area

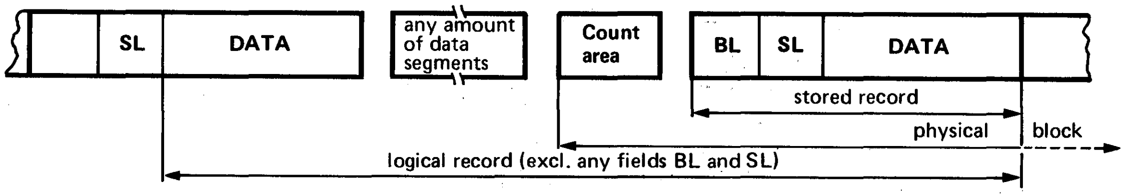
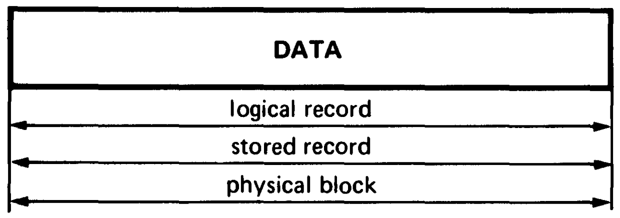

SL
DATA
any amount of data segments
Count area
BL
SL
DATA
stored record
physical
block
logical record (excl. any fields BL and SL)

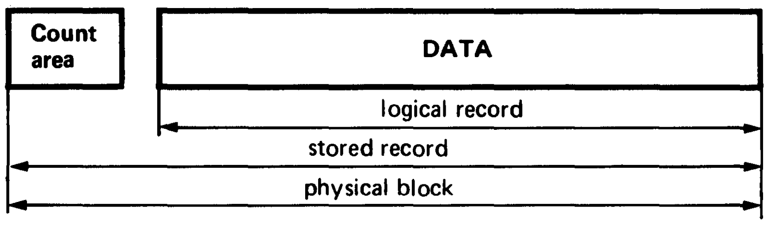
Figure 2.12. Record and block structures for the various devices (3 of 4)

UNDEFINED RECORDS:

Paper tape, MICR, OCR, Console, etc.



Direct access storage devices, without a key area



Direct access storage devices, with a key area

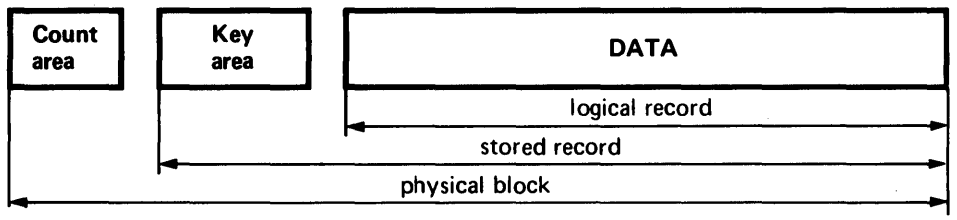


Figure 2.12. Record and block structures for the various devices (4 of 4)
A logical record is what is operated on by the user's program. A stored record is what is handled by the access method. A physical block is what is written on external storage.

Remarks regarding Access Methods:

Sequential-Access Method:
Records cannot be processed on DASD with a key area.

Direct-Access Method:
Only unblocked records can be processed.

Indexed-Sequential Access Method:
ISAM is the only access method that can process blocked records with a key.
No variable-length records can be processed.

Virtual Storage Access Method:
Only unblocked records, without a key area, are processed; the structure of VSAM records differs from those described here: see Section 4 of this manual.

File Structure

Many different types of data files are used in data processing applications. Theoretically, there is no restriction on the logical content of information that can be processed, on the relationships of various units of information in a file, or on the organization or format of the data. As long as the user's problem program includes the necessary processing capability and channel programs (provided by the data management routines of DOS/VS), the only restrictions are those imposed by the physical characteristics of the devices, or by the maximum capacity of the computer system.

We may talk about data files either logically or physically. When speaking about a logical file, we mean a collection of logical records that constitute the set of data being processed. *A logical file is a named,* *organized collection of logically related data.* When speaking of a physical file, we mean the physical characteristics of the data blocks as they appear on an external medium, such as block structure and physical sequence.

A logical file is reflected by the description of the data in one program, and this description normally applies to more than one physical file. For example, the description of logical file A in Figure 2.13 applies to different physical transaction files for different weeks.

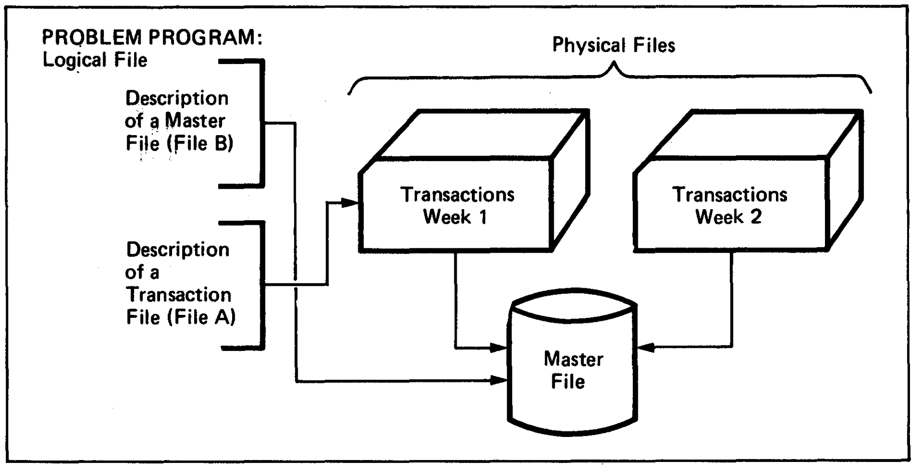


Figure 2.13. Logical files and physical files
A logical file is reflected by a descriptive framework that normally applies to more than one physical file.

Depending on the device or the operating system, files may be restricted only to a maximum size. At the other extreme, files may have to be completely fixed with regard to size, record format, logical and physical sequence, and, to a limited degree, logical content. Or intermediate combinations may be allowed.

A previous chapter discussed record formats acceptable to DOS/VS; this discussion was followed by a description of how these record formats are related to physical structures for certain types of I/O devices. It should be noted that, although DOS/VS may accept any of these record formats and block structures, the choice may be limited by the use of a certain part of the operating system (access method), or by a certain programming language. As for high-level programming languages, the reader is referred to the manuals of the language processor he is using for further specifications about the record formats and block structures that can be applied, for assembler language this information will be found in the chapter *Input/Output Control System* in this section of the manual, and in the section *Access Methods.*

Files and Volumes

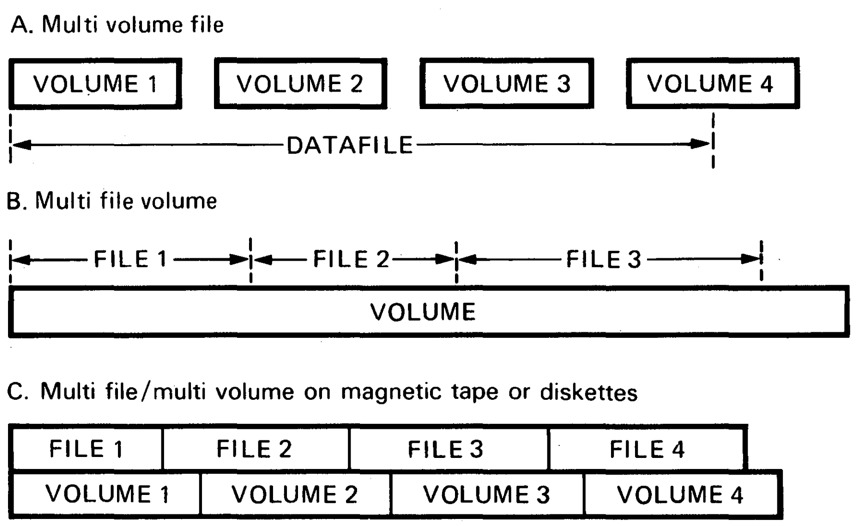
A recording medium that is mounted as a unit, like a reel of magnetic tape, a diskette, or a disk pack, is usually referred to as a volume. Data files are related to volumes in one of the following ways:

- one complete file, stored on one volume
- one complete file, stored on more than one volume (multi-volume file)
- more than one file, all completely stored on the same volume (multi-file volume)
- more than one file, all or some of them broken into *file segments*, and stored on more than one volume (multi-file/multi-volume)

These relationships are shown in Figure 2.14. The multi-volume file is often used in magnetic tape processing, diskette processing, and disk processing, simply because many files are so large that several volumes are required. The multi-file volume, although possible on magnetic tape, is mostly used on DASD and diskette volumes. This is because a file on a DASD or diskette volume can be located directly whereas a file on magnetic tape can be located only by scanning all files preceding it.

When a user has a number of large files that are processed randomly at different times (DASD), he may decide to use the multi-file/multi-volume concept shown in Figure 2.14D. The main advantage here is that the movements of the read-write heads are restricted to a limited number of cylinders within each volume, resulting in more efficient processing.

The user knows which volumes contain his files, and this knowledge is supplied to DOS/VS through job control cards that precede his object program (see *File labeling*). His program includes a set of label processing routines that make use of this information to check whether the proper volume is mounted and to locate the file on that volume. Whenever a volume must be mounted, the console operator receives a message on the console printer-keyboard (SYSLOG).



D. Multi file/multi volume on DASD or diskettes

File 1 part 1	File 1 part 2	File 1 part 3	File 1 part 4	
File 2 part 1	File 2 part 2	File 2 part 3		
File 3 part 1	File 3 part 2	File 3 part 3	File 3 part 4	File 3 part 5
VOLUME 1	VOLUME 2	VOLUME 3	VOLUME 4	VOLUME 5

Since there is space left on volume 4, one disk drive can be saved by allowing one more part of File 3 to be stored on volume 4. If possible we may decide to have that part of the file that is least active stored in the largest area available. It is not at all important where the parts of the files are stored on the volumes. Assuming that part 2 and part 5 of File 3 are less active than the other parts, the organization could very well be as follows (not applicable for diskettes):

File 2 part 1	File 2 part 2	File 2 part 3	File 1 part 1
File 3 part 1	File 3 part 3	File 1 part 3	File 3 part 2
File 1 part 4	File 1 part 2	File 3 part 4	File 3 part 5
VOLUME 1	VOLUME 2	VOLUME 3	VOLUME 4

Figure 2.14. Files and volumes
Multi-volume files may be used for magnetic tape, diskette, or disk files, whereas multi-file volumes are usually used for disk or diskette volumes only. Multi-file/multi-volume may also be used on magnetic tape, diskette, and disk. On disk, parts of files may be placed wherever convenient.

There is not much difference between the processing of magnetic tape and of, for instance, punched card. Both types of processing are serial, which means that records are encountered in the sequence in which they occur in the file. Magnetic tape has the advantage that the size of data blocks may vary between a few and many thousands of characters, with a maximum of 32K bytes. Also, there are no specific boundaries needed for magnetic tape blocks. Blocks are separated simply by *inter-record gaps*, which are empty spaces between the blocks. An inter-record gap may occur anywhere on the surface of magnetic tape.

Because the physical characteristics of DASD or diskette are somewhat different from those of magnetic tape, a DASD or diskette volume has certain boundaries that need to be kept in mind.

Cylinder Concept

A disk volume is constructed of a number of disks, one above the other. Each disk has two surfaces that may be written with data, except the top and bottom disks that have only one writing surface. Each disk in a volume contains a number of tracks of a fixed data capacity. Corresponding tracks on all surfaces are located one above the other, and may be pictured as forming a number of concentric cylinders, each of them containing a number of tracks. For example, as is shown in Figure 2.15, the IBM 3330 disk storage drives are accessing volumes that contain 411 cylinders (000 - 410) and each cylinder contains 19 tracks (00 - 18), so that one disk pack has a total capacity of 7,819 tracks.

Each surface is accessed by one read-write head, that can be moved from one cylinder to another. Tracks are numbered consecutively from top to bottom per cylinder, so that the read-write head mechanism needs to move only when the track to be accessed is in a different cylinder.

The above characteristics are different for each type of DASD; detailed information is supplied in Appendix 1.

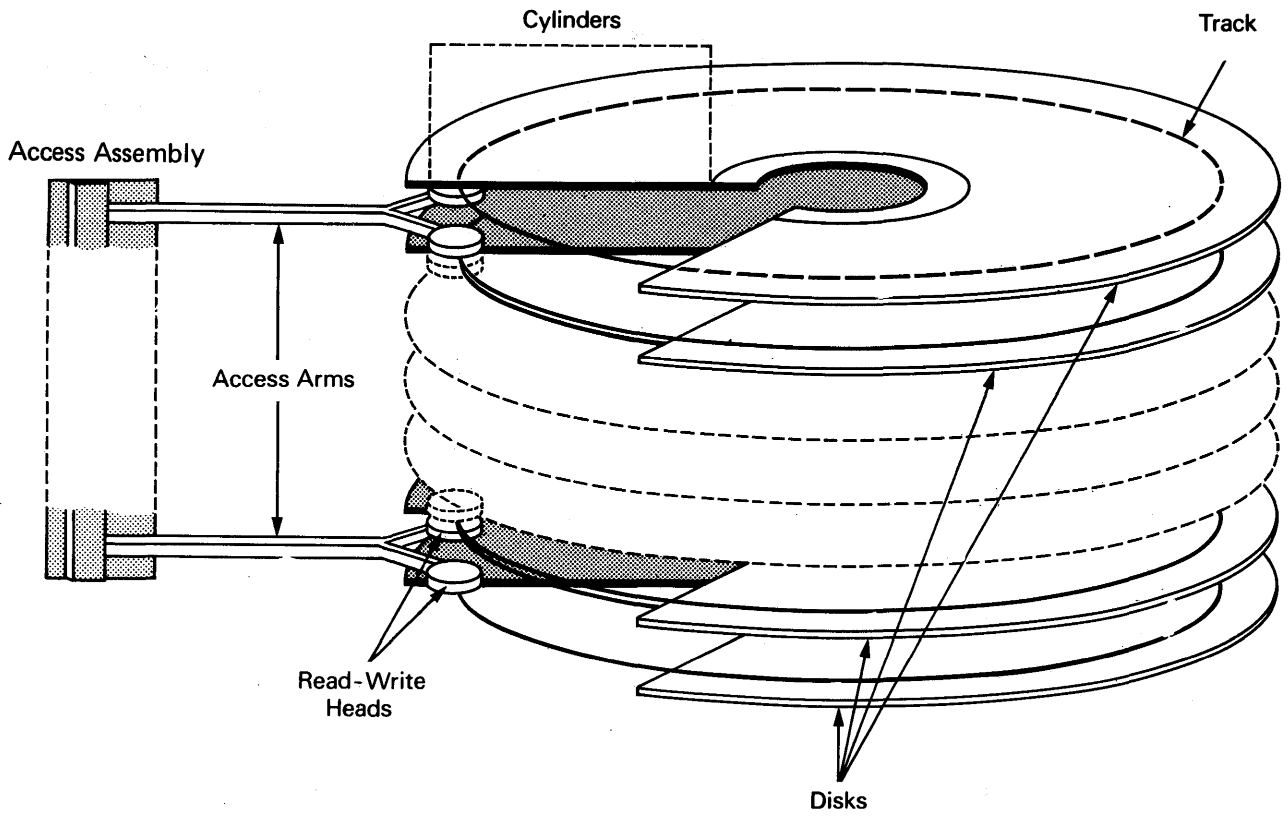


Figure 2.15. Cylinder concept, and access mechanism of disk
All access arms move together as a unit so that all tracks in one cylinder may be read at one positioning of the read-write head mechanism.

The IBM 3330 volumes (IBM 3336 disk packs) have a capacity of 411 cylinders. However, not all of them are available for direct use; certain cylinders are reserved as an alternate area, and are used only if some track in the remaining cylinders becomes defective. In that case a track in the alternate area is made an alternate track for the defective one. The alternate area ensures that the stated capacity of a disk pack (404 cylinders on the 3336 disk pack) can be maintained for the life of the pack. In the example of a 3336 disk pack, seven cylinders out of the total of 411 are reserved as alternate area (133 tracks). These figures are different for each type of DASD volume.

More information about alternate tracks and their use is supplied in the chapter *DASD initialization and maintenance.*

Track Capacity -- DASD

Because each physical block has certain non-data areas (such as count field and gaps), the net data capacity of a track varies with the number of records that must be located on a track (see: Appendix 2). The largest capacity would be obtained with one physical block per track. This illustrates that it is frequently useful to have logical records blocked to the nearest maximum blocksize, in order to make the most effective use of the potential external storage capacity. Appendix 2 lists the track capacities related to the amount of physical blocks per track for all DASD supported under DOS/VS.

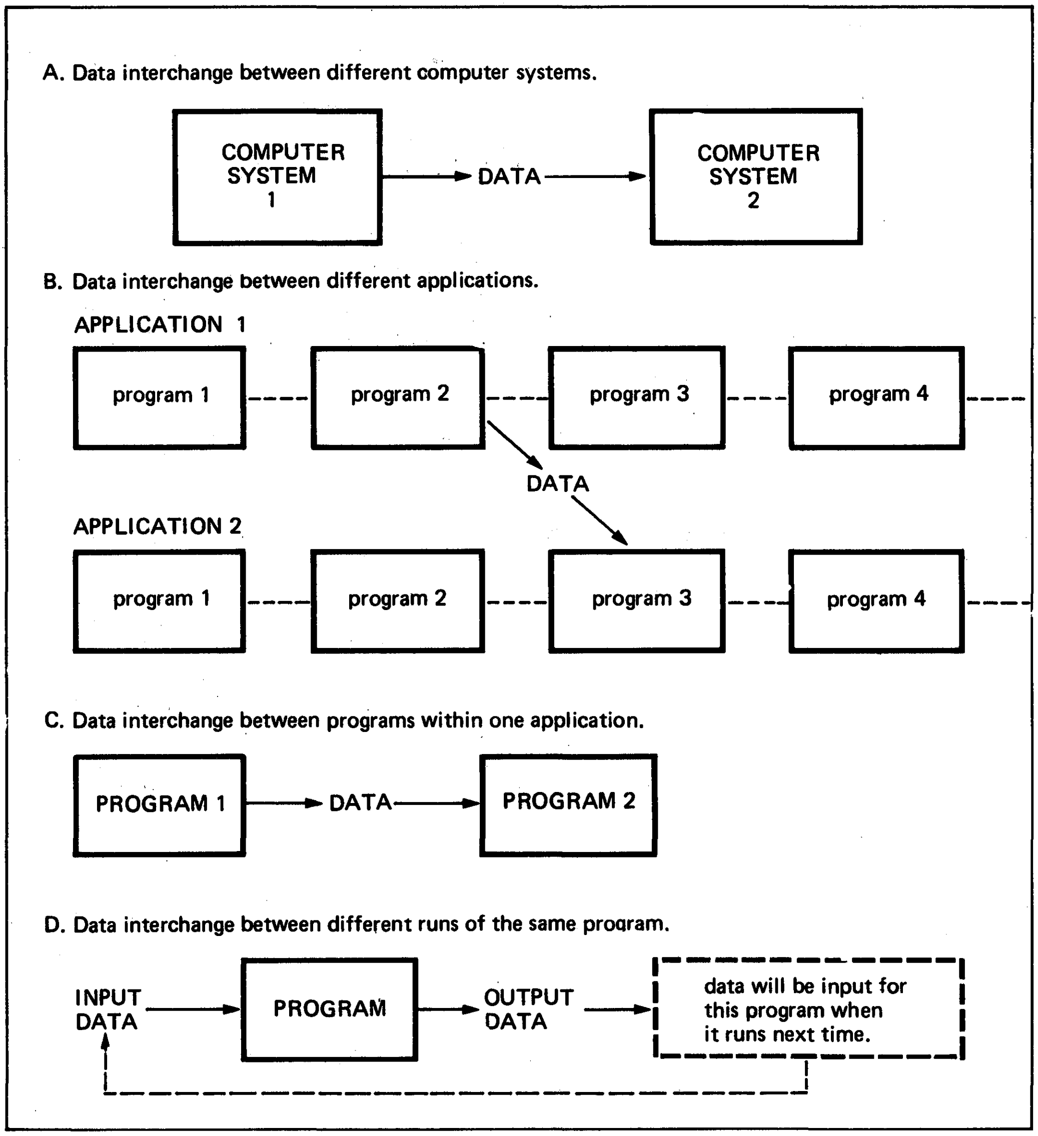
Track Capacity -- Diskette

A diskette volume has a single surface on which data may be written. This surface contains 77 tracks, each of which has a fixed data capacity. However, not all of these tracks are available for direct use. When the diskette is initialized, 74 of the tracks are available for use (track 0 for the VTOC and tracks 1-73 for data). Track 74 is not used and tracks 75 and 76 are alternate tracks.

Each track is in turn devided into 26 records, each 128 bytes in length. To make the most effective use of a diskette's potential external storage capacity, logical records of 128 bytes in length should be used. See Appendix 1, Figure 5.2 for a discussion of diskette device characteristics.

File Labeling

Data that is stored on an external storage device may be used for data interchange between different applications or even between different computer systems. Data can also be interchanged between different programs of the same application, or between different runs of the same program. An exception is a workfile, created and used by the same program in the same job. In all other cases. some time will elapse between the moment a file is created and the moment the same file is used again for input.



A. Data interchange between different computer systems.
COMPUTER SYSTEM 1
DATA
COMPUTER SYSTEM 2
B. Data interchange between different applications.
APPLICATION 1
program 1
program 2
program 3
program 4
DATA
APPLICATION 2
program 1
program 2
program 3
program 4
C. Data interchange between programs within one application.
PROGRAM 1
DATA
PROGRAM 2
D. Data interchange between different runs of the same program.
INPUT DATA
PROGRAM
OUTPUT DATA
data will be input for this program when it runs next time.

Figure 2.16. Data Interchange
Because data is often subject to multiple uses, it needs to be labelled for identification and protection.

It is obviously important that data files be safeguarded against unauthorized usage or destruction. Also it is very important that a file can be located and recognized when it is needed. File labeling is designed to enable the data management routines to protect and recognize files that are written on magnetic tape, diskette, or DASD.

The user is expected to know which volumes of magnetic tape, diskette, or direct access storage contain his data file. He enters this knowledge on job control cards before he creates a file, or uses it for processing. These control cards, containing information such as file name and volume serial number, are embedded in the input for a job, and read before the object program that uses the file is loaded. job control selects these control cards from the job stream and stores the information in the *label information cylinder*, a file on SYSRES that is owned and maintained by DOS/VS. Later, when the object program has been loaded and asks for a file on tape, diskette, or DASD, the DOS/VS data management routines retrieve this information from the label information cylinder, and notify the console operator which volumes are requested. After the operator has mounted these volumes, the DOS/VS label handling routines start the processing of labels. This sequence of reading job control cards and processing of labels is illustrated in Figure 2.17.

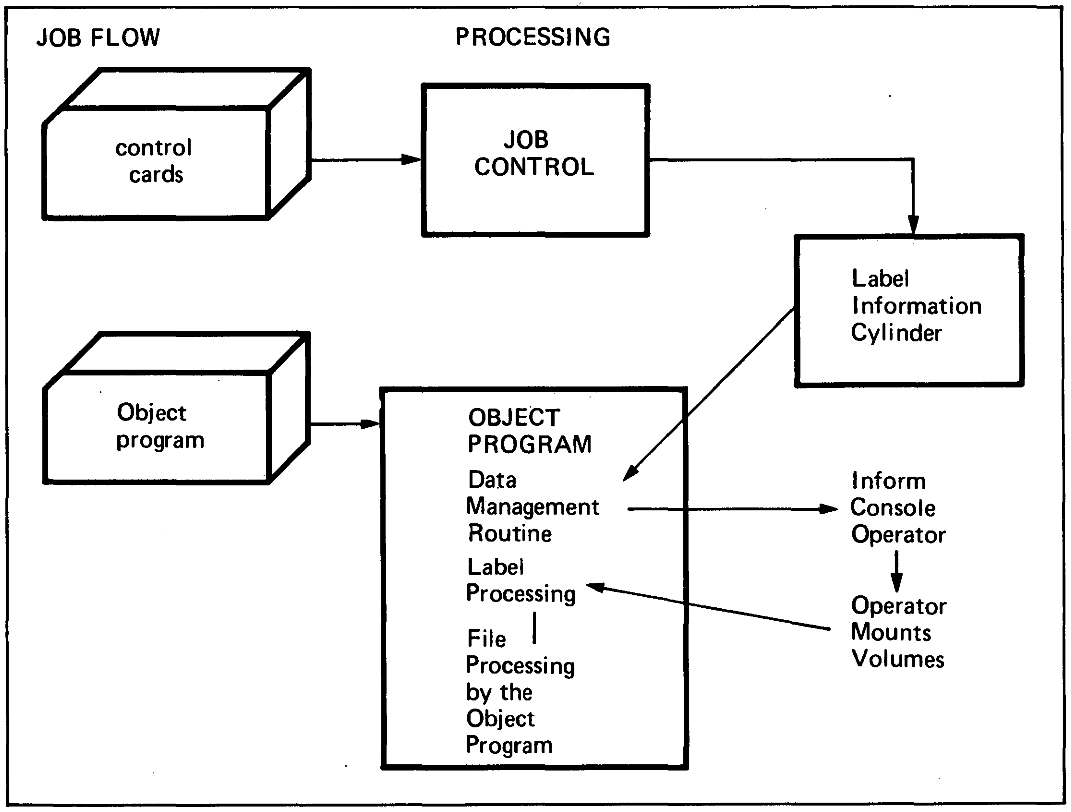


Figure 2.17. The function of Job Control in labeling
Job Control reads user-supplied information for identifying a file and stores it on SYSRES for later use by data management routines.

For examples of the format of job control cards in general, see *DOS/VS System Management Guide*, GC33-5371; detailed information about the format of job control cards will be presented in *DOS/VS System Control Statements*, GC33-5376. In addition, the format and the use of file labeling job control cards will also be presented in the publications:

- *DOS/VS Tape Labels*, GC33-5374.
- *DOS/VS DASD Labels*, GC33-5375.

File labeling in DOS/VS is conceptually the same for magnetic tape, diskette, and DASD; physically, however, it differs according to the media.

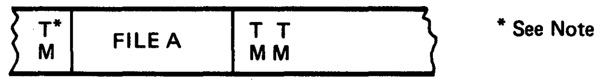
The following chapters describe the most important facts about magnetic tape labeling, diskette labeling, and DASD labeling under DOS/VS; detailed information can be found in the two labeling manuals listed above.

Magnetic Tape Labeling

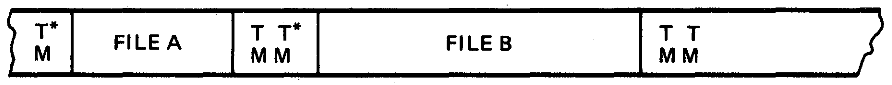
A magnetic tape to be processed under DOS/VS must conform to certain standards. These standards pertain to labels, placement of tapemarks, and the grouping (blocking) of records. Record blocking was discussed earlier under *File structure*; this chapter discusses tape labels and the placement of tapemarks.

Magnetic tape labels can be processed with or without labels. *Unlabeled tapes* do not contain labels; these files are delimited by tapemarks only:

A. Single file/volume:



B. Multifile/volume:



Note: A leading tapemark before the first data block of a file will always be written unless the program specifies otherwise. If it is specified that no tapemark must be written before the first data block of a file, a tapemark will follow the file only, and a double tapemark will occur at the end of a volume only.

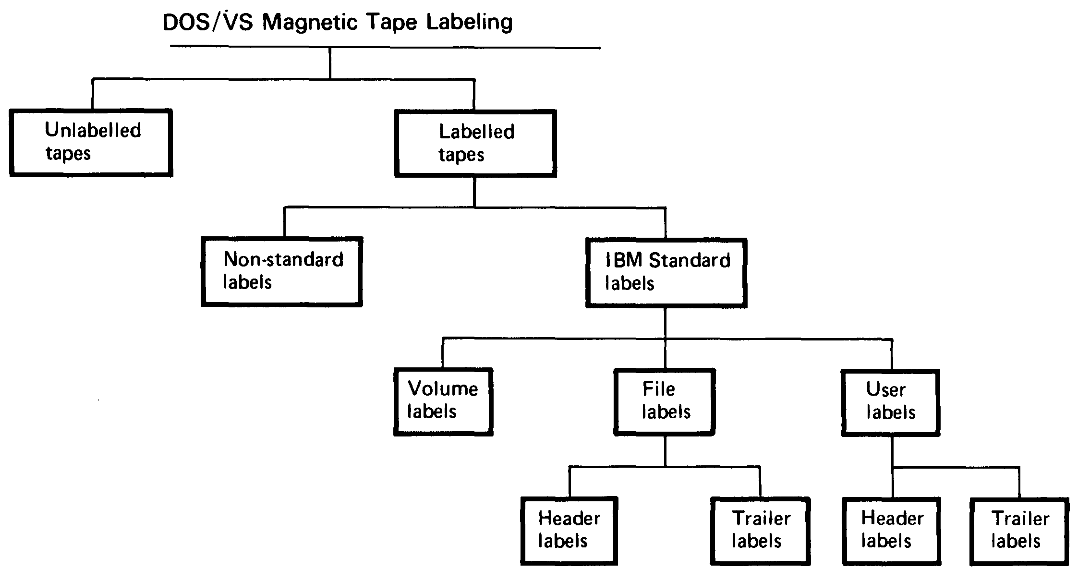
If labels are present, they can be:

- IBM-standard
- nonstandard

This chapter will discuss IBM-standard labels only; information about the processing of nonstandard labels can be obtained from the DOS/VS Tape Labels manual listed above.

The IBM-standard label set for magnetic tape labeling includes the following label types:

- volume labels
- file labels
- standard user labels



Standard labels have a size of 80 bytes, and label types are identified by a four-byte *Label Identifier Field.* The formats of the IBM-standard volume label and file labels are illustrated in Appendix 3 of this manual. It should be noted that, for standard labeling, only one volume label is required per volume, and only one header label and one trailer label are required per file. All other labels are optional and their use may even be prohibited for high-level programming languages. For information regarding label processing under a high-level programming language (COBOL, PL/I, FORTRAN, RPG), the user should consult the appropriate sections in the manuals for the language processor he is using.

Volume Labels

Whenever standard labeling is used on magnetic tape, the first block on the reel is the required standard volume label. The standard volume label is meant to uniquely identify one particular reel of magnetic tape (volume), and contains the volume serial number. Usually, this number is also placed on the outside surface of the reel for visual identification.

When standard labeling is specified for an output file, the standard volume label is assumed to be present. It is usually written by an IBM-supplied system utility program on each new, unused reel of the installation. Label handling routines of DOS/VS check the volume label and, if no standard volume label is present on an output tape, a diagnostic message is issued. The console operator then has the option of entering the volume serial number, so that the volume serial number can then be written.

The standard volume label is identified by the contents *VOL1* in the label identifier field. (See also *Additional labels.*)

File Labels
Standard file labels are used to uniquely identify one particular file on one reel of magnetic tape. Basically there are two types of file labels:

- File *header* labels: precede a whole file and, for a multi-volume file, each section which is contained on any one volume.
- File *trailer* labels: follow a file or each file section. A file always ends with an end-of-file label. On a multi-volume file, each section except the last is followed by an end-of-volume label.

For example:

1. A single-volume file:

FILE HEADER LABEL	FILE A	END–OF– FILE LABEL

2. A multi-volume file (two volumes) :

FILE HEADER LABEL	SECTION 1 OF FILE A	END–OF– VOLUME LABEL

FILE HEADER LABEL	SECTION 2 OF FILE A	END–OF– FILE LABEL

File Header Labels
The file header label is identified by the contents *HDR1* in the label identifier field. (See also *Additional labels.*) It contains a file identifier field, a volume sequence number to indicate the proper sequence on a multi-volume file, and a file sequence number to indicate the proper sequence on a multi-file reel. It also contains the volume serial number which is copied from the standard volume label of the first reel occupied by the file.
On a multi-volume file, the HDR1 label of the first reel is copied to each subsequent reel, with the volume sequence number incremented by one.

File Trailer Labels
The end-of-file label is identified by the contents *EOF1* in the label identifier field. (See also *Additional labels.*) A file is always terminated by an end-of-file label. The end-of-volume label is identified by the contents *EOV1* in the label identifier field. (See also (*Additional labels.*) It follows each section of a file except for the last. Both types of labels contain a block count field for the number of data blocks written between the last file header label and the file trailer label. Except for the block count field and the label identifier, the file trailer labels are a copy of the file header label (see Appendix 3).

Additional Labels

When standard labeling is used, the VOL1, HDR1, EOV1 (multi-volume files only), and EOF1 labels are required. Moreover, the user has the option of writing the following additional labels:

- Up to seven additional volume labels, identified by the contents *VOL2* ... *VOL8* in the label identifier field.
- Up to seven additional file header labels, identified by the contents *HDR2* ... *HDR8* in the label identifier field.
- Up to seven additional end-of-file labels, identified by the contents *EOF2* ... *EOF8* in the label identifier field.
- Up to seven additional end-of-volume labels, identified by the contents *EOV2* ... *EOV8* in the label identifier field.

The additional labels may contain any information required by the user. However, it should be noted that additional labels, when found on an input tape, are bypassed by the DOS/VS label processing routines.

A collection of labels of the same type (for example, *HDR1* ... *HDR8*) is called a *label set.* A header label set, for instance, may consist of the required HDR1 label only, or of the HDR1 label plus a number of additional HDRn labels.

Standard User Labels

To further define his file, the user can include standard user labels in addition to the required and optional standard file labels just described.

As many as eight standard user header labels (*UHL1* ... *UHL8*), and eight standard user trailer labels (*UTL1* ... *UTL8*) may be written.

A standard user header label set (if used) always follows the standard file header label set. The two sets together constitute a *file header label group.* A standard user trailer label set (if used) always follows the standard file trailer label set (EOV or EOF). The two sets together constitute a *file trailer label group.*

Thus, a file header label group may consist of the required HDR1 label only, or a file header label set only (HDR1 ... HDRn), or a file header label set plus a user header label set (UHL1 ... UHLn). And a trailer label group may consist of the required EOV1/EOF1 label only, or a file trailer label set only (EOV1/EOF1 ... EOVn/EOFn), or a file trailer label set plus a user trailer label set (UTL1 ... UTLn).

Placement of Tapemarks

DOS/VS distinguishes between file labels and data by means of a special one-character block, called a *tapemark.* The placement of tapemarks is normally of no concern to the user.

As a general rule, every label group , except the volume label set/group, is followed by a tapemark. The last label group on a reel is followed by two tapemarks. A summary of standard magnetic tape labeling is given in Figure 2.18.

1. Standard VOLUME label set consists of:

   VOL1 label (required)
   VOL2 . . .VOL8 labels (optional)

2. Standard FILE HEADER label set consists of:

   HDR1 label (required)
   HDR2 . . . HDR9 (optional)

3. Standard FILE TRAILER label set consists of:

   EOV 1, or EOF1 (required)
   EOV2 . . . EOV9, or EOF2 . . .EOF9 (optional)

4. Standard USER HEADER label set consists of:

   UHL1 . . . UHL8 (optional)

5. Standard USER TRAILER label set consists of:

   UTL1 . . . UTL8 (optional)

6. A FILE HEADER LABEL GROUP MAY CONSIST OF A FILE HEADER LABEL SET (2) PLUS A USER HEADER LABEL SET (4).

7. A FILE TRAILER LABEL GROUP MAY CONSIST OF A FILE TRAILER LABEL SET (3) PLUS A USER TRAILER LABEL SET (5).

8. EXAMPLES OF LABELING CONFIGURATIONS:

A. One single file, on one single volume:

VOLUME LABEL SET	FILE HEADER LABEL GROUP	TM	DATA BLOCKS OF FILE A	TM	FILE TRAILER LABEL GROUP	TM	TM

B. Multi volume file:

All volumes except last one (EOV labels)

VOLUME LABEL SET	FILE HEADER LABEL GROUP	TM	FILE SECTION OF FILE A	TM	FILE TRAILER LABEL GROUP	TM	TM

Last volume (EOF labels).:

VOLUME LABEL SET	FILE HEADER LABEL GROUP	TM	LAST SECTION OF FILE A	TM	FILE TRAILER LABEL GROUP	TM	TM

C. Multi file volume :

VOLUME LABEL SET	FILE HEADER LABEL GROUP	TM	FILE A	TM	FILE TRAILER LABEL GROUP	TM	FILE HEADER LABEL GROUP	TM	FILE B	TM

Figure 2.18. Summary of standard magnetic tape labeling under DOS/VS
The processing of optional labels is the responsibility of the user.

DASD Labeling

DOS/VS provides positive identification and protection of DASD files by recording labels on each volume (see also *Data security/data integrity*).

The IBM-standard label set includes volume labels and file labels. The format of these labels is shown in Appendix 3; they are described below. Provision is made for additional user volume labels and user file labels; user file labels, however, are not processed by the DOS/VS label handling routines. They are recognized by the DOS/VS label handling routines but must be processed by user-written label handling routines as part of the problem program.

DASD Volume Labels

Each volume in an installation must have an IBM-standard volume label which uniquely identifies a volume. It is always located on cylinder 0, track 0, record 3. Provision is made for having up to seven additional volume labels that immediately follow the required volume label.

Volume labels are identified by the contents *VOL1* ... *VOL8* in the label identifier field, which is the first four bytes of the data portion of the label.

The volume label contains the volume serial number, and a pointer (DASD address) to the Volume Table of Contents (see below). It is written on a volume by an IBM-supplied utility program when the volume enters an installation, and remains the same for its entire life in the installation. (See *DASD initialization and maintenance*).

DASD File Labels

For all files (or parts of a file) on a DASD volume, file labels are stored in a separate area on the same volume. This area is called the *Volume Table Of Contents* (VTOC). The VTOC is essentially a directory of all data blocks on a DASD volume.

DASD files can be written in one or more continuous areas called *extents*, and the boundaries of each file extent are maintained in the file labels in the VTOC. Figure 2.19 shows the fundamental concept of DASD file labeling through a VTOC.

The number of file labels for a file depends on the type of the file, the number of extents that constitute the file, the organization of the data in the file, and the number of volumes containing parts of the file. IBM-standard DASD file labels are written in a 140-byte format: 44 bytes of key area and 96 bytes of data. The first four bytes of the data portion of a label make up the label identifier field.

For the purpose of identifying and locating different types of data files, three different formats of DASD file labels are available. In addition, there is another label format which is used as a label for the VTOC itself. Thus, the VTOC is considered to be just another file, addressed through a pointer in the volume label, and accessed and maintained by DOS/VS only.

The complete range of IBM-standard DASD file labels consists of five label formats; the format 5 DASD label, however, is not used by DOS/VS.

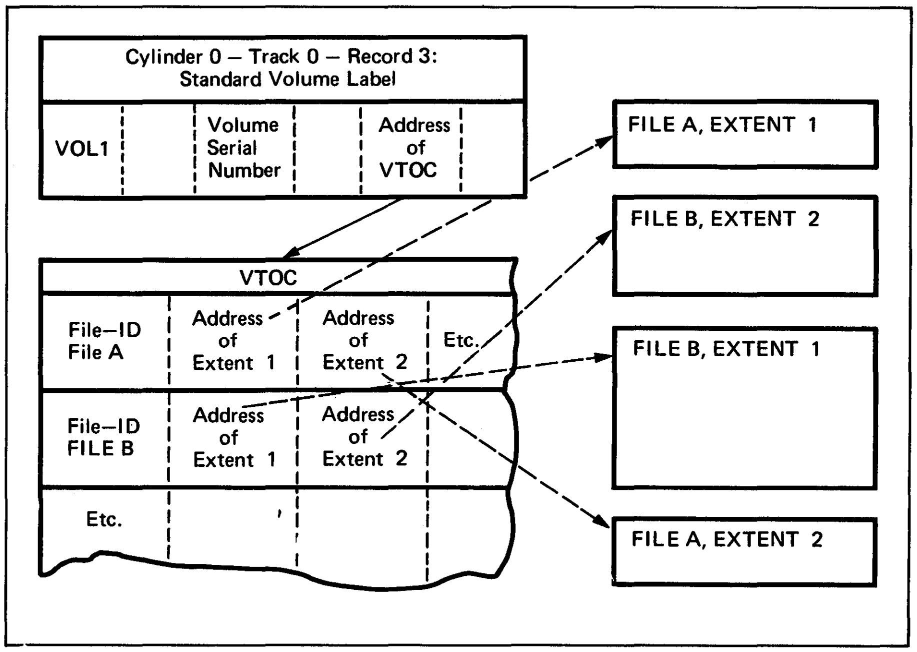


Figure 2.19. The fundamental concept of DASD labels in a VTOC
The VTOC lists the labels of all files or parts of files contained in the same volume.

The Five DASD Label Formats (see Appendix 3)

- Format 1 file label.
  This label is used for all files on DASD, regardless of type, number of extents, or organization. It contains the file identification and the specifications of the file and its records. Each volume of a file contains a format 1 label. It provides for the specification of three extents on a volume. If more than three extents on a volume are used for the file, these additional extents are specified in one or more subsequent format 3 labels (see below).

- Format 2 file label.
  The format 2 file label is only required for any file that is maintained by the Indexed Sequential Access Method (see Section 4, *Access Methods*). The 44-byte key area, although present, is not used by DOS/VS. The 96-byte data portion contains additional information about the file unique for this type of organization, such as size and location of indexes, data area, and overflow areas. If an ISAM file is written over more than one volume, this additional format 2 label is written on the first volume only.

- Format 3 file label.
  The format 3 file label specifies any additional file extents on a volume in excess of the first three extents, which are specified by the format 1 label. Each format 3 label can specify thirteen extents; any number of format 3 labels can be used.

- Format 4 file label.
  The format 4 label describes the VTOC. It is always the first label in the VTOC. In addition, this label provides the location and number of

available tracks in the alternate track area (see *Files and volumes, Cylinder concept*).

- Format 5 file label.
  When the VTOC is preformatted (see *DASD initialization and maintenance*), the second label in the VTOC is reserved as the format 5 file label. It is not used, however, by DOS/VS.

DASD User Header and Trailer Labels
The user can include additional labels to further define his file. Such labels must be processed by the user in his problem program. The DOS/VS Input/Output Control System allows up to eight additional user header and trailer labels on disk, and five on the IBM 2321 Data Cell. There are certain restrictions:

- A file to be processed with physical IOCS macro instructions (see *Input/Output Control System*) cannot contain user trailer labels.
- The Indexed-Sequential Access Method (see *Access Methods*) makes no provision for any user labels.

User header and trailer labels are not placed in the VTOC. Instead, they are written on the first track of the first extent allocated by the user for the logical file. If a file is written on more than one volume, this label track is reserved in the first extent of the file on each volume. The user's label track is defined by IOCS as a separate extent, invisible to the user.

User labels must be 80 bytes in length. The first four bytes are the label identifier field, the remaining 76 bytes may contain whatever information a user requires. User labels are preceded by a four-byte key field. Header labels are identified by *UHL1 ... UHL8* in both key field and label; trailer labels are identified by *UTL0 ... UTL7* in the key field, and by *UTL1 ... UTL8* in the label. Each header or trailer label set is terminated by an end-of-file record, which is a record with a *data* length zero. For example, if a file has five user header labels and four user trailer labels, the user label track contains the following:

R0	Standard information, see description of record 0 under *Record Structures for the Various Devices.*	
R1	UHL1	UHL1, user's 1st header label
R2	UHL2	UHL2, user's 2nd header label
R3	UHL3	UHL3, user's 3rd header label
R4	UHL4	UHL4, user's 4th header label
R5	UHL5	UHL5, user's 5th header label
R6	UHL6	(end-of-file record)
R7	UTL0	UTL1, user's 1st trailer label
R8	UTL1	UTL2, user's 2nd trailer label
R9	UTL2	UTL3, user's 3rd trailer label
R10	UTL3	UTL4, user's 4th trailer label
R11	UTL4	(end-of-file record)

For more detailed information on DASD user labels the reader is referred to the DASD labels manual, listed in the introduction to this chapter.

Labeling of VSAM Files

Labeling for VSAM files (see "Virtual Storage Access Method" in section 4 "Access Methods") is totally different from the labeling scheme above. VSAM uses one DASD area over one or more volumes, that is used for an arbitrary number of separate logical data files. Although this DASD area as a whole fits in the labeling scheme described in this section, individual VSAM files in this area are identified, located, and maintained through a VSAM catalog which is located on SYSCAT.

Diskette Labeling

DOS/VS provides positive identification and protection of diskette files by recording labels on each volume (see also *Diskette Security/Integrity*).

The IBM-standard label set includes labels and file labels. The form of these labels is shown in Appendix 3; they are described below. User labels are not supported for diskette devices.

Diskette Volume Labels

Each diskette volume in an installation must have an IBM-standard volume label which uniquely identifies that volume. It is always located on track 0, record 7.

Volume labels are identified by the contents VOL1 in the label identifier field, which is the first four bytes of the data portion of the label.

The volume label contains the volume serial number, an accessibility indicator (AI) and a standard label level indicator (SLL).

Diskette File Labels

For all files (or parts of a file) on a diskette volume, file labels are stored in a separate area on the same volume. This area is called the *Volume Table of Contents* (VTOC). The VTOC is essentially a directory of all files on a diskette volume and is located on track 0, records 8 through 26.

Diskette files can be written in one or more continuous areas called extents; the boundaries of each file extent are maintained in the file labels in the VTOC. Each file may have only one extent per volume. Figure 2.20 shows the fundamental concept of diskette file labeling through a VTOC.

The number of file labels for a file depends on the number of extents (volumes) that constitute the file.

IBM-standard diskette file labels are 80 bytes in length. The first four bytes of the label make up the label identifier field. A single file label format, identified by HDR1, is supported by DOS/VS. It contains the file identification, and the specifications of the file and its records. Each volume of a file has a HDR1 label associated with it.

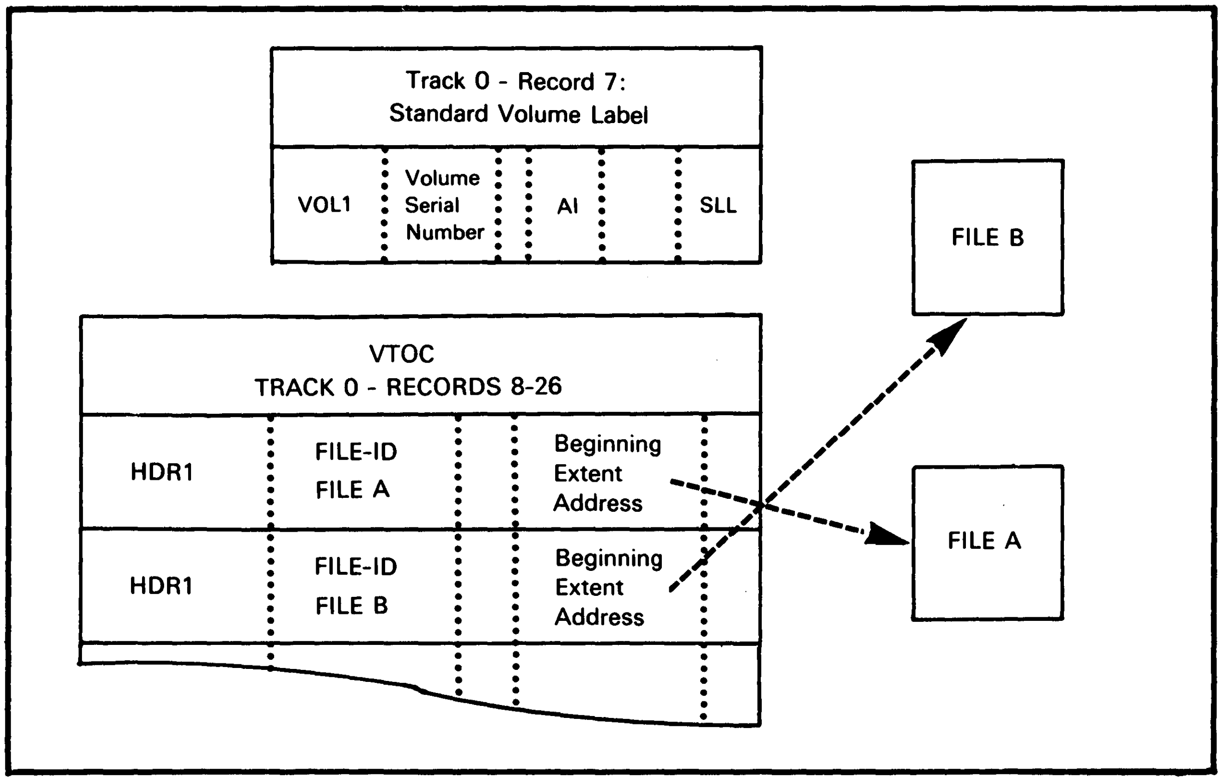


Figure 2.20 The fundamental concepts of diskette labels in a VTOC. The VTOC lists the labels of all the files or parts of a file contained on the volume.

Label Processing

Standard labels are processed by the transient label handling routines of the DOS/VS Input/Output Control System (IOCS); user labels and nonstandard labels are processed by user-written label handling routines which are part of the problem program. Detailed information about this subject is given in the DOS/VS Tape Labels and DASD Labels manuals.

To process standard labels, the user need only provide the appropriate job control cards; he need not be concerned with the actual process.

Physical Devices and Symbolic Device Names

Input/output devices are addressed through DOS/VS data management via physical channel and device addresses. In his problem program, a user can refer to an I/O device by means of symbolic device names. This offers the user a great deal of flexibility, because he can thereby change device numbers from run to run without modifying the object program.

As will be explained later, some symbolic device names even allow a programmer to assume a specific device type when creating a program, although, when the program is actually running, another device type is used.

DOS/VS translates the symbolic device name into a physical channel and device address. This is shown in Figure 2.21.

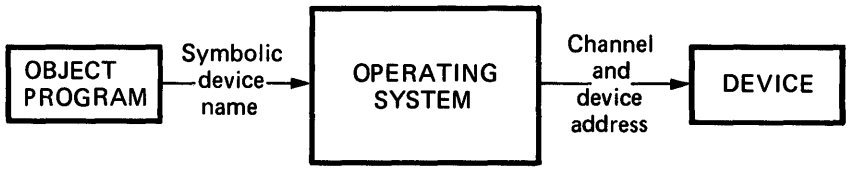


Figure 2.21. Symbolic device addressing
By avoiding the direct use of physical device names in his problem program, the user is not restricted to particular devices when actually running his program.

DOS/VS uses two tables which are part of the resident supervisor in lower main storage. The first table, called the *Logical Unit Block table* (LUB), acts as an interface between the user's problem program and DOS/VS and contains a list of all symbolic device names that are available in the system. Each of the symbolic device names in the LUB table is connected by means of a pointer to some element in the second table, which is called the *Physical Unit Block table* (PUB). The PUB table contains a list of channel and device addresses of all devices that are physically available in the system, and indicates such status information as:

- the device is operative or not;
- the device is performing an I/O operation or not;
- the device is assigned to a symbolic device name or not.

Before a program starts running, the proper connection between a symbolic device name and a physical device must be established. This means that (Figure 2.22) the pointer from LUB table (symbolic device name) to PUB table (actual device address) must be set by the job control program. The user indicates the proper connection to job control by means of the ASSGN job control statement or command.

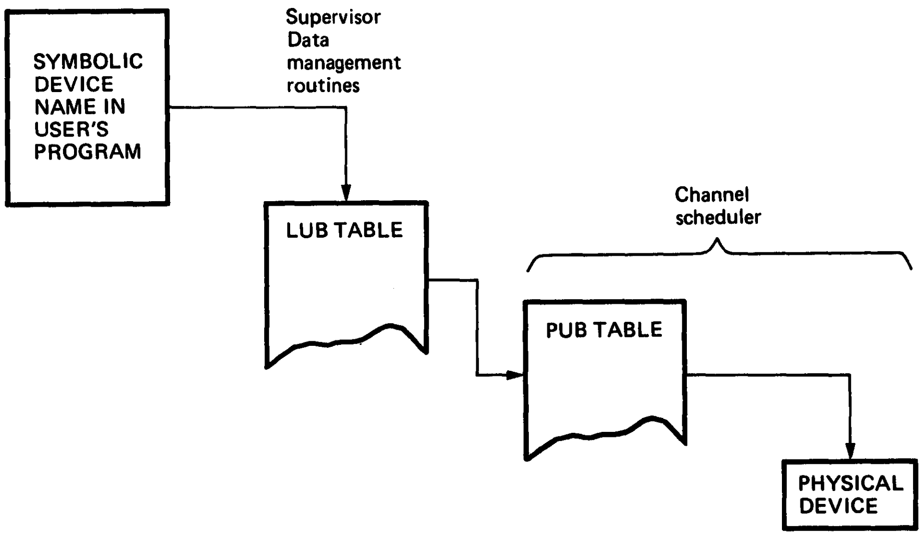


Figure 2.22. Relationship between symbolic device name and physical device, via LUB and PUB tables
The ASSGN job control statement sets the pointer from LUB table to PUB table.

Symbolic Device Name Format

A fixed set of symbolic names is used in an installation to refer to I/O devices. No other names can be used. All symbolic device names have the format *SYSxxx*, where *xxx* can be either numeric or alphabetic.

Detailed information about symbolic device names and job control is given in *DOS/VS System Management Guide*, GC33-5371, and *DOS/VS System Control Statements*, GC33-5376.

Multiprogramming Considerations

In multiprogramming, there are special problems associated with data management:

- Programs in different partitions may attempt to address:

    1. the same physical device, using different symbolic device names;
    2. different physical devices, using the same symbolic device name;
    3. the same physical device, using the same symbolic device name.

- Programs in different partitions may attempt to access the same DASD file.

The system solves the problem of maintaining a unique relation between symbolic device names and physical devices by applying the following rules:

- A non-DASD device can be assigned to one partition only.
- If assigned to SYSLST or SYSPCH, a tape unit cannot be assigned to any other logical unit.

Violations of these rules result in an error message to the console operator, who then can take corrective action.

Device Independence

Symbolic device addressing gives the user a great deal of flexibility in writing programs. He may write his programs as if a certain device were always available. When the program is actually run and the device happens not to be available, the symbolic device name can be assigned easily to some other device. In some cases, this other device may even be of a different type. This device independence is especially important in a multiprogramming environment.

For example, each of five programs in virtual storage may assume that it is reading punched cards from a card reader. If only one card reader is in the installation, however, four programs may obtain their card-image input from some other device (magnetic tape or disk) while only one is actually using the card reader. Of course, if a program reads card-image input from a device which is not a card reader, the actual card input must be written on that other device before it is used for input.

Punched card output, or printed output, may be treated the other way around: instead of data being actually transferred to a card punch or printer, it may be written in a card- or printline-image format, to disk or magnetic tape, and actually punched or printed later.

The POWER Program

POWER (Priority Output Writers, Execution processors, and input Readers) system control programming offers facilities for improving system performance, and is designed to reduce CPU dependence on I/O unit record devices by using intermediate storage on disk for input and output, thus providing unit record device independence.

Full information on POWER is presented in the *DOS/VS System Management Guide*, GC33-5371.

Checkpoint / Restart

A number of causes, such as program errors and machine or power failures, may cause processing to stop and programs to be terminated prematurely. DOS/VS offers the user a checkpoint/restart facility to aid him in reducing time lost in restarting programs that were interrupted. This feature has been designed for programs that operate sequentially.

The checkpoint/restart feature provides for checkpoint records to be taken periodically during the run. These records contain the status of the job and system at the moment they are written. Thus, they provide a means of restarting at some intermediate point rather than at the beginning of the entire job:

......C......C......C.......C.......C.......C.....error

C = checkpoint taken;   restart at last checkpoint

DOS/VS writes a checkpoint in response to a CHKPT macro in the problem program (see *Macro system* in the chapter *Input/Output Control System*).

Checkpoint records can be written on magnetic tape or on a disk pack. For magnetic tape, the programmer can use a separate file for checkpoint records, or he may have them written within an output tape file.

When a program is to be restarted, information about the restarting point is given in the RSTRT job control statement. I/O files must be repositioned to the point at which the checkpoint was taken. In some cases, this repositioning is done by DOS/VS; in other cases, it must be done by the restarting program provided by the user. It is important, therefore, to take checkpoints at the right time. Examples of proper times for taking checkpoints are:

- After a certain number of input records have been processed. The I/O routines of DOS/VS provide for taking checkpoints after each nth record.
- When tape volumes are switched. The user may have the DOS/VS label handling routines enter a user-written label processing routine which may include taking a checkpoint.
- On operator demand. For large input files, for example, the operator may decide to take a checkpoint when switching from one box of cards to the next. A request for a checkpoint can then be issued through the console.

The checkpoint/restart feature is a very useful tool, especially in a context of sequential processing. For random processing, the problem of checkpointing becomes more difficult. In many cases, random processing includes the updating of existing records, whereby the original status of updated records is not known. When restarting such a job, it is not always possible to reset the data to the point at which a checkpoint was taken, and the restart procedure may give different results.

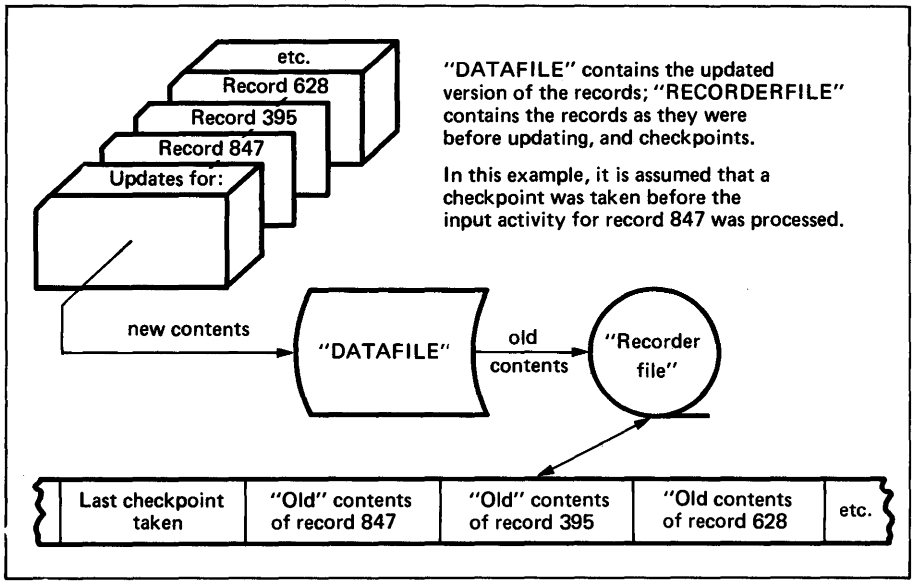
When the random file is only small, it is useful to take a dump periodically onto magnetic tape so that the original status of a file can be reconstructed. For large files that are used frequently within a certain period of time, however, this dumping may be very time consuming. A method of resetting the status of such a file to a previous status is as follows (see Figure 2.23).

Before records in a random file are updated, their original content is written to a separate file (the *Recorder file* in Figure 2.23). Checkpoints, whenever taken, are also written to this file. This means that if a restart is necessary, the latest checkpoint is used to restore the status of the system, after which the data following the checkpoint record is used to restore the 'old' content of the updated records to their original location. Then, the updating procedure can actually be restarted: it should start with the first activity input record that followed the last checkpoint when it was taken.

It should be clear that, even for large random files, it is always advisable to have a dump taken periodically.

Detailed information about the checkpoint/restart feature is presented in the publications:

- *DOS/VS System Management Guide*, GC33-5371
- *DOS/VS Supervisor and I/O Macros*, GC33-5373



etc.
Record 628
Record 395
Record 847
Updates for:
"DATAFILE" contains the updated version of the records; "RECORDERFILE" contains the records as they were before updating, and checkpoints.
In this example, it is assumed that a checkpoint was taken before the input activity for record 847 was processed.
new contents
"DATAFILE"
old contents
"Recorder file"
Last checkpoint taken
"Old" contents of record 847
"Old" contents of record 395
"Old contents of record 628
etc.

Figure 2.23. How to make checkpoint/restart possible for a random file
Because records that are updated are first written to RECORDERFILE, the status of DATAFILE can be restored should a restart be necessary.

Data Security / Data Integrity

It is extremely important that sensitive data be protected against inadvertent destruction (*data integrity*) and unauthorized access (*data security*). Payroll data about an individual should be retrieved only by a select number of users such as the payroll department or a particular manager. Other data, while it may be available for reading, should still be protected against accidental loss or inaccuracy. Equally important as prevention is the detection and correction of events that lead to violations of security and integrity.

In DOS/VS, labeling is the main tool for preventing illegal use of a data file, since a user must supply keywords (file name) before he can access the data. Also, since a label may contain an expiration date, files can be protected against destruction of current data, provided that the expiration date is well chosen.

The Input/Output Control System provides for the detection of I/O errors that may occur and, in many cases, the correction of such errors. Unrecoverable errors are communicated to the console operator.

Both file labeling and IOCS are discussed separately in this manual. In addition, DOS/VS provides four other tools that can be used in order to improve data security and integrity.

See *Diskette Security/Integrity* for a discussion of data security applicable to diskette volumes.

Track Hold Feature

In multiprogramming, more than one program may try to access the same file or wish to modify the same record simultaneously. This could result in wrong results, as the following example illustrates.

Suppose that two different programs A and B both have read a certain record X from file Y. The contents of record X is changed in two separate work areas: program A modifying in its work area field X1, and program B modifying in its work area field X2. Then both programs write record X back to its original location in file Y. The modification of the record that is written first is canceled when the second write takes place.

The situation described above will only happen on DASD, when a workfile is created to communicate with another program or task, or when independent programs process the same file randomly or sequentially. The track hold feature provided within DOS/VS prevents a track that is modified by some program or task from being accessed by another program or task at the same time. This facility can be active within one program (multitasking) or between different programs in different partitions.

A full description of the track hold facility and how it is controlled is presented in the publications:

- *DOS/VS System Management Guide*, GC33-5371
- *DOS/VS Supervisor and I/O Macros*, GC33-5373

DASD File Protect

This DOS/VS feature prohibits a user from reading or writing on cylinders not specified in the file EXTENT statements. Thus, it is a useful tool in direct access file organization for protecting data from illegal addressing, should a randomizing algorithm produce a DASD address beyond the file limits.

A full description of the DASD file protect feature is presented in the publications:

- *DOS/VS System Management Guide*, GC33-5371
- *DOS/VS Supervisor and I/O Macros*, GC33-5373

Data Set Security Facility

The format 1 label (see Appendix 3) for DASD provides for a data set security code bit setting. At OPEN time, if a file is accessed for which the data set security bit in the format 1 label is ON, a warning message is issued to the console operator. The operator must decide whether or not this file may be accessed by the program issuing the message.

Data set security will also be active also if a data set secured file is accidentally accessed, provided that the accessing program uses relative addressing techniques rather than physical track addressing techniques (see *Direct Access Method* in section 4 *Access Methods*).

The data set security bit in the format 1 DASD label is set ON by means of the DLBL job control statement.

Virtual Storage Access Method (VSAM)

VSAM offers positive data security through its password option, whereby a user must supply the correct password before he is allowed to access to a file.

VSAM also provides data integrity through its design: shared data is protected by means of the track hold feature (see above), and VSAM can verify write operations to prevent from errors to be introduced.

VSAM is described in detail in Section 4 of this manual.

Diskette Security/Integrity

DOS/VS provides five tools that can be used to improve data security and integrity for diskette volumes.

Security

- The *volume accessibility indicator* in the volume label is checked for all input and output files processed by IOCS. If at OPEN time the volume accessed is indicated as being inaccessible (secured), a warning message is issued to the console operator. The operator must decide whether or not this volume may be accessed by the program causing the message.
- The *file security indicator* in each HDR1 label for a file is checked for all input files processed by IOCS. The user has the option of creating a secured file at output time, using IOCS. If at OPEN time for an input file the file being accessed is found to be secure, a warning message is issued to the console operator. The operator must decide whether or not this file may be accessed by the program causing the message.

  If at OPEN time a secured file is being created, the volume label is also updated to indicate the volume is now inaccessible (secured).

Integrity

- The file can be protected against destruction by providing a well chosen *expiration date*.
- The file will never be overwritten by IOCS if the HDR1 label filed indicates that the file is *write-protected*.
- At OPEN time, IOCS determines that only one DTF is open to a device.

DASD Initialization and Maintenance

IBM provides an initialize disk utility program as part of the DOS/VS SCP package to prepare disk packs for use in an installation. This program writes the volume label and establishes the VTOC area for cataloging file labels.

Disk packs for the IBM 2314, IBM 3330, and IBM 3340 are partly initialized before being sent to a user: packs for the IBM 2314 (2319) contain home addresses; packs for the IBM 3330 and 3340 contain home addresses and track descriptor records (R0). For these types of disk packs a track analysis has been performed, and the user need only write a volume label and a VTOC on the new volume.

Disk packs for the IBM 2311 do not contain home addresses and R0s. Nor has track analysis been performed. When a new pack must be prepared, therefore, the Initialize Disk program also writes the required home addresses and R0s, and performs a track analysis on each track to check for any defective recording surfaces. If a defective track is spotted, an alternate track is assigned for this defective track. The address of the alternate track is placed in R0 of the defective track; the alternate track contains the address of the defective track, which is flagged as defective.

Defective DASD Tracks

Whenever a block of data cannot be successfully read from a DASD track and standard error recovery procedures have not been able to remedy this situation, a message indicating the error condition and the address of the failing track is issued to the console operator. If the program cannot continue, it is terminated. The user has the option of specifying whether he wants IOCS to verify each physical block after it has been written. The option does not require any addressable space for data in the user's program, although it does require additional CPU time. If it is not used, write data checks are not discovered until the block is read later for processing. If the option is used, write errors are discovered and the track address is included in an error message to the console operator.

How to Correct a Defective DASD Track

When the error message is issued, the operator should note the address of the defective track. For such a track, an alternate track must be assigned, and this is done by another IBM-supplied utility program: the alternate track assignment program. This program flags the home address of each defective track and selects an alternate track from the cylinders that are reserved for that purpose. The address of the alternate track is written in R0 of the defective track, and the address of the defective track is written in R0 of the alternate track.

Whenever a program wants to access track that has been flagged as a defective track, this is detected. For the IBM 3330 and IBM 3340 defective tracks are detected by the device; for the IBM 2311 and IBM 2314 (2319), the necessary checks are done by DOS/VS. In all cases the physical channel program is restarted for the alternate track, using the address noted in R0 of the defective track. No user action is required.

Whenever a channel program that operates in multiple track mode reaches the end of an alternate track, this is also detected. IOCS error recovery procedures then cause the track following the defective one to be accessed next.

Since all switching to alternate tracks and back is done by IOCS routines, problem programs need never be concerned with flagged defective tracks or alternate tracks. There is, of course, a slight decrease of performance, each time an alternate track must be accessed.

Preparing DASD Volumes for Data

The Initialize Disk utility program is used to prepare one or more complete volumes for use in an installation. It is also used to re-initialize volumes in case of changing job requirements. To guard against accidental destruction of current data, the VTOC is always checked for labels reflecting unexpected data files.

After a volume has been processed with the Initialize Disk program, each track contains a track descriptor record (R0), that describes the entire track as 'free space' which then can be used for new data. This is an important feature, allowing the storing of variable-length data blocks through a direct access processing method. It is explained in detail in section 4 of this manual, under *Capacity record* in the chapter *Direct Access Method.*

For random files containing fixed-length records, the situation is somewhat different. Here it is advisable to have the file preformatted with dummy records of fixed length. This can be done by means of the Clear Disk utility program, supplied by IBM. A preformatted file has the advantage that all possible record locations have been initialized and are accessible for both input and output. Detailed information about the preformatting of random files is presented under *Loading and processing direct access files*, in the chapter *Direct Access Method*, section 4 of this manual.

Summary of DASD Initialization and Maintenance

The Initialize Disk utility program must be used to prepare a new volume for use. The volume will then contain:

- A VTOC, preformatted with a series of 140-byte DASD labels. The first label in the VTOC contains a format 4 DASD label, describing the VTOC itself. The second label is reserved, but will not be used by DOS/VS. All other labels are filled with binary zeros.
- A volume label, as the third record on cylinder 0 in track 0.
- A home address and a track descriptor record (R0) on each track.
- Defective tracks replaced by alternate tracks.

The volume can then be used in the following procedures:

- Sequentially loading of a file with either fixed-length or variable-length data blocks.
- Random loading and processing of a file with variable-length data blocks.
- Preformatting of a file with fixed-length data blocks by means of the Clear Disk utility program, which writes a complete file, filled with dummy records. Such a file can be used for randomly loading and processing fixed length data blocks. (*The Clear-Disk utility program operates on logical files only; the Initialize Disk utility program operates on complete volumes.*)

The user of a volume decides where the VTOC must be placed, within the following restrictions:

- The VTOC must be placed in the area which is normally available for data, including cylinder 0; cylinders in the alternate track area must not be used. In cylinder 0, the VTOC may begin on track 0, immediately following the last volume label. An exception is the system residence pack: on this volume the VTOC must be placed outside the residence area.
- The VTOC must be on one or more full tracks, with the exception of track 0 on cylinder 0, as noted in the previous lines. The VTOC must be contained within one cylinder.

Section 3: DATA MANAGEMENT SUPPORT BY DOS/VS

This section describes the actual data management support provided by DOS/VS. The first chapter is devoted to organization of data files and the processing of data files in general, and explains differences between the organization and processing of data. It also introduces logical and physical organization as opposed to logical and physical processing. Various organization and processing techniques are introduced. An important conclusion of this chapter is that a certain data organization (random or sequential) does not always automatically imply a certain processing method (random or sequential).

The second chapter introduces the three main access methods of DOS/VS in general terms. It presents the main features of each of them separately, as well as their restrictions.

The third chapter of this section gives guidelines for choosing the right access method for a specific file in a specific processing environment. It discusses the various attributes of a data file, and provides examples.

The last chapter explains the concepts of the DOS/VS Input/Output Control System (IOCS). It discusses the macro system in terms of data management, and explains its relationship to IOCS. As such it acts as an introduction to section 4 of this manual which makes use of terms introduced here.

This section will be of interest mainly for assembler language programmers since they have extended control over all features provided by IOCS. Users of high-level programming languages may find the discussions presented interesting because they give information on how IOCS actually processes their data. They should realize, however, that not all of the features discussed may be available for them in their language, and they should, therefore, consult the manuals for their language processor as well.

Access Methods and File Organization

Users tend to look at their data logically: they prefer to talk about logical records rather than physical blocks or stored records. Therefore, users also tend to regard the organization of the data logically: the sequence of the logical records and the way the data is stored and retrieved are usually discussed from a logical point of view.

An operating system, on the other hand, handles data physically: it arranges and accesses the data in a physical way.

The interface between these two points of view is the access method. An access method is a technique for moving data between virtual storage and I/O devices. It includes the mapping of logical records (user-oriented) into physical blocks (device-oriented), via stored records (operating system-oriented) and vice-versa. It also includes the physical organization, which is the arranging of data, physically, in the file.

Organization and Processing of Data

Data can be organized, physically as well as logically, in two basic ways:

- randomly
- sequentially

Organization usually refers to the way a file is created and maintained, or to the order of the input transaction records from which the file is created and maintained. Organization should not, however, be confused with the processing of an existing file that has been created and updated. Processing falls into the following basic categories (Figure 3.1):

1. Sequential processing of a sequentially organized file.
2. Random processing of a sequentially organized file.
3. Sequential processing of a randomly organized file.
4. Random processing of a randomly organized file.

	Sequential Organization	Random Organization
Sequential Processing	1	3
Random Processing	2	4

Figure 3.1. Sequential/random organization vs sequential/random processing
Organization refers to the way data is arranged in a file, whereas processing refers to the way in which this same data is accessed and used.

Serial Devices such as Tape, or Card

Sequential
Records on tape and card files must be processed as they are encountered because of the physical nature of the storage medium. Therefore, tape and card master files, and their associated transaction files (regardless of the medium), tend to be sequentially organized.

Random
Random processing for tape files is impractical, since the desired record can fall anywhere within the file limits, and a search for a specific record must begin with the first record of the file. A random organization of transaction records is common when the master file is stored on a direct access device, since such devices are capable of retrieving records randomly. In those cases where it is desirable to maintain information in the same sequence as the source documentation to facilitate control and validation procedures, a transaction file may be created in the order of occurrence, with sorting as an intermediate step, before processing against the sequentially organized master file.

Direct Access Storage Devices (DASD)

Sequential

In a sequentially organized DASD file, the records are stored in consecutive order. Thus, a sequential DASD file can be processed strictly sequentially in the same fashion as magnetic tape. This method does not, however, take full advantage of the ability of a DASD to locate a specific record directly and thereby eliminate the time required to read inactive records. In a multiprogramming environment this type of sequential processing can be more time consuming than the processing, sequentially, of magnetic tape. In most cases a DASD volume will contain more than one file, and some of them may be processed at the same time, by different programs or by the same program. The read-write heads will then move many times between files, which will result in an increased average access time.

Records are usually stored in primary key sequence, so that records with successively higher primary keys have successively higher address numbers. Normally there is no direct relationship between the content of a primary key in a record and the address number of that record; the only requirement is that the primary keys be in sequence and in sequential (not necessarily consecutive) disk storage locations.

Additions to the file present the greatest design challenge. It is not possible to insert new records physically in primary key sequence within the existing file, since the records in the original (prime) area are placed adjacent to one another. New records are known as overflow records and are put into a separate area called overflow area. The overflow area is set aside on specific tracks of the same cylinder or, since it may be difficult to predict the overflow pattern, on a single cylinder or group of cylinders. An index system is used to retrieve the records from prime area and overflow area in primary key sequence. This index system may also be used to locate individual records. Figure 3.2 shows a simplified example of a sequential DASD file containing a prime area and an overflow area. In this example, the index system mentioned above is replaced by pointers in the data records: each pointer contains the address of the record that has the 'next higher' primary key.

a. Initial status of the sequential DASD file.

	Address #	Primary key	Address # of next record *
	000001	0002865	000002 P
Prime	000002	0002866	000003 P
area	000003	0002885	000004 P
only	000004	0002890	000005 P
	000005	0002891	000006 P
	etc.	etc.	etc.

b. New records must be inserted. Their primary keys are:

0002867
0002880
0002888
0002889

c. Status of the sequential DASD file after inserting.

	Address #	Primary key	Address # of next record *
	000001	0002865	000002 P
Prime	000002	0002866	000001 O
area	000003	0002885	000003 O
	000004	0002890	000005 P
	000005	0002891	000006 P
	etc.	etc.	

	Address #	Primary key	Address # of next record *
	000001	0002867	000002 O
Overflow	000002	0002880	000003 P
area	000003	0002888	000004 O
	000004	0002889	000004 P

* P indicates an address # in the prime area,
O indicates an address # in the overflow area.

Figure 3.2. A simplified example of sequential DASD file maintenance

Random
In a random file, records are stored at an address that is obtained by applying a mathematical formula to the primary key. No indexes are required to locate a specific record, since the DASD address can be found using the same conversion routine for retrieval. The main difficulty is finding a proper conversion algorithm, especially when the primary key contains alphameric characters. In many cases users tend to develop a special coding scheme for the primary keys in order to obtain a more computer-oriented (DASD-oriented) code that is easy to convert into DASD addresses.

Example: assume an application where three types of loans (A, B, and C) are to be stored on DASD. For those three types, the file is logically divided into three parts, as follows:

Type of loan	DASD addresses	Number of available DASD storage locations
A	00600–34099	33500
B	34100–47999	13900
C	48000–59999	12000

In this file, the loan account number (primary key) is used as input for a randomizing algorithm, to obtain a DASD address for each record, as follows:

1. Determine the type of loan, for step 2 (*assume a type C loan*).
2. Multiply the loan account number (*assume account number 38596*) by the number of available DASD storage locations minus 1 (*which is, for type C loans, 12000-1 =11999*):
   38596x11999 = 463113404
3. Drop the five low-order positions of the product obtained in step 2; the remaining part (*4631*) is then used as a relative DASD address: relative to the beginning of one of the three parts of the file (*for type C loans, address 4631 is relative to location 48000*).
4. Add the relative address obtained in step 3 (*4631*) to the lowest location of the allotted part (*48000*); the result is the DASD address desired:
   4631+48000 = 52631
   Converting two different primary keys into DASD addresses can result in the same DASD address. Such duplicate addresses are called synonyms. If the conversion algorithm is supposed to convert primary keys into single DASD *record* addresses, then all but the first synonym become overflow records. A conversion algoritm may also convert a primary key into a DASD *track* address; in this case a synonym becomes an overflow record only if the track is completely filled. Overflow records are put into

a separate overflow area. The organization of random files is discussed in more detail in the chapter *Direct Access Method* in the section *Access Methods.* A random file may be processed sequentially be reading the records in consecutive order. It should be kept in mind, however, that not all record locations are necessarily filled with data. In fact, since a randomizing algorithm is used for creating as well as for updating a random file, the records will very often be clustered in the file, with 'open' spaces between the clusters; not all available record locations may be used. As long as a record location has not been written with any kind of data, dummy or current, this will normally cause no problem for the user: these locations will be ignored by the sequential routines. But as soon as a record location has been written with any kind of data, dummy or current, the sequential routines will retrieve those records; it is then the user's responsibility to recognize dummy records from current data, and to ignore the dummy records. An example of a dummy record is a record that contains deleted current data, for example overwritten with zeros, or with a particular field set to a specific value. More information about the use of dummy records is presented in the chapter *Direct Access Method* in the section *Access Methods.* For random processing, all volumes that constitute a file must be online at the same time, so that the size of a random file is in fact limited by the online DASD capacity of an installation.

Organization of Physical Data

The last chapter discussed the types of processing that are practical for different types of I/O devices. It was explained that certain device types are especially suitable for sequential processing, whereas DASD can be used for both sequential and random processing. It was also explained that the type of processing is not necessarily dependent on the way the data is organized: on DASD it is possible to process a sequentially organized file randomly, and a randomly organized file sequentially. Other device types are designed for sequential processing and a sequential organization of the data is then most practical.

Sequential Organization

In a sequential file, records are organized solely on the basis of their successive physical locations in the file. The records are generally (but not necessarily) in a sequence according to their primary keys as well as in physical sequence. The records are usually updated or read in the same order in which they appear. For example, the hundredth record is usually processed after the first 99 have been processed.

Records usually cannot be deleted or added unless the entire file is rewritten. Individual records cannot be located quickly, that is, in most cases there is no direct relationship between the physical location of a record and the primary key of that record. An individual record then can be located only by scanning the entire file, until the desired record is found. If, however, a direct relationship between the location of a record and its primary key does exist, which is totally the responsibility of the user, the file may also be processed randomly, provided that the file resides on a direct access device.

Random Organization

A file organized in a direct (random) manner is characterized by a predictable relationship between the primary key of a record and the location (address) of that record on a direct access device. This relationship is established by the user of the file. This organization method is generally used for files whose characteristics do not permit the use of a sequential organization or for files where the time required to locate individual records must be kept to an absolute minimum.

This organization method has considerable flexibility. Its accompanying disadvantage is that, although the operating system includes data management routines to read or write a file of this type, the user is largely responsible for the logic and programming required to locate individual records, since he establishes the relationship between the primary key of the record and its address on the direct access device. It will be explained later that, within certain restrictions, a file that is organized randomly, can also be processed sequentially, that is, in physical sequential order.

Indexed Sequential Organization

Logically, the indexed sequential organization is an extension of sequential organization, and includes the capability of random processing and updating. Physically, depending on the implementation, the indexed sequential organization is a technique which includes facilities for the maintenance and management of the data in a particular type of file, and which allows for different types of processing. DOS/VS includes two access methods that can be characterized as indexed sequential methods.

An indexed sequential file is similar to a sequential file in that rapid sequential processing is possible, that is, depending on the implementation it may allow for the processing of records in physical sequential order (the sequence in which the records appear), or in logical sequential order (the sequence determined from the contents of all primary keys), or both in physical sequential and logical sequential order.

An indexed sequential file is similar to a random file in that rapid location of individual records is possible. The main difference here is that individual records in a random file are located through a randomizing algorithm established by the user, whereas individual records in an indexed sequential file are located through a set of indexes.

The physical organization of an indexed sequential file is totally under control of the operating system, so that an implementation of this type of organization method is often called a file management system. The user needs to do only very little I/O programming.

Relating Organization to Processing Methods Provided by DOS/VS

The three methods of data organization listed above are supported under DOS/VS by four different access methods, each of which is described in detail in following chapters. The four access methods are:

- Sequential Access Method (SAM), for processing files in physical sequential order.

- Sequential Access Method (SAM), for processing files in physical sequential order.

- Direct Access Method (DAM), for processing files in random order.

- Indexed Sequential Access Method (ISAM), for organizing as well as processing data in either a logical sequential order or a logical random order, on the basis of the key of the individual logical records, provided that the data is stored in the ISAM format.

- Virtual Storage Access Method (VSAM), for organizing as well as processing files in four different ways, provided that the data stored has the corresponding VSAM data format:

  1. Physical sequential, based on the physical sequence of the individual logical records.
  2. Physical random, based on the relative location of individual records in the file (relative to the beginning of the file, assuming that all logical record are stored adjacent to each other).
  3. Logical sequential, based on the primary keys of the individual logical records.
  4. Logical random, based on the keys of the individual logical records.

The reader should note that both ISAM and VSAM perform the function of actually organizing the data in external storage, while the user is required to do this when using SAM or DAM.

Introduction to the Access Methods Provided under DOS/VS

It is the user's responsibility to choose both a data organization method and a processing method to fit his application. As was explained before, a sequentially organized file may be processed sequentially or randomly, and a randomly organized file may be processed sequentially or randomly.

The Input/Output Control System of DOS/VS includes various types of I/O processing routines for the user; these routines are grouped into four different types of access methods, each access method supporting either a processing method (sequential or random) or both a processing method and an organization method (sequential or random):

- Strictly sequential processing: reading and writing records in consecutive order is done by the Sequential Access Method (SAM), and by the Virtual Storage Access Method (VSAM) in an entry-sequenced data organization.

- Random (direct access) processing: reading and writing records in a non-consecutive order is done by the Direct Access Method (DAM), and by the Virtual Storage Access Method in an entry-sequenced data organization.

- Sequential DASD organization with the ability of locating individual records directly: reading and writing records in either a primary key sequence or individually is done by the Indexed Sequential Access Method (ISAM), and by the Virtual Storage Access Method (VSAM) in a key-sequenced data organization.

DAM, ISAM, and VSAM can be applied to DASD files only; SAM is applicable for any type of device. Each of the access methods has its own strengths and weaknesses; the characteristics of the access methods are briefly introduced in the following pages. The next chapter will then guide users in deciding which access method to choose. Detailed information about programming for each of the four access methods is presented in the last section of this publication.

In addition to the four access methods introduced above, which are normally used for online batch processing, DOS/VS provides for teleprocessing by means of two teleprocessing access methods:

BTAM:	Basic Teleprocessing Access Method
QTAM:	Queued Teleprocessing Access Method

Teleprocessing is beyond the scope of this manual; readers who wish information about this subject are referred to their IBM representative or the IBM branch office serving their locality.

Sequential Access Method (SAM)

The Sequential Access Method allows a programmer to store and retrieve the records of a file in consecutive order. This method can be used for card, printer, printer-keyboard, magnetic tape, optical character reader, magnetic ink character reader, paper tape, diskette, and DASD files.

Direct Access Method (DAM)

The Direct Access Method is a flexible access method provided specifically for use with direct access storage devices. Some of the features of these devices are:

- Flexible record referencing, either to physical track and record address (record ID) or to record key (control field of the physical block).
- Ability to search sequentially through an area for a physical block, using a minimum of central processing unit time.

The Direct Access Method does not include elaborate routines for handling file maintenance functions such as:

- Adding records to existing files
- Handling overflows
- Locating synonym records
- Deleting records

These functions are entirely the user's responsibility. This may seem a disadvantage, but, once a user has determined the way he will handle his data, DAM will prove to be the most powerful and flexible method available. Many of the problems associated with file maintenance are virtually eliminated because of the advanced recording and addressing technique of the direct access storage devices used with the IBM System/370. High-level programming languages, on the other hand, may not be able to support the devices fully, due to the restricted nature of the languages themselves; high-level language programmers should consult their language reference manual in order to learn about the device features that are under their control.

Indexed Sequential Access Method (ISAM)

The Indexed Sequential Access Method is a *file management system* developed for use with DASD; logical records are organized by ISAM on the basis of a collating sequence determined by their primary keys.

As a file management system, ISAM takes care of the data organization where SAM and DAM do not. Difficulties such as handling overflows when inserting records on an existing file and retrieving those records later, are solved by ISAM data management routines, invisible to the user. As a result, the management of ISAM data requires only very little I/O programming by application programmers.

ISAM offers the programmer a great deal of flexibility in the operations he can perform on a file. He has the ability to:

- Read or write logical records whose primary keys are in ascending collating sequence.
- Read or write individual records randomly, on the basis of the primary keys. If a large portion of a file is being processed, reading records in this manner is somewhat slower than reading according to a collating sequence. A search through indexes is required for each logical record.
- Add logical records with new keys to the existing file. The file management routines of ISAM find proper locations in the file for the new records and make all necessary adjustments to the indexes so that the new records may be retrieved easily. New logical records are physically stored in a separate *overflow area*; the logical sequence to other logical records in the file is maintained through the indexes. As new records are added, the performance of ISAM decreases slightly, until it becomes advisable to reorganize the file (see below).

ISAM has the following restrictions:

- Data records may be blocked or unblocked but must be fixed length only.
- All *physical* blocks must contain a key area; all key areas in an ISAM file must be of the same length.
- For multivolume files, all volumes must be online for any function to be performed.
- ISAM uses three types of data areas in auxiliary storage: prime data area, overflow area, and indexes. The prime data area must be allocated in one continuous area which may be over more than one volume; it must begin on the first track (track 0) of a cylinder and it must end on the last track of a cylinder. For a multivolume file, the prime data area must continue from the last track of the last cylinder on one volume to the first track of cylinder 1 of the next volume, so that the area is considered continuous by ISAM (cylinder 0 is reserved for labels). The overflow area and the indexes may be located on separate volumes.
- An ISAM file cannot be used as input for sort/merge programs. The data is organized on a logical basis: the logical records are sequenced logically, according to the primary keys of the records. Should a user attempt to re-sort this file, the indexes would no longer be an interface between the user's problem program and the data records,

and individual records can no longer be located. It is possible, of course, to create another file from the contents of an ISAM file, and then to re-sort this newly created file.

- Once a file has been created as an ISAM file, it should be processed and updated by means of ISAM only. SAM, DAM, or VSAM must never be used to process an ISAM file; doing so might cause serious problems in the area of data integrity. ISAM manages the data completely, and this management function might be made impossible if a user were to destroy an index/data relationship as established by ISAM.

- ISAM does not provide for actually deleting logical records from an ISAM file. However, since a user may update any logical record, he can 'delete' a logical record by overwriting the data portion with, for example, binary zeros, decimal zeros, or blanks. He must not include the primary key in overwriting. He may also enter a special field in his logical record which contains the status of the data: current data or deleted data (See note, below). The main issue is that the user himself must distinguish deleted data from current data, and choose some satisfactory way of doing so.

  A disadvantage of this restriction is that an ISAM file increases in size as many logical records are 'marked' as deleted, and many new records are added. Eventually, the file must be reorganized in order to obtain a 'clean' file. During this reorganization, the file is read sequentially (logically, according to the primary keys) and written (loaded) to a new file; logical records that have been 'marked' as deleted are then ignored by the user, and are not written to the new file.

Note: Under OS/VS1 or OS/VS2, deleted records are flagged by placing the hexadecimal value "FF" in the first byte. It is recommended that DOS/VS users, who plan to use DOS/VS ISAM data under OS/VS, follow this procedure.

Virtual Storage Access Method (VSAM)

VSAM is a file management system developed for use with DASD. It differs from the access methods described earlier in that:

1. It allows two different ways of data organization, each of which allows different ways of processing.
2. It includes a facility for automatic space allocation for files on DASD.
3. It includes a set of service programs which can:
   - define or reorganize VSAM files
   - load records into a VSAM file
   - copy or print VSAM files
   - create a backup copy of a VSAM file
   - convert a SAM file or ISAM file to the VSAM format
4. It allows for the processing, by means of ISAM macros, of ISAM files that have been converted to the VSAM format, which means that most programs need not be changed.
5. It offers device independence by means of a special format of its physical blocks, into which the logical records are mapped in a special form of spanned records.

6. It offers data integrity and data security by means of design, and security and integrity options.

   In VSAM, a user may choose between two types of data organization:

   - Key-sequenced data organization.
   - Entry-sequenced data organization.

7. It allows to save storage in a multipartition system, since its modules can be loaded into the SVA. For details see *DOS/VS Operating Procedures*, GC33-5378.

In a *key-sequenced* data organization, logical records are stored on the basis of a collating sequence determined by the content of the primary keys of those records. As new logical records are added and old logical records are deleted, the entire file is kept in sequence according to the collating sequence of the keys.

The key-sequenced data organization is basically similar to the organization of an ISAM file. VSAM, however, does not use overflow areas, and the decrease in performance when records are added in an ISAM file is virtually absent in VSAM.

The key-sequenced data organization allows four types of processing as ISAM; the first two are also allowed by ISAM.

- Processing of individual logical records randomly on the basis of their primary keys (*keyed-direct processing*).
- Processing of a series of logical records in a logical sequence according to their primary keys (*keyed-sequential processing*).
- Processing of individual logical records randomly on the basis of their locations in the file (*addressed-direct processing*).
- Processing of a series of logical records in a physical sequence according to their locations in the file (*addressed-sequential processing*).

In keyed-direct and keyed-sequential processing, the physical location of a logical record is determined by VSAM through an index structure. In addressed-direct and addressed-sequential processing, the user must know the location of a logical record, relative to the beginning of the file.

As logical records are deleted, VSAM adds the space that was occupied by those records to the amount of free space in the file. *(automatic space reclamation)*. In this process the physical location of other logical records will change and, although a change of location is communicated to the user, addressed processing requires that a user must keep track of the location of logical records at all times. For keyed processing the user need not be concerned about address changes of logical records, since VSAM maintains each change in address in the index structure.

The advantage of a key-sequenced VSAM file above an ISAM file is that, to a certain extent, this type of file is self-reorganizing, so that the average retrieval time will be virtually constant.

In an *entry-sequenced* data organization, logical records are stored physically in the same sequence in which they were entered. New logical records, when added to an entry-sequenced VSAM file, are physically stored at the end of the file. This organization is basically similar to that of a SAM file.

The entry-sequenced data organization allows only two types of processing:

- Addressed-direct processing
- Addressed-sequential processing

These terms were explained above.

The space once occupied by deleted records in an entry-sequenced VSAM file is not added to the amount of available free space, as is done for a key-sequenced file. This implies that an entry-sequenced VSAM file is not self-reorganizing. Any logical record that is entered will remain in the same location for the life of the record, unless a user decides to change its length; in that case the old version is automatically deleted, and the new version is added to the end of the file. Logical records may, however, be updated or even replaced by other logical records, provided that they are of the same length.

VSAM accepts unblocked logical records which may be of a fixed length or a variable length format; both formats are treated the same: logical records are mapped into units of disk storage called *control intervals.* A control interval is the unit of information that VSAM transfers between virtual storage and auxiliary storage, and may consist of one or more physical blocks. The amount of space available in a control interval is the same for all control intervals that constitute a VSAM file.

Logical records are transformed into a special type of spanned record structure; they may span physical blocks within a control interval, but they will never span control intervals.

A VSAM file may physically be written over one or more extents, over one or more volumes; logically, however, a VSAM file is considered one continuous string of bytes, each byte addressable by the user. Individual logical records are addressed by their *relative byte address* (RBA), relative to the logical beginning of the file, rather than by their actual DASD address. This concept allows one file to be copied to another area on the same device or on another, even on another device type, without changing the address scheme. VSAM calculates DASD addresses from the RBAs, taking the physical characteristics of the file and of the device into account.

In addition to the data organization facilities described above, VSAM has the following features:

- Automatic space allocation.
  In the other access methods (SAM, DAM, and ISAM) the user is required to keep track of occupied and free space on DASD volumes, and to assign free space to his files before they are created. This contrasts with VSAM. An arbitrary total amount of DASD space is assigned to VSAM. From this total amount of space, VSAM selects the amount required and assigns it to a VSAM file. All of the VSAM data space is maintained through a separate *catalog* which describes the logical and physical attributes of all VSAM files in one system.

Space allocation by VSAM is dynamic; as needed, more space is added. During the allocation process (generally, during most of the processes that require updating of the catalog only) the volumes involved do not usually need to be online. The total space assigned to VSAM may amount to more volumes than the number of DASD drives available in an installation.

- ISAM interface routine.
  After having been converted into the VSAM data format, ISAM files may continue to be processed by the same program which continues to use 'old' ISAM macros. It is recommended to convert ISAM files into the VSAM format since reorganization of the data will then not be needed as frequently. A VSAM file has no overflow areas; a new record is inserted physically in a sequence according to its key.
- Data integrity and data security.
  VSAM protects data by means of its design and its security options. The design of VSAM allows users to access data only by specifying the correct catalog information; the catalog itself points to the data, and the knowledge stored in the catalog is restricted to VSAM. Using the security options, a VSAM user may specify that only a correct password will permit data to be accessed.
- Device independence.
  The common VSAM record structure into which all logical records are mapped, and the fact that a VSAM file is logically considered as a continuous area, allows a VSAM file to be processed on different device types without reprogramming. The only restriction is that a VSAM file must be entirely stored on volumes of the same device *type*. Also, a VSAM file may be copied to another file, and the copy can be processed without changing the program: the catalog will point to the proper extents, and VSAM will compute correct DASD addresses in all cases.
- Data portability.
  Because the data is cataloged in the VSAM catalog, a VSAM file belongs to a specific installation. The set of utility programs available, however, includes the possibility of *'exporting'* a VSAM file from one installation, and of *'importing'* the same file into another. The catalog information is then carried along with the data. Data portability includes the possibility of interchanging VSAM files between DOS/VS and OS/VS.

VSAM has the following restrictions:

- Logical records may be of fixed length or of variable length format, but must be unblocked. Logical records are stored in VSAM record format, invisible to the user.
- A VSAM file must be processed by VSAM only. Other access methods do not have access to the VSAM catalog and therefore cannot determine the required DASD addresses.
- A VSAM file cannot be used as input for sort/merge programs, because this would affect the catalog information. In addition, the location of a logical record could no longer be found via the index after re-sorting the file. This does not preclude the creation of another

VSAM file in another sequence by a user program, provided that the original file remains undisturbed or is deleted afterwards.

- The user should realize that the physical locations of logical records in a key-sequenced VSAM file will change as other logical records are added or deleted. Therefore, addressed processing of such a file will present a problem unless the user keeps track of all these changes. VSAM communicates all address changes to the user, and the user could maintain his own indexes as separate VSAM files. Such a 'user index file' could contain logical records consisting of a key and, as the only data, the address of a logical record in the main file. An address change in the main file should then result in an update of a record in the 'user index file'.

Choosing the Right Organization and Processing Methods

It was explained earlier in this section that there are differences between the organization of a file and the processing of a file. It was explained also that it is mainly the DASD which allows for different organization methods and processing methods. The other device types such as card, magnetic tape, and printer, have characteristics that call for sequential organization and processing. It is evident that the choice of a device type which can handle only sequential processing limits the possible later changes in the application. Therefore, a choice of a specific device type must be based on an overall consideration of the application and of its requirements for the near future, rather than on a few programs in that application or on temporary requirements.

This chapter indicates the considerations that must be kept in mind for files that are stored on direct access storage devices. It stresses the organizational aspects, by relating those aspects to processing requirements and characteristics of the data. It indicates how a file should be created and maintained rather than how it should be processed.

Organization and Processing of DASD files

An installation equipped with DASD lends itself to different file organization and processing methods. It is important, therefore, to analyze each file and the programs that process it to ensure that the chosen method constitutes the best solution to the data processing requirements of the installation.

In many cases, the most suitable type of organization and processing is immediately evident. However, some applications may require additional study because of their complexity, their unusual processing requirements, or because of the wide range of programs that use a particular file. This is an important aspect of planning for a data processing system. Decisions in this area may affect system configuration requirements and should be made before programming even begins. The general level of efficiency of the data processing installation may be affected.

There are no absolute rules for the resolution of uncertainties regarding the organization and retrieval alternatives. However, there are several criteria that may provide an indication of the best solution.

Criteria for Choosing an Organization Method

The following items form a basis for a decision concerning the organization of a file:

- File activity
- File volatility
- Filesize
- Response time

These items are discussed in the following text. The reader should note the following:

In the following text, if a reference is made to sequential organization, the user may choose between SAM or the entry-sequenced organization of VSAM. If a reference is made to indexed sequential organization, the user may choose between ISAM or the key-sequenced organization of VSAM (the VSAM approach is generally preferred). If a reference is made to random organization, the user may choose between DAM and the entry-sequenced organization of VSAM; special considerations are necessary for VSAM, however.

File Activity

Activity refers to the number of records in a file for which there are transactions. This is usually expressed in a percentage. For example, 10% activity in an inventory file means that, during a period of time, there are transactions to be posted to 10% of the logical records contained in the inventory file.

As the file activity increases to 60% or more, sequential *processing* becomes more efficient. Sequential processing (sequential on the basis of the primary key) implies either sequential or indexed sequential *organization.* A high activity implies batch processing which means that transactions need not be posted at the moment they occur. In fact, the time between the occurrence and the posting may vary from a few hours to weeks or even months, depending on the application.

Some applications do not allow transactions to be batched. An example would be an online inventory file where the transactions have to be handled as they occur.

A low file activity may justify direct retrieval instead of sequential retrieval; for direct retrieval on the basis of a primary key either an indexed sequential or a direct organization must be used.

File Volatility

Volatility refers to the number of additions to, or deletions from, an existing file. First, consider the effect of making additions to or deletions from a file that is sequentially organized by means of SAM. As is illustrated in Figure 3.3, this case is similar to a construction which is often used for magnetic tape. The existing file is merged with a transaction file which is ordered in the same sequence. As output an updated version of the original file is created. (The transaction file, of course, may be on any device, as required).

With a low volatility (few additions and deletions), an indexed sequential organization provides a practical solution. One of the advantages of an indexed sequential organization is that additions and deletions can be handled without copying the file. This is illustrated in Figure 3.4.

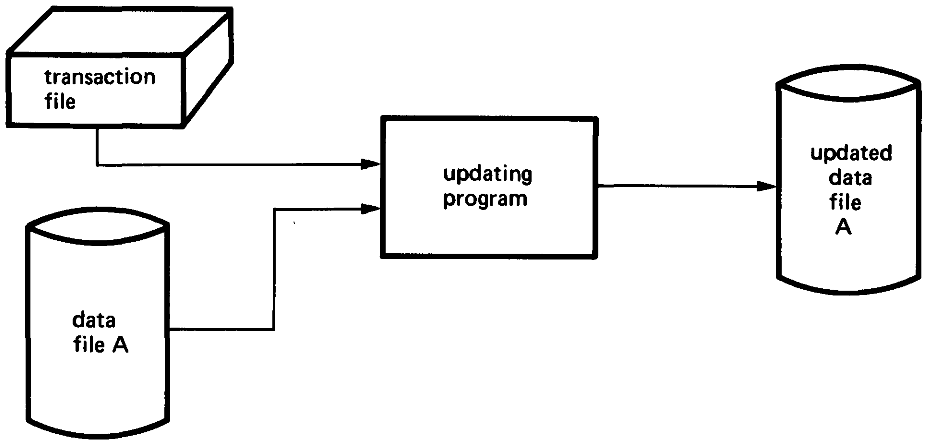


Figure 3.3. Updating a sequentially organized file (SAM)
A sequential file has to be copied for updating.

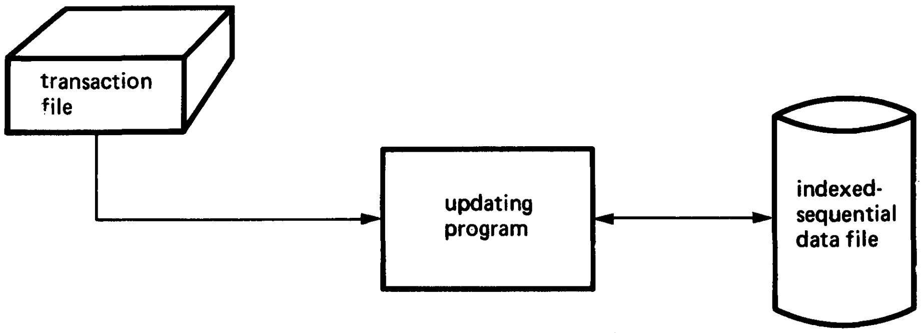


Figure 3.4. Updating an indexed sequential file (VSAM or ISAM)
An indexed sequential file need not to be copied for updating.

The processing efficiency of an ISAM file decreases as the quantity of activity increases. Logical records cannot be physically removed from the file: they can only be 'tagged' for deletion, and be removed later when the entire file is reorganized. Additions cause logical records to be placed in an overflow area, and retrieval of those records in the collating sequence of the file requires more time than the retrieval of a logical record from the prime data area where they were stored when the file was created (loaded). This is caused by the additional access arm movements between prime data area and overflow area, and by the record-by-record search needed for overflow records. Therefore, an ISAM file must be reorganized from time to time; processing with a reorganized ISAM file is highly efficient.

The cycle of creating an ISAM file, followed by processing - reorganizing - processing, is shown in Figure 3.5. The reorganization of an ISAM file consists of reading the existing file in collating sequence, and copying it to a new one, ignoring all logical records that have been 'tagged' for deletion. The new file then replaces the old one in the problem programs processing the file.

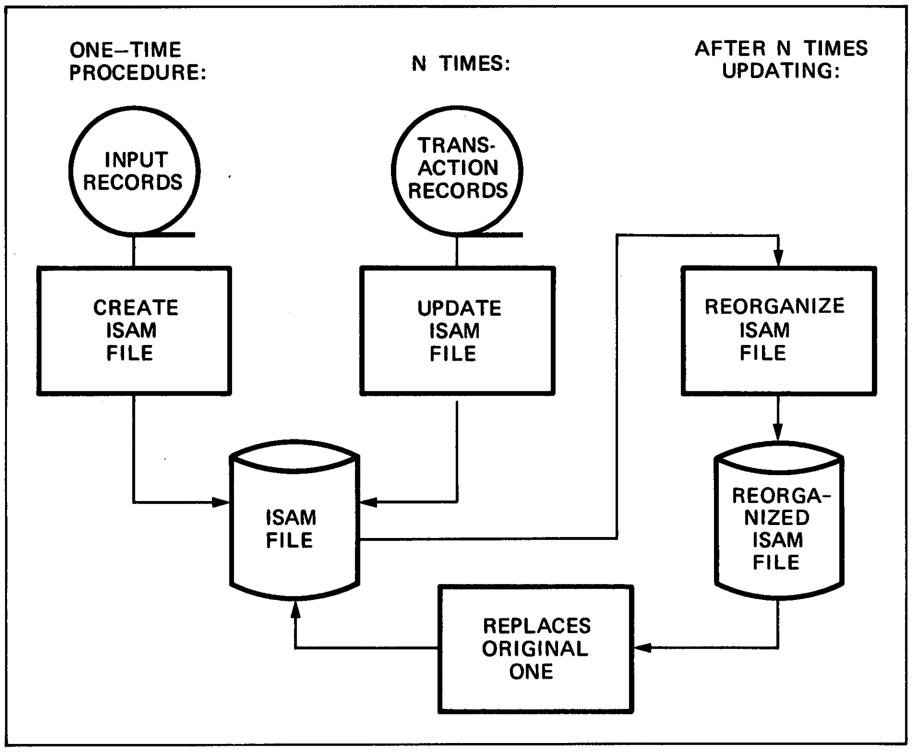


Figure 3.5. The cycle of creating, processing, and reorganizing an ISAM file
Because logical records cannot be physically removed from this file, occasional reorganization is needed to maintain processing efficiency.

Processing efficiency is not affected in VSAM files by the factors that influence ISAM files. A key-sequenced VSAM file has the following advantages over an ISAM file:

- Deleted logical records are physically removed from the file, and the space they occupied is added to the amount of free space in the file.
- Added logical records are not placed into an overflow area; the entire file is kept in collating sequence, and new records are physically entered in the proper collating sequence.

Efficiency of a VSAM file may be slightly lower than that of a *reorganized* ISAM file. However, the efficiency of a VSAM file is almost constant and therefore the average processing efficiency of an average key-sequenced VSAM file is higher than of an ISAM file. Moreover, a VSAM file does not require frequent reorganizing as an ISAM file does.

This does not mean that a key-sequenced VSAM file never requires reorganizing. As logical records are deleted during processing, the space they occupied is added to free space within the collating sequence in a control interval (see Figure 3.6). This free space will be used again for another logical record within the same range of sequence of that control interval. As long as such new records are not added the space will not be used, unless the entire control interval becomes completely empty. Thus, although there may be free space available in some place within the file, there may be other places in that file where more space could be required than is available. This may result in an inefficient usage of space, which will be resolved when the file is reorganized like an ISAM file.

Figure 3.6 illustrates two subsequent control intervals from which a number of records are deleted. The space that is freed can be used again for other logical records within the same range of sequence only (3 to 44, and 45 to 85, in the example given). If such records are not added, the space will not be re-used.

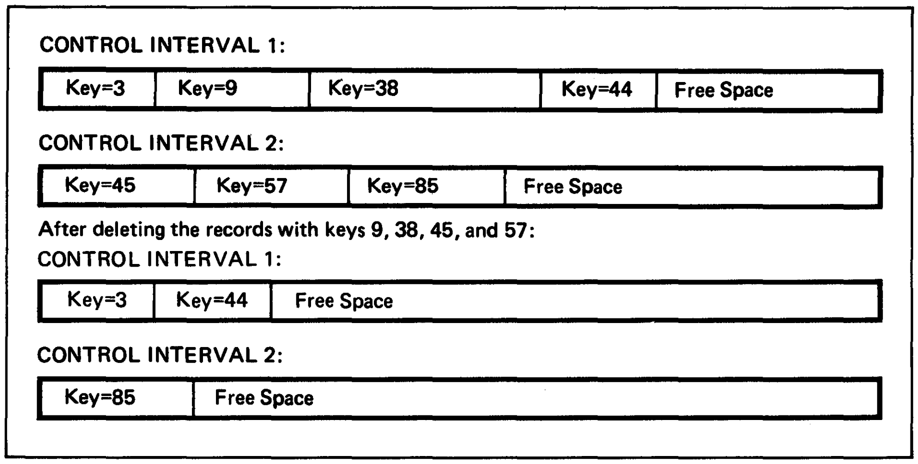


Figure 3.6. Deleting logical records from a key-sequenced VSAM file
If the resulting free space is not to be used for new records, reorganization may be advisable.

Additions to and deletions from a random file which is organized and processed by means of the Direct Access Method do not necessitate the creation of a new file, as they do for a sequential file. However, as the file extents fill up, the randomizing algorithm and its corresponding synonym (overflow) processor are more heavily taxed. On the other hand, DAM offers various possibilities for randomizing: if a randomizing algorithm can be established which (more or less) guarantees a unique DASD address for all records, the user may decide to randomize to a record address and move the few synonyms to an overflow area. Moreover, the devices allow the use of algorithms that randomize to track addresses or even cylinder addresses, in this case the number of synonyms may be reduced for more difficult randomizing problems.

If a user can establish a randomizing algorithm that produces unique addresses, an entry-sequenced VSAM organization may be an efficient solution. In such a case he may pre-format an 'empty' entry-sequenced file consisting of dummy records, and replace these with real logical records while loading the file, using addressed-direct processing. This has the advantage of retaining all of the features of VSAM such as data portability, data integrity, and data security, as well as computing logical addresses relative to the beginning of the file. This approach, however, is practical for fixed-length logical records only.

Finally, the following items should be considered in regard to file volatility:

- What are the operational limitations on each method of organization? For example, remember that an ISAM file cannot be sorted, and that ISAM or VSAM files can be processed only by their own access methods respectively.
- If ISAM seems applicable, the key-sequenced VSAM organization is preferable because it offers all of the ISAM facilities and some more of its own. Whether reorganization is required and when, should also be taken into consideration.
- If random processing is required it should be considered whether addressed processing must be performed through pointers from one user's record to another. Such pointers must be created and maintained by the user in his problem program. In a key-sequenced VSAM file they will be difficult to maintain because the physical locations of logical records may change as records are added or deleted.

Many variables must be considered in answering these questions. It is impossible to provide direct answers except in terms of a specific file and a well-defined application.

Filesize

The user must consider the fact that his online capacity is limited. Three important aspects of file organization are affected by the size of a file:

- A sequential file may be written on any number of volumes, that are then mounted and processed consecutively. Each volume may be mounted as needed, but the manual intervention that is required is time consuming.
- An ISAM file must be entirely online for any type of processing; a VSAM file must be entirely online unless only address-sequential processing is done. Then each volume may be mounted as needed.
- A DAM file must be entirely online for any type of processing.

The fact that an ISAM file or a DAM file must be entirely online whenever it is processed, imposes obvious physical restrictions on the maximum filesize. These restrictions are, in fact, the same for a VSAM file when it is involved in random processing; although it is not required that the file is entirely online, frequent changing of volumes during processing will result in much loss of time.

Perhaps less obvious, but just as important is the fact that the possibilities for reorganizing a file are also affected by the online capacity of the system. For the purposes of reorganizing an ISAM file, two files must be defined: the file to be reorganized (as an input file) and the reorganized file (as an output file). Both files must be kept online when they are processed. If the user's installation does not include magnetic tape units, the online DASD capacity must be sufficient to accommodate both files at the same time (assuming that a user does not want to punch his entire file onto cards). If his installation does include magnetic tape units, the reorganization may be accomplished in two steps. The first step consists of reading the ISAM file to be reorganized and writing it in a reorganized form to magnetic tape; the second step consists of loading the reorganized data from the magnetic tape to disk. In this way the maximum size of an ISAM file can be doubled. This is shown in Figure 3.7.

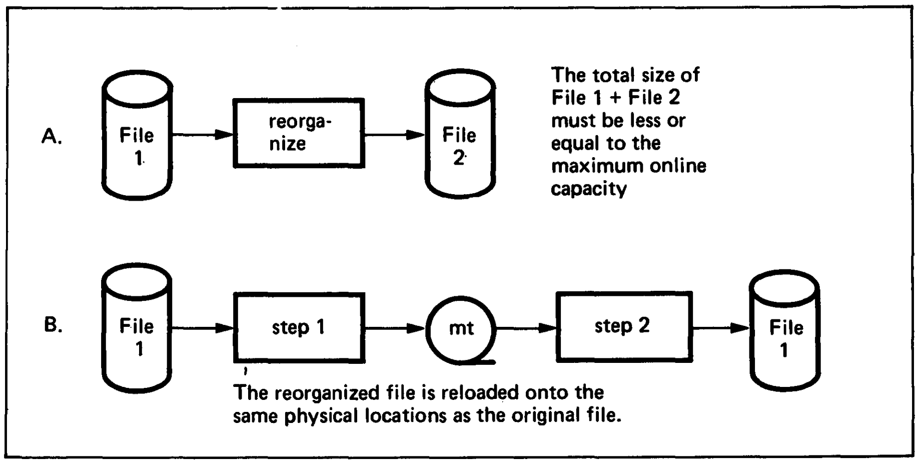

A.
File 1
reorga-nize
File 2
The total size of File 1 + File 2 must be less or equal to the maximum online capacity
B.
File 1
step 1
mt
step 2
File 1
The reorganized file is reloaded onto the same physical locations as the original file.

Figure 3.7. Two possibilities of reorganizing an ISAM file
A. Without using magnetic tape. B. Using magnetic tape as an intermediate storage medium. Using magnetic tape as an intermediate storage medium effectively doubles the maximum filesize.

Response Time to Inquiries

(An inquiry is a request for information from storage. Thus, it may be a request for a single record, for a number of records, or for an entire file, in order to obtain information. There are very advanced and specialized inquiry systems, for example airline-reservation systems. A system like this uses a number of remote terminals from which inquiries are issued and to which the computer supplies the information required. A user, however, may also make use of inquiry by way of a set of programs that are called into a high-priority partition, and that react on control cards in the card reader. He even may consider a normal program as an inquiry program, since any program may deliver information. The main issue of an inquiry is that the request for information normally concerns only a small portion of a file.)

One of the important advantages of a computer system with a direct access storage device is the possibility of answering inquiries without the need for processing complete files. Not all applications require inquiry capability. In many data processing installations there are no inquiry applications at all. Where inquiry is required, however, response time to inquiry is a vital consideration. The less critical the response time, the greater the choice of organization and retrieval options.

The user should consider the following:

- Can the answer to an inquiry wait until the next, batched, sequential updating of the relevant file? If it can, then inquiries become an additional transaction type and are processed sequentially with all other transactions against the file. Data organization, in this case, should be either sequential or indexed sequential. If the response provided by this method is not fast enough, random access is required.
- Can the answer wait until the end of the present computer run? If so, the relevant file is mounted at the completion of the current job, the inquiry program is loaded, and the file is processed to produce the required answers. Obviously, the time delay involved here varies considerably, depending on the job that is in process when the inquiry arrives. If the answer to an inquiry cannot wait, the current program must be terminated. In this case, problems may arise with respect to the status of the current job, problems that may be solved with the checkpoint/restart procedure that is included in DOS/VS. (See: *Checkpoint/Restart*, later in this section.) In a multiprogramming environment it may be possible to have the partition with the highest priority handle the inquiry jobs, while the partitions with a lower priority continue processing the normal jobstream.

Applying Criteria to Sample Files

This chapter illustrates the choice of file organization and retrieval methods for some typical sample files. The characteristics of the sample files were chosen arbitrarily. Different characteristics could be attributed to files with similar functions. The examples are furnished to demonstrate the application of the criteria that have been discussed to specific file characteristics, and to show the best organization and retrieval methods under these circumstances. In all cases, adequate online storage capacity was assumed.

Sample file 1: Table File.

Characteristics:	The file is stable and requires few changes and infrequent additions and deletions. Normal processing involves retrieval only.
Organization:	Indexed sequential.

Sample file 2: Payroll File.

Characteristics:	The file has generally low volatility, and a relatively low level of additions and deletions. However, there is a high activity rate. Processing for each period involves updating a high percentage of the year-to-date master payroll information. Batching of transactions (time cards, changes, etc.) is normal. Fast response to inquiries is not required.
Organization:	Sequential.

Sample file 3: Wholesale Inventory File.

Characteristics:	The file has moderate volatility and activity. Normal transactions may be batched for posting once or twice a day. Recurring stock status, activity, and reorder reports are sequential. Response to inquiries concerning availability and stock level is required within one hour.
Organization:	Indexed sequential; additional processing requirements may change this choice to random.

Sample file 4: Online Inventory, Parts.	
Characteristics:	The file has a low volatility but a high activity. Transactions are processed as they are received. Responses to inquiries concerning availability and stock level are required within two minutes. Recurring stock status, activity, and reorder reports are sequential but are only produced bimonthly.
Organization:	Random (likely), possibly indexed sequential.

Sample file 5: Accounts Receivable File.	
Characteristics:	The file has a low volatility and low activity. Transactions are combined in batches for daily posting. Billing is cyclic. Statements are written throughout the month by sequentially retrieving logical records from the file within specific limits. Inquiries are processed twice a day.
Organization:	Indexed sequential.

Summary

The method of organization best suited to a particular file of DASD records depends upon many factors. These factors must be analyzed for each file in any one particular application. Often, more than one organization scheme could be considered for the same file. In one application, records could be processed purely at random; in another, the same records could be processed in sequence by various control fields. A file such as this would have to be analyzed to determine whether it should be organized:

- Randomly, thus keeping processing time to a minimum during one run but possibly negating the advantage of sequential organization during another.
- Sequentially, thus minimizing the time required to produce reports but increasing the time used in updating.
- Randomly for updating and then sorted into sequence for reports.

The decision would depend on the nature of the file. Other considerations might be:

- Can transactions be batched and sorted before processing, or must they be processed as they occur?
- Is the activity distributed throughout the file in such a manner as to warrant passing the entire file when updating?
- Would the processing time saved by sorting warrant the time and effort required?

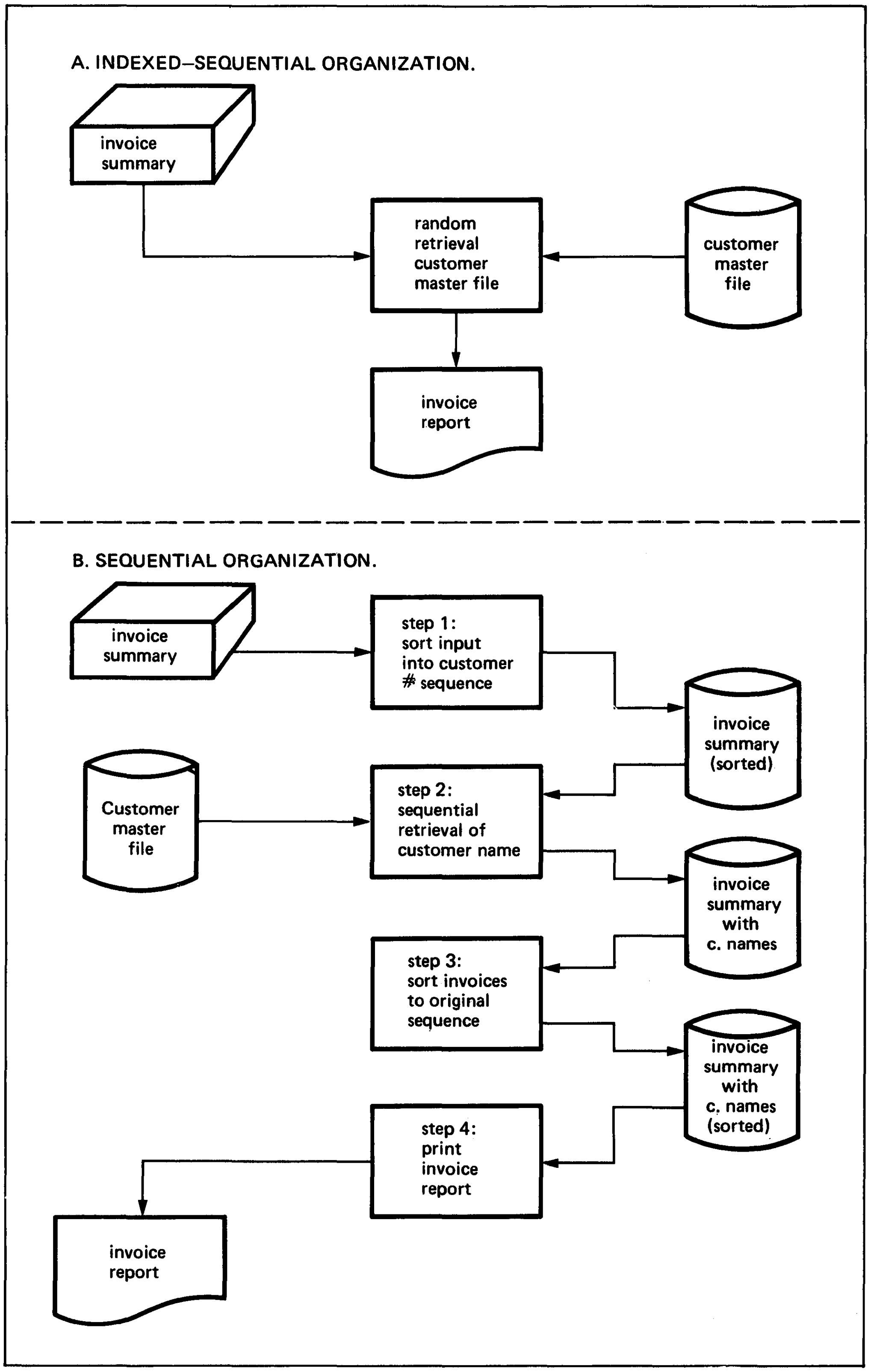
Questions of this kind apply to each file in an installation. In choosing organization methods, the overall processing objectives of the system must be kept in mind at all times.

Random Retrieval Consideration

Many files that could be organized sequentially, are nevertheless organized as indexed sequential files in order to facilitate system design. It is often possible to reduce the number of peripheral operations by using random retrieval from an indexed sequential file. This is true, for example, for files that have fields in their records that are used in several jobs.

As an example, assume that invoice summary cards are to be listed in the sequence of invoices. Further assume that the cards do not contain customer names but that these names are required in the listing. Customer names may be obtained from a separate customer master file through random retrieval (if that file is organized as an indexed sequential file) or through sequential retrieval. In the latter case the invoice summary cards must first be sorted into customer number sequence, which is the sequence of the customer master file. Figure 3.8 illustrates the two solutions and shows the additional steps that are required if the customer master file is organized as a sequential file.

The example for the sequential file is a typical procedure for sequential processing equipment. If this job were run frequently, system design considerations would probably preclude the use of a sequential file organization, and point to an indexed sequential organization.


A. INDEXED–SEQUENTIAL ORGANIZATION.
invoice summary
random retrieval customer master file
customer master file
invoice report
B. SEQUENTIAL ORGANIZATION.
invoice summary
step 1: sort input into customer # sequence
invoice summary (sorted)
Customer master file
step 2: sequential retrieval of customer name
invoice summary with c. names
step 3: sort invoices to original sequence
invoice summary with c. names (sorted)
step 4: print invoice report
invoice report

With sequential organization, several extra steps are required to get the same result.

Figure 3.8Indexed sequential vs sequential organization

High-Level Language Considerations

The access methods introduced in the preceding chapters were described in an assembler language context. Most high-level programming language users can make use of these facilities in their own programming language. An overview of the access methods in relation to high-level programming languages is given in Figure 3.9.

Access Method / Language	SAM	ISAM	VSAM	DAM		BTAM QTAM
				Using Key and Track Reference	Using Record ID and Track Reference	
ANS COBOL	YES	YES	YES	YES	YES	NO
FORTRAN	YES	NO	NO	NO	YES[1]	NO
PL/1	YES	YES	Through ISAM Interface Program Only	YES	YES[1]	NO
RPG II	YES	YES	Through ISAM Interface Program Only	YES	YES	NO
Assembler SCP	YES	YES	YES	YES	YES	YES

Note: The assembler is included in this table for comparison purposes.

[1] The direct access support is implemented by having the user specify the relative record number within the file.

Figure 3.9. Access methods supported for the high-level programming languages

Input/Output Control System

In a modern computer system like the IBM System/370, many different functions can be performed simultaneously, independently of each other. For example, different I/O devices may transfer data to and from the CPU at the same time, while a problem program may be also processing other data. Or since the design of the computer allows multiprogramming, more than one program may be loaded into storage, each of them expecting to have control over the system.

It is the task of an operating system to make sure that system resources are used efficiently, thereby assuring maximum throughput. This task includes monitoring the job flow without the need for frequent manual intervention, keeping track of idling I/O devices, and detecting I/O requests for those devices at the proper moment.

Data management is also part of an operating system. The routines that perform functions such as actual reading and writing, blocking and deblocking, label processing, and error recovery are referred to collectively as the Input/Output Control System (IOCS). The IOCS functions are performed without intervention from the user, who normally is not even aware of the fact that IOCS is doing the job for him.

The details of IOCS that are discussed in the chapters following are mainly addressed to programmers who use the DOS/VS Assembler language. For high-level programming language users, while reading of these chapters may be of interest, it is not mandatory.

The Input/Output Control System acts as an interface between the user's file processing routines in the problem program and the data in external storage. IOCS is regarded as having two parts (Figure 3.10):

- logical IOCS (LIOCS)
- physical IOCS (PIOCS).

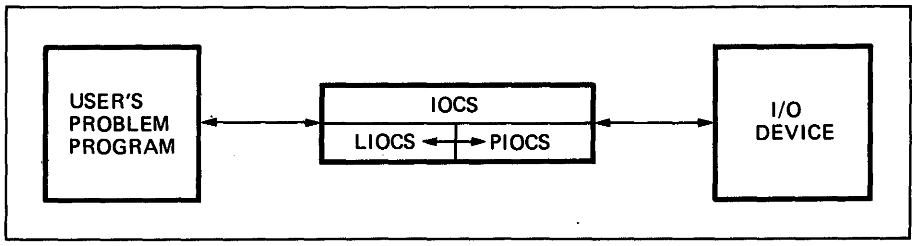


Figure 3.10. Logical IOCS (LIOCS) and Physical IOCS (PIOCS)
LIOCS handles logical data files, whereas PIOCS handles the physical transfer of data in those same files.

LIOCS performs all functions a user needs to locate and access logical records for processing. LIOCS routines are linked with, and executed as part of, the user's problem program. As shown in Figure 3.10, they provide an interface between the user's file processing routines and the PIOCS routines.

The term LIOCS refers to routines that perform the following functions:

- Blocking and deblocking of logical records
- Switching between I/O areas when two areas are specified for a file
- Handling end-of-file and end-of-volume conditions
- Issuing requests to PIOCS to execute appropriate channel programs.

SAM, DAM, or ISAM LIOCS routines are assembled from *logic modules,* which are discussed in a separate chapter later in this section. The user, when including LIOCS routines in his program, has two options:

- The routines may be assembled with his *source* program

- A selected group of LIOCS object routines may be kept in the relocatable library. They will then be called by the linkage editor when unresolved *EXTRN* statements are encountered in the object program at the moment it is edited.

VSAM routines are loaded from the core image library when a VSAM file is opened. The user does not have to assemble or link-edit VSAM modules.

LIOCS is always used for programs written in high-level programming languages. Programmers using those languages will never be aware of the fact that IOCS is performing I/O functions. Assembler programmers have a choice between the use of LIOCS which implies the use of PIOCS as well, or PIOCS only. When using PIOCS only, these programmers must construct their own channel programs; when using LIOCS the channel programs are included in the LIOCS routines. Even the Assembler programmer is normally not aware of IOCS, although he has more means of controlling it than a high-level language programmer has.

LIOCS operates on logical data files, the characteristics of which are specified by the programmer in his programs. These specifications include all or part of the following information:

- file name
- I/O device type
- organization structure
- record format and blocksize
- number of I/O areas and their locations
- location and field length of record identification fields (control fields)
- labeling procedures
- error options
- other optional information.

The specifications above differ from the physical characteristics of the file in that PIOCS knows only the actual location of the data being accessed, and other information about the physical I/O device being used.

The following text is especially focussed on assembler language. The high-level language programmer should consult the appropriate reference manuals to find the proper way for him of specifying the characteristics of his data and the means of processing that data.

LIOCS consists of a number of routines which are called IOCS logic modules and which access logical files. Each logic module is generalized, and designed to perform I/O operations on a particular type of file. A module is tailored according to the specifications made by the programmer in his program, by means of *declarative macro instructions*, which are used to specify the characteristics of a specific file.

PIOCS is that part of IOCS that controls the actual transfer of data between an external storage device and virtual storage. The PIOCS routines are permanently in lower real storage, as part of the control program (supervisor). They supervise the execution of channel programs supplied by the problem programs, without regard to the logical content, format, or the organization of the data being transferred.

PIOCS includes facilities for:

- Scheduling and queuing I/O operations
- Checking for, and handling of, error conditions and other exceptional conditions relating to I/O devices
- Handling I/O interruptions to maintain maximum I/O speeds without burdening the user's problem program.

PIOCS consists of the following routines:

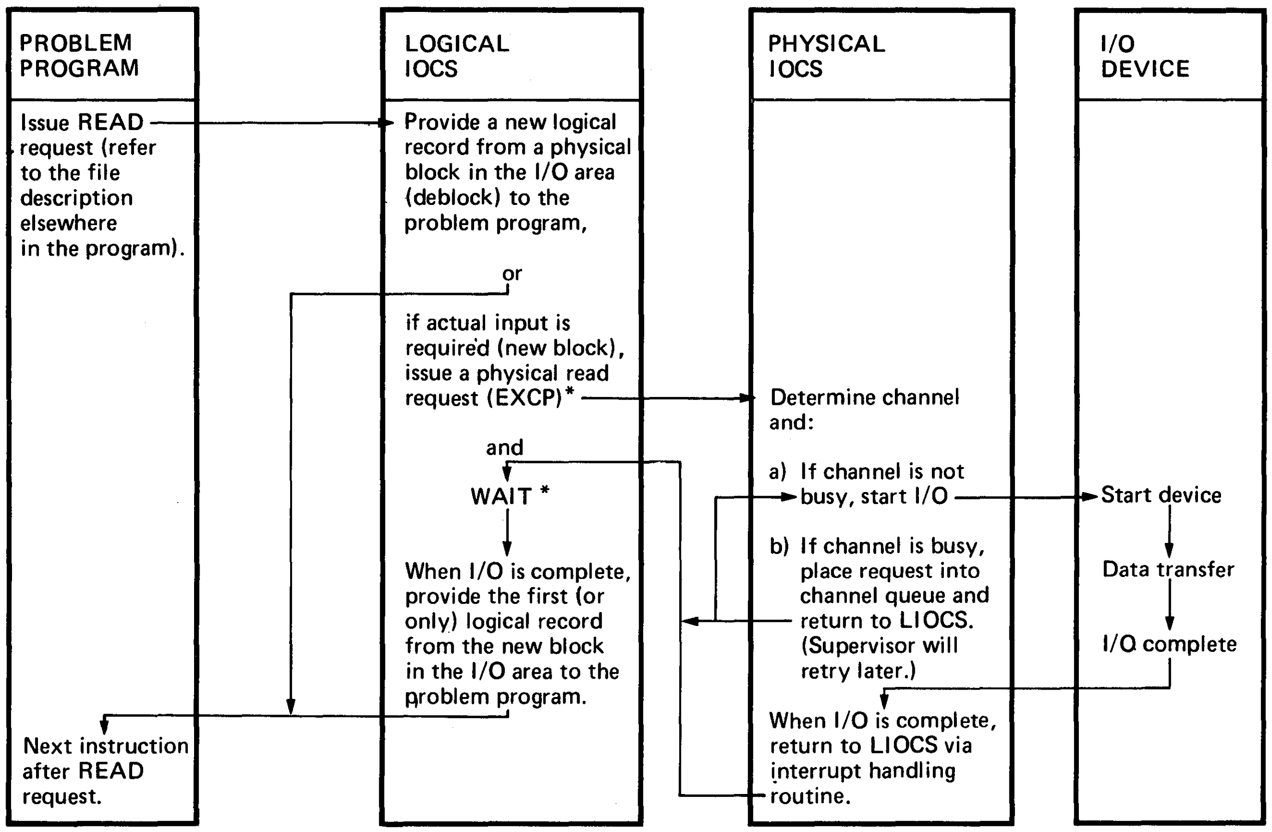
- Start I/O routine
- Interrupt routine
- Channel scheduler
- Device error routines.

All programs make use of these PIOCS routines, which are invisible to the user.

Figure 3.11 provides an example of how the user's program, LIOCS, and PIOCS are clearly separated. It shows the kind of functions performed to obtain a new logical record for processing from an input file.

Logical IOCS (LIOCS)

LIOCS is based on a set of IBM-supplied macro instructions. A full description of the macro system is supplied in *OS/VS and DOS/VS Assembler Language*, GC33-4010; *DOS/VS System Management Guide*, GC33-5371; *DOS/VS Supervisor and I/O Macros*, GC33-5373. The following text discusses the macro system in its relationship to LIOCS only.


PROBLEM PROGRAM
LOGICAL IOCS
PHYSICAL IOCS
I/O DEVICE
Issue READ request (refer to the file description elsewhere in the program).
Provide a new logical record from a physical block in the I/O area (deblock) to the problem program,
or
if actual input is required (new block), issue a physical read request (EXCP)*
and
WAIT *
When I/O is complete, provide the first (or only) logical record from the new block in the I/O area to the problem program.
Determine channel and:
a) If channel is not busy, start I/O
b) If channel is busy, place request into channel queue and return to LIOCS. (Supervisor will retry later.)
When I/O is complete, return to LIOCS via interrupt handling routine.
Start device
Data transfer
I/O complete
Next instruction after READ request.

* Physical I/O instructions are issued by means of two PIOCS macro instructions: EXCP and WAIT. These are explained in the detailed discussion of PIOCS.

Figure 3.11. Example of the use of LIOCS and PIOCS in reading a record
The actual coding of LIOCS is part of the problem program and taken from IBM-supplied logic modules; the coding of PIOCS is integrated within the Supervisor.

Macro System

The main purpose of macro instructions is to reduce repetition in source coding, and consequently the probability of error. For example, IBM provides a set of macro instructions which allow the programmer to perform I/O on a logical level. Thus, the programmer can concentrate on the actual problem that he is expected to solve. He needs not, for example, be concerned about blocking and deblocking records, or specifying channel programs for input and output. These functions are performed by DOS/VS which responds to the macro instructions issued in a problem program.

The macro system has two basic parts:

1. Macro definitions.
   These are generalized functions, written in the *macro definition language* as source statements.

2. Source program macro instructions.
   These are the symbolic instructions used in the problem program. Within the macro system, IBM supplies the following types of macro instructions and their definitions:

   - Supervisor communication macro instructions.
     These communicate with the supervisor and give access to the communication region. The supervisor communication macro instructions are not discussed in this manual, since here we are concerned with input and output.

   - Declarative macro instructions.
     These macro instructions can be divided into two classes:

     1. File description macro instructions.
        They specify the characteristics of a specific file to be processed. For example, the DTFMT (Define The File for Magnetic Tape) macro instruction specifies the characteristics of a particular magnetic tape file, such as blocksize, and I/O areas used.
     2. Logic module generation macro instructions.
        They give information about the type of logic module to be generated. A logic module is an object code routine that can handle specific conditions typical for a certain type of file. For example, the CDMOD (CarD MODule) macro instruction generates a logic module to handle card files.

   - Imperative I/O control macro instructions.
     These macro instructions identify the kind of I/O operation that is desired. For example, a GET macro instruction indicates that a user wants a logical record to be retrieved.

A macro instruction (*macro*) has a *name* (for example *DTFMT*), usually accompanied by a group of *parameters.* The parameters specify which options are used. For example, some of the parameters of the DTFMT macro are the address of the I/O area, the entry point of a routine that handles end-of-file conditions for input files, and the size of a logical record. The macro name is used to locate the macro definition in the library, whereas the parameters are used to select the proper source code from this macro definition, according to the options chosen. The result of this location and selection (which is performed by the assembler processor) is a set of assembler code called a *macro expansion.* The macro expansion is inserted into the user's problem program, as shown in Figure 3.12. This macro expansion has the format of normal assembler language and is processed into machine object code along with the user's source statements.

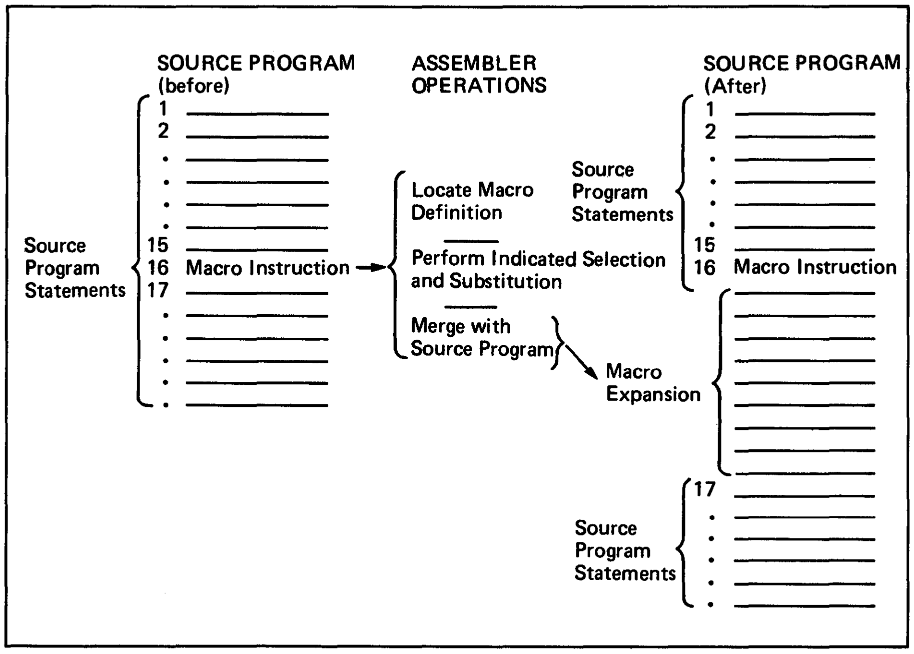


Figure 3.12. Macro processing
The assembler processor selects specified elements of a macro definition for insertion into the user's problem program.

IBM provides a number of pre-written macro definitions and specifies the macros that programmers can use. A user is free to write other macro definitions and to add them to the already existing library. The IBM-supplied macro instructions are described in full detail in *DOS/VS Supervisor and I/O Macros*, GC33-5373.

In the following text, general information is supplied about the following types of I/O macros:

- File description macros (*DTFxx*)
- Imperative macros (*GET, PUT, OPEN, CLOSE, etc*)
- Logic module generation macros (*xxMOD*).

File Description Macros
Whenever a file is to be processed under LIOCS, the characteristics of that file must be specified by means of a *DTFxx* macro:

- For sequential processing (SAM):

DTFCD	for card files
DTFPR	for printer files
DTFCN	for console (printer-keyboard files)
DTFMT	for magnetic tape files
DTFSD	for DASD files
DTFPT	for paper tape files

DTFOR	for optical character reader files (OCR)
DTFMR	for magnetic ink character reader files (MICR)
DTFDI	for device independent files
DTFSR	for serial device files
DTFDU	for diskette files

- For random processing (DAM):

DTFDA	for DASD files only

- For processing through ISAM:

DTFIS	for DASD files only

- For processing through VSAM:

ACB	for VSAM, the ACB macro specifies the Access-method Control Block (*ACB*); the ACB macro is similar to the DTF macro for the other access methods.

Figure 3.13 is an example of a DTFxx macro instruction; it describes all of the options that can be used for a magnetic tape file; Figure 3.14 shows the DTFMT macro instruction which specifies the characteristics of one particular magnetic tape file.

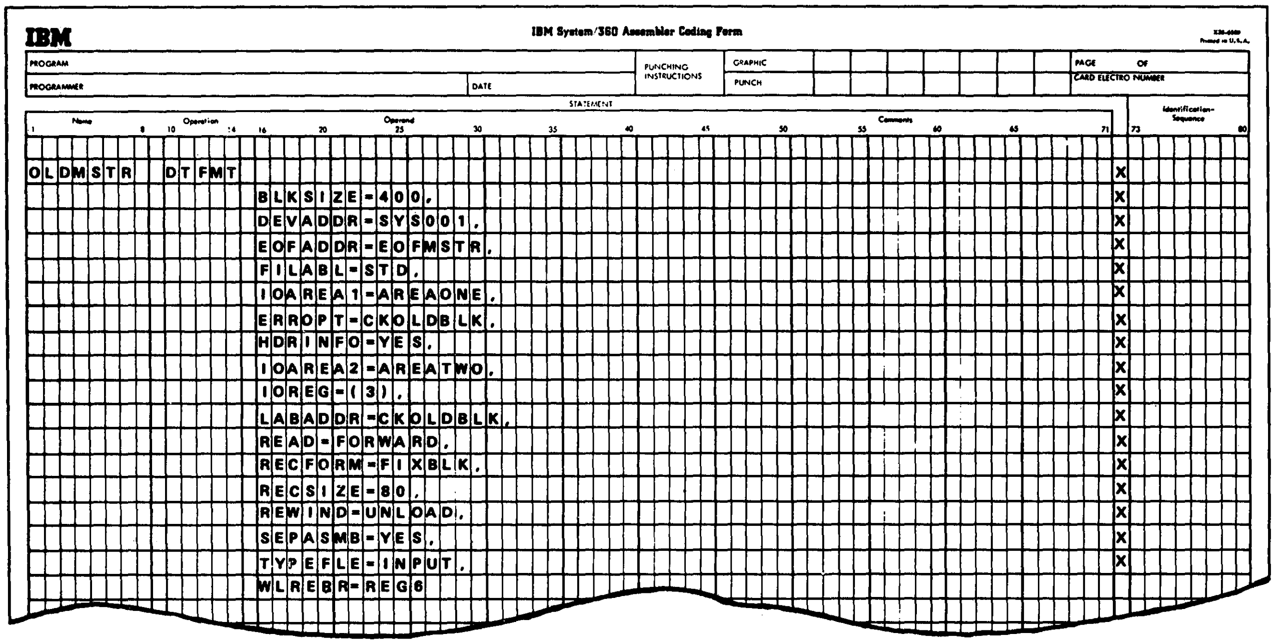
Applies to					
Input	Output	Work			
X	X	X	M	BLKSIZE=nnnnn	Length of one I/O area in bytes (maximum = 32,767).
X	X	X	M	DEVADDR=SYSxxx	Symbolic unit for tape drive used for this file.
X		X	M	EOFADDR=xxxxxxxx	Name of your end-of-file routine.
X	X	X	M	FILABL=xxxx	(NO, STD, or NSTD). If NSTD specified, include LABADDR. If omitted, NO is assumed.
X	X		M	IOAREA1=xxxxxxxx	Name of first I/O area.
X	X		O	ASCII=YES	ASCII file processing is required.
X	X		O	BUFOFF=nn	Length of block prefix if ASCII=YES.
X			O	CKPTREC=YES	Checkpoint records are interspersed with input data records. IOCS bypasses checkpoint records.
X	X	X	O	ERREXT=YES	Additional errors and ERET are desired.
X	X	X	O	ERROPT=xxxxxxxx	(IGNORE, SKIP, or name of error routine). Prevents job termination on error records.
X	X	X	O	HDRINFO=YES	Print header label information if FILABL=STD.
X	X		O	IOREG=(nn)	Register number. Use only if GET or PUT does not specify work area or if two I/O areas are used. Omit WORKA. General registers 2-12, written in parentheses.
X	X		O	LABADDR=xxxxxxxx	Name of your label routine if FILABL=NSTD, or if FILABL=STD and user-standard labels are processed.
X			O	LENCHK=YES	Length check of physical records if ASCII=YES and BUFOFF=4.
X	X	X	O	MODNAME=xxxxxxxx	Name of MTMOD logic module for this DTF. If omitted, IOCS generates standard name.

Figure 3.13. General format of the DTFMT macro(Part 1 of 2)

Applies to					
Input	Output	Work			
		X	O	NOTEPNT=xxxxxx	(YES or POINTS). YES if NOTE, POINTW, POINTR, or POINTS macro used. POINTS if only POINTS macro used.
X	X	X	O	RDONLY=YES	Generate read-only module. Requires a module save area for each task using the module.
X		X	O	READ=xxxxxxx	(FORWARD or BACK). If omitted, FORWARD assumed.
X	X	X	O	RECFORM=xxxxxx	(FIXUNB, FIXBLK, VARUNB, VARBLK, SPNUNB, SPNBLK, or UNDEF). For work files use FIXUNB or UNDEF. If omitted, FIXINB is assumed.
X	X		O	RECSIZE=nnnn	If RECFORM=FIXBLK, no. of characters in record. If RECFORM=UNDEF, register number. Not required for other records. General registers 2-12, written in parentheses.
X	X	X	O	RECWIND=xxxxxx	(UNLOAD or NORWD). Unload on CLOSE or end-of-volume, or prevent rewinding. If omitted, rewind only.
X	X	X	O	SEPASMB=YES	DTFMT is to be assembled separately.
	X		O	TRMARK=NO	Prevent writing a tapemark ahead of data records if FILABL=NSTD or NO.
X	X	X	O	TYPEFLE=xxxxxx	(INPUT, OUTPUT, or WORK). If omitted, INPUT is assumed.
	X		O	VARBLD=(nn)	Register number, if RECFORM=VARBLK and records are build in the output area. General registers 2-12 are written in parentheses.
X			O	WLRERR=xxxxxxxx	Name of wrong-length-record routine.
X	X		O	WOKA=YES	GET or PUT specifies work area. Omit IOREG.

M = Mandatory; O = Optional

Figure 3.13. General format of the DTFMT macro (Part 2 of 2)


IBM System/360 Assembler Coding Form

```
OLDMSTR  DTFMT                                                         X
               BLKSIZE=400,                                            X
               DEVADDR=SYS001,                                         X
               EOFADDR=EOFMSTR,                                        X
               FILABL=STD,                                             X
               IOAREA1=AREAONE,                                        X
               ERROPT=CKOLDBLK,                                        X
               HDRINFO=YES,                                            X
               IOAREA2=AREATWO,                                        X
               IOREG=(3),                                              X
               LABADDR=CKOLDBLK,                                       X
               READ=FORWARD,                                           X
               RECFORM=FIXBLK,                                         X
               RECSIZE=80,                                             X
               REWIND=UNLOAD,                                          X
               SEPASMB=YES,                                            X
               TYPEFLE=INPUT,                                          X
               WLREBR=REG6
```

Figure 3.14. Sample DTFMT macro for a particular file

Imperative Macros
IBM supplies a variety of imperative macros, each of them performs a specific action on a logical record, a physical block, or a file. For example, when an input file is processed sequentially by SAM, each next logical record is obtained from that file by issuing a GET macro. Another example is label processing that is performed automatically in response to the OPEN and CLOSE macros. Also device control, such as backspace tape, can be performed; the CNTRL macro provides an easy and flexible way of doing this. In general, imperative macros provide direct control over many varying functions, on a logical level, so that the programmer can concentrate his effort on the problem program itself.

Files must be defined to IOCS by means of one of the DTFxx declarative macros before any of the imperative macros can be used for processing.

The imperative macros can be divided into three types:

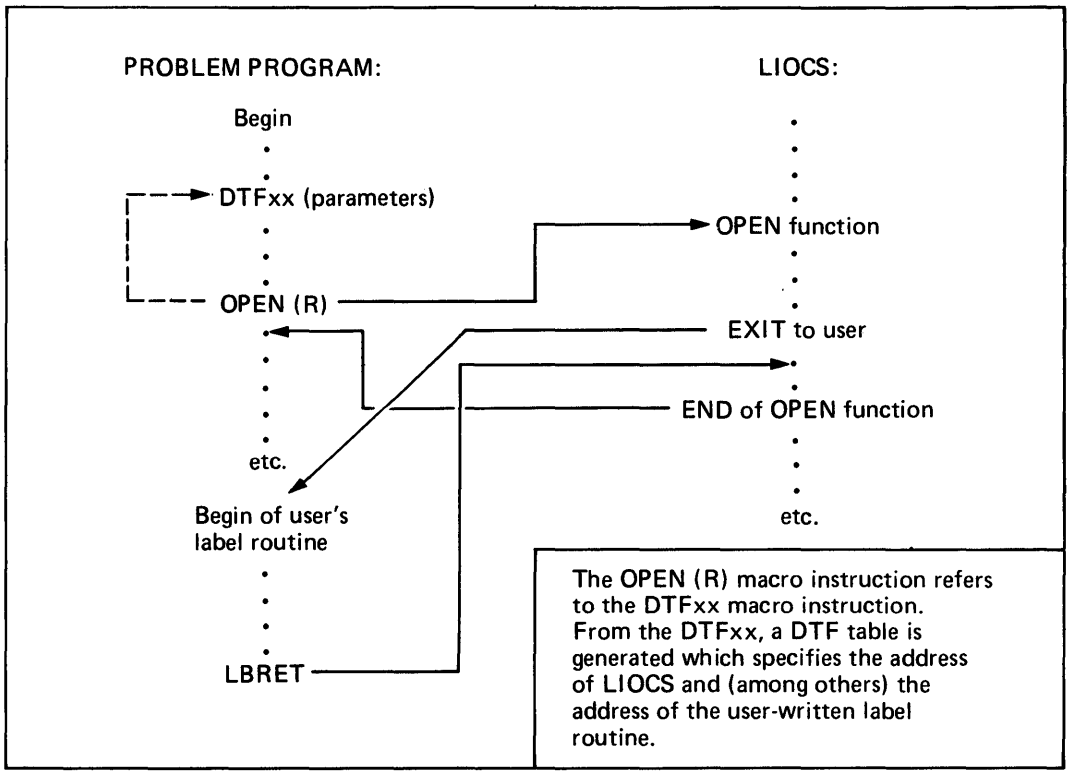
- Initialization macros
- Processing macros
- Completion macros.

Initialization macros provide secundary functions that must be performed before a file can be processed. These functions include label processing and, for example, setting a magnetic tape to load point, and reading a first block of an input file into a buffer.
The following initialization macros are available:

- *OPEN*
- *OPENR*
- *LBRET*

OPEN and *OPENR* initialize a file for processing. The difference between the two is that *OPENR* must be used in self-relocating programs, whereas *OPEN* is used in all other cases. If a user specifies in the DTFxx macro that a user-written label processing routine is present in the problem program for additional label processing (*LABADDR=name*), the OPEN, CLOSE, and end-of-volume routines of LIOCS will contain a link with that user-written routine (CLOSE and end-of-volume are discussed below under *Completion macros*).

User-written label handling routines can be used for additional label checking on input, or for writing additional user labels on output; such routines must always end with a *LBRET* macro which will cause control to be returned to LIOCS so that execution of the OPEN(R), CLOSE, or end-of-volume routine can be resumed and completed. Figure 3.15 illustrates the use of a user-written label handling routine as an extension of the LIOCS OPEN routine.



The OPEN (R) macro instruction refers to the DTFxx macro instruction. From the DTFxx, a DTF table is generated which specifies the address of LIOCS and (among others) the address of the user-written label routine.

Figure 3.15. Initialization macros
OPEN(R) initializes a file for processing, and may link to a user-written label routine that ends with LBRET.

Processing macros provide functions directly related to actual input and output. Examples of such functions are:

- Actual reading and writing
- Stacker selection control and printer carriage control
- Rewinding and backspacing of magnetic tape
- Eject documents on OCR
- Seek to a specified track on DASD.

A large variety of processing macros is supplied by IBM. The functions performed depend on the options chosen and the access method used. Explanations of the processing macros, in relation to specific access methods, are provided in the next section of this manual: "Access methods".

Completion macros provide secondary functions that must be performed after processing a file and before the program is terminated. These functions include label processing and, for example, rewinding a magnetic tape. The following completion macros are available:

- *CLOSE*
- *CLOSER*
- *FEOV*
- *FEOVD*

Both *CLOSE* and *CLOSER* deactivate a *file*; the difference between the two is that *CLOSER* must be used in self-relocating programs, whereas *CLOSE* is used in all other cases.

Both *FEOV* and *FEOVD* are used to deactivate a *volume*, while processing *sequentially*. FEOV forces an end-of-volume condition for a magnetic tape volume before sensing a tapemark (*input file*) or reflective marker (*output file*) on magnetic tape. FEOVD forces an end-of-volume condition on a DASD volume before the end-of-volume condition actually occurs.

If an FEOV(D) macro is issued while the last volume of an input file is being processed, it will result in an end-of-file condition for the entire file. LIOCS will then branch to the user's end-of-file routine, the name of which is specified in the DTFxx macro (*EOFADDR=name*).

In most cases the FEOV(D) macro will not be used: end-of-volume conditions and end-of-file conditions are detected by LIOCS as they occur, and necessary actions such as switching to a next volume are performed automatically, without necessitating any action from the user.

The usage of CLOSE(R) and FEOV(D) is illustrated in Figure 3.16.

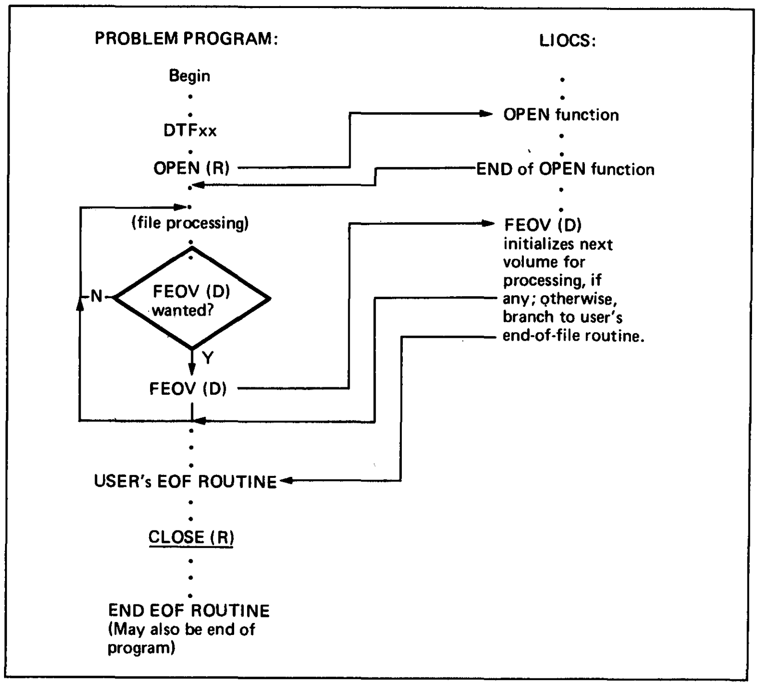

PROBLEM PROGRAM:
LIOCS:
Begin
DTFxx
OPEN (R)
OPEN function
END of OPEN function
(file processing)
FEOV (D) wanted?
N
Y
FEOV (D)
FEOV (D) initializes next volume for processing, if any; otherwise, branch to user's end-of-file routine.
USER's EOF ROUTINE
CLOSE (R)
END EOF ROUTINE (May also be end of program)

Figure 3.16. Completion macros
CLOSE(R) in the user's EOF routine deactivates the file, after FEOV(D) has deactivated the last volume of that file.

Logic Module Generation Macros
From a DTFxx macro, a *DTF table* is generated by the assembler processor. This table links to a logic module which provides the necessary machine instructions to perform the required I/O functions. For example, a logic module reads and writes physical blocks, tests for unusual I/O conditions, blocks and deblocks logical records if necessary, and places logical records into a work area. Most of the imperative macros enter a logic module (through addresses provided in the DTF table) to perform the activity needed.

The activity to be performed depends on the requirements of the problem program and the characteristics of the file to be processed. These characteristics and requirements are specified in the problem program by means of the *xxMOD* macro. According to the parameters of this macro, the programmer has the option of selecting or omitting some of the available functions of a logic module.

It is also possible to have a logic module *name* generated through the DTFxx macro, according to the processing requirements specified there. As a result a *standard logic module* is generated, and in this case the programmer has no means of selecting or omitting any specific function.

Logic modules can be assembled together with the problem program, or separately. A selected group of LIOCS object modules can be pre-assembled and stored in the relocatable library; required logic modules can then be retrieved and included in the user's object program when it is edited by the linkage editor. This eliminates the need for lengthy macro generation each time the program is assembled.

A pre-assembled logic module can be furnished to the linkage editor in three ways:

1. *INCLUDE* the logic module from SYSIPT
2. *INCLUDE* the logic module from the relocatable library
3. *AUTOLINK* the module from the relocatable library

Detailed information about this subject can be found in:

- *DOS/VS System Management Guide*, GC33-5371, and
- *DOS/VS Supervisor and I/O Macros*, GC33-5373.

When many programs make use of a specific logic module, each of them would need an equal amount of space for the module in the core image library, after the module has been included in the program by the linkage editor. If there is only limited space available for the core image library, the user may consider having the logic module removed from the program after the linkage editor process and, instead, have the module stored only once in the core image library, separately from all problem programs. At object time, when the program is to be executed, a *LOAD* macro must then be issued before the first I/O macro (OPEN) to actually load the logic module into virtual storage.

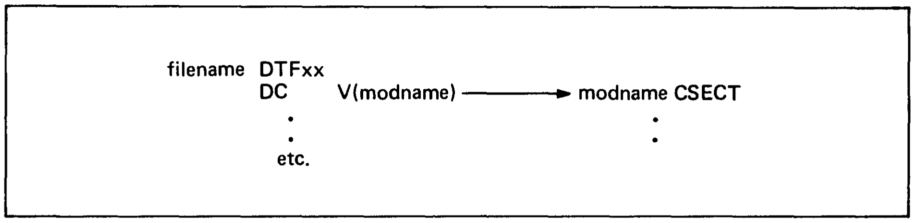
Interrelationships of the I/O Macro Instructions
Imperative macros refer to the DTFxx macro by specifying a filename, as follows:

```
filename DTFxx parameters
          .
          .
         OPEN filename
          .
          .
         GET  filename
          .
          .
         etc.
```

In the macro expansions of OPEN and GET, above, the parameter *filename* is translated into an address constant which refers to the address of the macro expansion of the DTFxx macro:

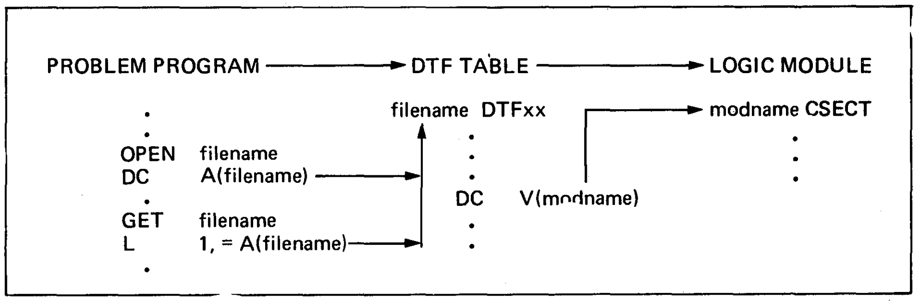
```
        filename DTFxx parameters
                  .
                  .
                 OPEN filename
*                DC   A(filename)
                  .
                  .
                 GET  filename
*                L    1,=A(filename)
                  .
                  .
                 etc.
```

The expansion of the DTFxx macro is called the *DTF table*, and one of the items in this table is the address of the logic module associated with the file. One of the parameters of the DTFxx macro is *MODNAME=modname.* The value of *modname* is specified by the user, or generated as a *standard module name.* In the latter case, a module name is determined by the system, in accordance with the requirements specified by the parameters of the DTFxx macro in general. In the DTF table, the parameter *MODNAME=modname* is translated into a V-type address constant specifying the address of the logic module, and the logic module is designed as a named CSECT:



In the examples above, only part of the macro expansions is shown. The reader should note that in the DTF table a V-type address constant is generated. This means that, as was said earlier, the logic module need not be included in the user's problem program when it is assembled; it is called from the relocatable library when the program is edited.

As a result, the user's program, by means of imperative macros, refers to the DTF table for a file, and the DTF table refers in turn to the logic module:



Physical IOCS (PIOCS)

PIOCS normally operates under control of LIOCS so that programmers, being hardly aware of the presence of LIOCS, are not aware of PIOCS at all. Assembler programmers may choose to control PIOCS themselves, however, thereby ignoring the features offered by LIOCS. With PIOCS a programmer can obtain maximum flexibility, as will be explained later in this chapter. It should be noted, however, that all logical functions normally provided by LIOCS such as, for example, the blocking and deblocking of logical records, are entirely the responsibility of a programmer if he chooses for direct control over PIOCS. PIOCS operates on physical blocks only.

PIOCS generally processes in the following sequence: a routine (usually LIOCS) issues an I/O request in the form of a *Supervisor Call* (SVC) instruction accompanied by a *Command Control Block* (CCB). As soon as the I/O request has been satisfied and the operation has been started, control is returned to the requesting routine, without waiting for completion of the I/O operation.
The completion of an I/O operation is detected through an interruption, after which PIOCS tests for errors and again, returns control to the program, taking priorities into account.

The CCB contains information about the I/O device to be accessed, such as the symbolic device name SYSxxx, the location of the first *Channel Command Word* (CCW) of a channel program to be executed, and error options as selected by the programmer. (*LIOCS also creates a CCB that occupies the first 16 bytes of the DTF table which is generated from the DTFxx macro*).

The main element of PIOCS is the *channel scheduler* which is entered through the SVC mentioned above. The channel scheduler determines the channel to be used (by means of the symbolic device address SYSxxx), and places the I/O requests into a queue for that channel. As soon as the channel is not busy, the first request in the queue is taken and the requested I/O operation is started. At the completion of an I/O operation, the currently running program is interrupted, the interrupt routine determines which channel caused the interrupt, checks are made for errors, and control is given to the program with the highest priority that is able to proceed (not waiting for any I/O). This process was, in simplified form, illustrated in Figure 3.11.

Direct Control over PIOCS in Assembler Language

An assembler programmer may choose to control PIOCS directly in his programs. This means that, for each I/O request, he is responsible for supplying a CCB and for issuing the SVC instruction. In addition, the programmer must construct his own channel programs. Finally, he must also provide for all necessary logical functions such as blocking and deblocking logical records.

In return, PIOCS offers all the flexibility a programmer may desire. For example, he is able to use only part of a physical block on input, or to construct a physical block from noncontinuous areas in virtual storage on output, because he is able to perform any typical function in his own channel programs (within the capabilities of the system).

PIOCS Macro Instructions

By means of the macro system (described in the chapters on LIOCS), an assembler programmer is able to write PIOCS macros, allowing him to work with symbolic terms and values rather than with difficult constructions of bytes and bits.

The following PIOCS macros are available:

CCB A CCB (*Command Control Block*) macro must be specified in the problem program for each device that is controlled directly through PIOCS. The CCB macro has the following operands:

- The symbolic device name (*SYSxxx*).

- A *command-list-name* which symbolically specifies the address of the first CCW in a channel program (*ccwname*). If different channel programs use the same device, this address may be modified in accordance with the channel program to be executed, or separate CCB macros may be used for each separate channel program.
- User options, coded into a hexadecimal value (optional).
- A sense address, which indicates that there is a user-written error recovery procedure (optional).

(*When under control of LIOCS, the CCB macro is generated as a result of the DTFxx macro*)

The CCB macro must have a *ccbname*; the general format is:

ccbname   CCB   SYSxxx,ccwname(,optional parameters)

EXCP

The EXCP (*EXecute Channel Program*) macro is used for issuing an I/O request to PIOCS. It is translated into an SVC instruction (which calls the channel scheduler) and a reference to the Command Control Block. This reference can be given in one of two ways:

1. As a symbolic reference to the CCB by means of a *ccbname*.
2. As a reference to a general register which contains the address of the CCB.

The general format of the EXCP macro is:

(name)   EXCP   ccbname (or Rn)

WAIT

PIOCS does not wait for the completion of an I/O operation after the operation has been started. Instead, control is returned to the problem program, which must be sure that it does not start processing data that has not been completely read, or start overwriting an output area before the previous block has been completely written.
A problem program can wait for the completion of an I/O operation by issuing a WAIT macro and by referring, in that macro, to a CCB. The effect of a WAIT macro is another SVC instruction which checks, in the interrupt routine, the status of the I/O operation in process.

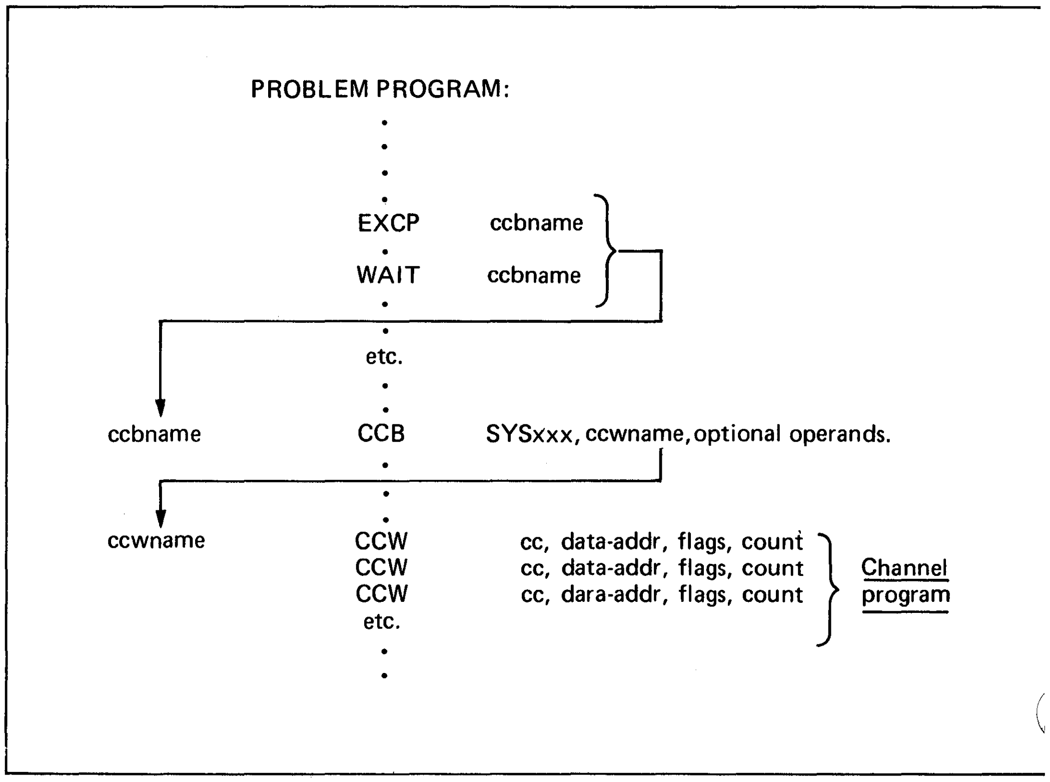
The general format of the WAIT macro is:

(name)   WAIT   ccbname (or Rn)

The reference to a CCB can be made in either way described above for the EXCP macro.

Interrelationships Between the PIOCS Macros

Only three different macros (CCB, EXCP, and WAIT) are used for PIOCS. The relationships between the three macros are shown in Figure 3.17.



PROBLEM PROGRAM:
EXCP ccbname
WAIT ccbname
etc.
ccbname CCB SYSxxx, ccwname, optional operands.
ccwname CCW cc, data-addr, flags, count
CCW cc, data-addr, flags, count
CCW cc, dara-addr, flags, count
etc.
Channel program

Figure 3.17. Relation between PIOCS macros
Note that in this figure the assembler instruction CCW is illustrated as well. The CCW instruction is not a macro, but it is included here because it is the last missing link in PIOCS programming.

PIOCS Programming Considerations

The following paragraphs indicate a few programming problems and explain how they can be solved. Some restrictions are also mentioned.

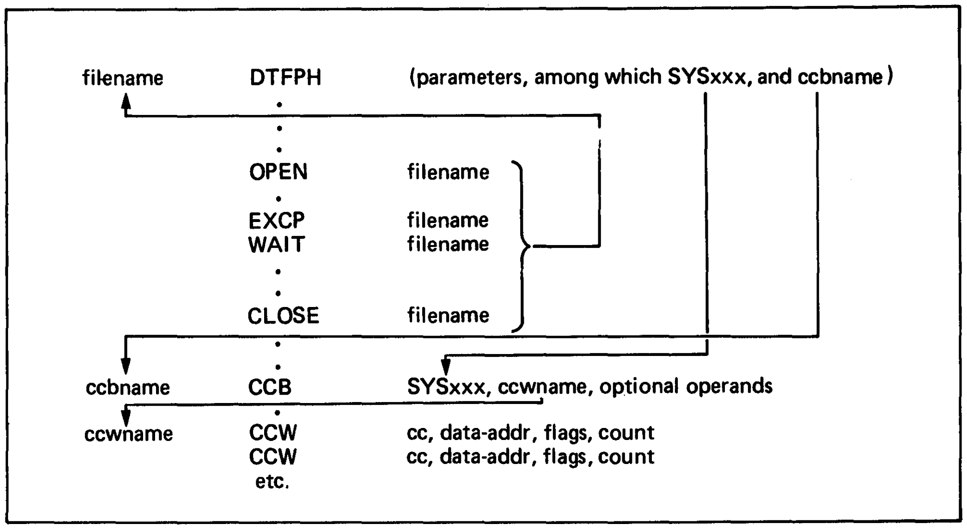
Situations Requiring LIOCS Functions in PIOCS Processing.
In explaining PIOCS it was said that the programmer is responsible for providing all of the logical functions that are normally provided by LIOCS routines. There are, however, two exceptions to this rule. In fact, a programmer *must* use some of the logical functions of LIOCS for these two types of files:

1. DASD files that are file-protected.
2. Magnetic tape files, diskette files, or DASD files that require standard label processing.

These two situations are closely related to data security and data integrity, for which a system can accept responsibility only if it is recognized as the only authority having access to system information.

In either of the two situations above, files must be defined to LIOCS by means of the DTFPH macro. This macro, like any other DTFxx macro for LIOCS, specifies the characteristics of the file. The logic module will provide the minimum facilities necessary for label processing and protecting files, where applicable. Label processing will be performed, as usual, in response to the OPEN(R) and CLOSE(R) macros. In addition, the FEOV macro can be used for volume switchin on output magnetic tape files.

If the DTFPH macro is used, a program may look slightly different from the previous example. As was explained before, the DTF table contains the CCB in the first 16 bytes, so that the EXCP and WAIT macros can now refer to the name of the DTFPH macro. The DTFPH macro in turn contains a parameter *CCWNAME=ccwname*, so that the CCB has the proper reference to the first CCW in the appropriate channel program:



filename DTFPH (parameters, among which SYSxxx, and ccbname)
OPEN filename
EXCP filename
WAIT filename
CLOSE filename
ccbname CCB SYSxxx, ccwname, optional operands
ccwname CCW cc, data-addr, flags, count
CCW cc, data-addr, flags, count
etc.

Figure 3.18 Channel Programming Considerations.
for information about the CCW format and the concepts of data chaining and command chaining, the reader is referred to *System/370 Principles of Operation*, GA22-7000.

- Command chaining retry.
  If a system has been generated to support command chaining retry, the user can use this option for his PIOCS channel programs by setting the *command chaining retry bit* in the CCB to *ON*. Then, if an error that involves retry occurs, the retry will begin with the last CCW executed. If this bit is *OFF*, the entire channel program will be re-executed.

When the command chaining retry bit is *ON*, the user must move the address of the first CCW in the channel program to bytes 9 - 11 of the CCB before an EXCP is issued. This ensures that the CCB always contains the correct CCW address; bytes 9 - 11 are modified by PIOCS for a retry after an error with the address of the CCW to be re-executed, and it is not reset to its original value.

If a command chain is broken by some exceptional condition (for example, wrong length record, or unit exception) that does not result in device error recovery by IOCS, the user can determine the address of the last CCW executed and if necessary, restart the channel program at that point. To obtain the address of the last CCW executed, subtract 8 from the address in bytes 13-15 of the CCB. On a 1403 printer, a command chain is broken after sensing channel 9 or 12. When using command chaining on a 1403, therefore, the program should always check if the entire CCW chain has been executed.

The command chaining retry bit must not be used to read multiple blocks from SYSIPT or SYSRDR. Moreover, this bit should *never* be *ON* for DASD or diskette channel programs.

- Data chaining.
  When performing data chaining, the CCW in a channel program should all contain the proper command code of the operation to be executed, in order to ensure proper I/O error recovery. In normal cases where nothing goes wrong, the command code is not used if a preceding CCW has the data chaining bit *ON*. In case of an error, however, recovery frequently depends on the command being executed and the command code in the last CCW is often examined. In such a case, a 'dummy' command code might prevent error recovery.

- DASD channel programs.
  The user should begin a DASD channel program with a full seek (command code X'07'); if the channel program contains embedded seeks, they should be full seeks as well.

If embedded seeks are used, a program cannot run under DASD file protection, nor can it take full advantage of the seek separation feature. With DASD file protection, an embedded seek causes cancellation of the program in error.

The seek separation feature initiates a seek and separates it from the channel program chain. Thus, the channel is available for other input or output operations on the same channel. The seek separation feature, however, applies only to the first seek in a channel command chain.

When executing a channel program (Figure 3.19), the supervisor sets up a channel program with three commands:

1. A *Seek* that is identical to the user's seek.
2. A *Set File Mask* that prevents other X'07' seeks from being executed.

3. A *Transfer In Channel* (TIC) command that transfers control to the command following the user's seek.

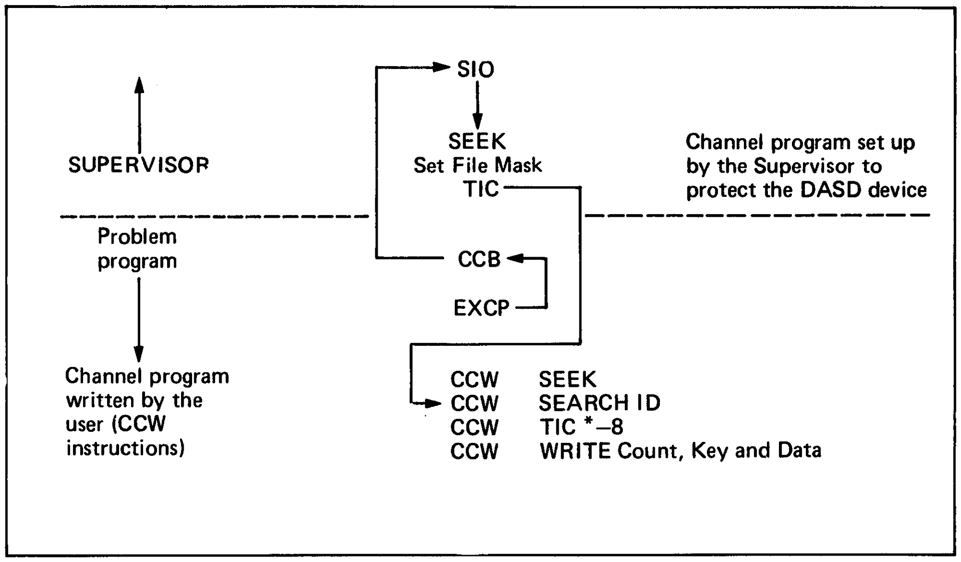

SIO
SEEK
Set File Mask
TIC
SUPERVISOR
Channel program set up by the Supervisor to protect the DASD device
Problem program
CCB
EXCP
Channel program written by the user (CCW instructions)
CCW SEEK
CCW SEARCH ID
CCW TIC *–8
CCW WRITE Count, Key and Data

Figure 3.19. Example of channel programming a file protected DASD file
By setting the file mask, the supervisor prevents further seeks on DASD.

- Diskette channel programs.
  The user must begin a diskette channel program with a define operations command (command code X'2F'). This command is intended for use during program initiation, and sets the operating mode for a file during program execution. It defines whether read or write operations will be done; if write operations are to be done, the define operations command determines how many writes will be done beween seeks. This command must be reissued to change the mode of operations on the file.

  Following the define operation should be a seek (command code X'07').

  Following the seek should be the read or write CCWs. You can chain 1, 2, 13, or 26 read/writes.You have to check however, where your chaining begins.With 26 chained records, for instance, you have to start chaining on track boundary.Record length can be chosen freely up to 128 bytes. If write operations are being performed, a NOP command should be chained to the last write command to ensure that any errors occurring on this channel program are returned.

- Console (printer-keyboard) buffering.
  If the console buffering option is specified at system generation, and if the printer-keyboard is assigned to SYSLOG, throughput on *output* under PIOCS can be increased for physical blocks that do not exceed 80 characters. This is accomplished by starting the I/O command and returning to the problem program before the output is completed.

Blocks are always printed in a first-in-first-out (FIFO) order, regardless of whether the output blocks are buffered (queued on an I/O completion basis) or not.

Console buffering is performed on output only if the following conditions are maintained:

- The actual block to be written must not exceed 80 characters.
- No data chaining or command chaining must be performed.
- The acceptance of unrecoverable I/O errors, of posting at device end, or of user error routines must not be indicated in the CCB associated with the operation.
- Sense information must not be requested by the CCB.

- Alternate tape switching.
  Alternate tape drives cannot be used on *input* processed by PIOCS.

  On output, automatic alternate tape drive switching can be done through the DTFPH and FEOV macros. The FEOV *(Force End Of Volume)* macro writes the trailer label sets (standard labels and any desired user labels), and deactivates the current volume. The next volume is then mounted on the alternate tape drive, and IOCS writes the header label sets (standard labels and any desired user labels) on the new volume.

- Bypassing embedded checkpoint records on magnetic tape (see Checkpoint/Restart").

  Checkpoint information is written as a set of magnetic tape records:

  - One 20-byte header record;
  - One status descriptor record in which the status of the system is saved;
  - As many core-image records as are needed to save the required parts of virtual storage;
  - One 20-byte trailer record which is identical to the header record.

  Depending on whether the file is processed forward or backward, the header or trailer record can be used to recognize and bypass checkpoint sets. The format of both header and trailer record is:

Bytes:	Contents:
00-11	///ƀCHKPTƀ// (note the two space characters: one before, and one after 'CHKPT').
12-13	The total number of records that constitute the checkpoint set, in unpacked decimal. This number includes the header and trailer records and the status descriptor record.
16-19	The serial number of the checkpoint taken.

Checkpoint sets can always be identified by the first 12 bytes of the header record or trailer record (depending on whether the file is read forward or backward).

When the file is read forward, the checkpoint header record occupies the 20 high-order bytes of the I/O area; when the file is read backward, the checkpoint trailer record occupies the 20 low-order bytes of the I/O area.

Three methods may be used to bypass checkpoint sets:

1. Go into a read loop, until a checkpoint trailer (reading forward) or header (reading backward) is encountered.
2. Extract the count from bytes 12-13 in header or trailer record (depending on whether reading forward or backward), add 2 to it, and space forward or backward that number of records.
3. Extract bytes 14-15, pack and convert the contents to binary, and space forward or backward that number of records.

In methods 2 and 3, read commands could also be used. When bypassing checkpoint sets on 7-track tapes in translate mode, only method 3 can be used. Read commands cannot be used then because they would create data checks.

Section 4: ACCESS METHODS

This section provides full information about the access methods: SAM, DAM, ISAM, and VSAM. Each access method is described in a separate chapter.

First, general information is given about the features of the access method; this includes information about the record formats and block structures supported and the devices that can be used. Special considerations on data organization are also included.

Each chapter is completed with considerations on the usage, through assembler language, of the access method. The I/O macros are explained in a context of programming procedures, showing the assembler programmer how to control the access method. As such this section acts as an introduction to the publication *DOS/VS Supervisor and I/O Macros*, GC33-5373, which deals with specific formats of the macros.

This section is mainly addressed to programmers using the assembler programming language; it may be of interest to others, though they may find the assembler-oriented approach difficult to understand.

Sequential Access Method

The input/output routines of the Sequential Access Method permit the programmer to store and retrieve the logical records of a file sequentially, without the need for coding blocking/deblocking routines. The programmer can, therefore, concentrate all his efforts on processing the data.

Another major feature of this access method is its ability of using one or two I/O areas, and of processing the data either in a work area or in the I/O area. This permits the processing of a record while data is being transferred to and from I/O areas. This section discusses these factors in terms of achieving a maximum overlap of processing with I/O.

Storage Areas and Effective I/O Overlap

Routines are designed to provide for overlapping the physical transfer of data with the processing of the data. Figure 4.1 shows the difference between processing with and without overlap. The top diagram shows that no overlap is taking place; for each I/O request, a certain amount of time elapses before the data has been transferred and the problem program can resume processing. The bottom diagram shows that the transfer of data is overlapped with the processing of other data.

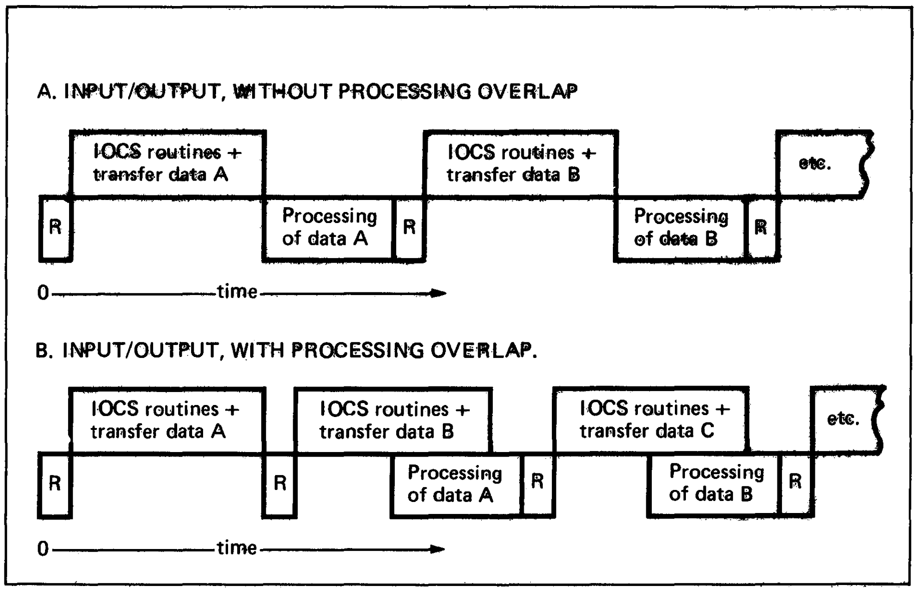

A. INPUT/OUTPUT, WITHOUT PROCESSING OVERLAP
IOCS routines + transfer data A
IOCS routines + transfer data B
etc.
R
Processing of data A
R
Processing of data B
R
0 time
B. INPUT/OUTPUT, WITH PROCESSING OVERLAP.
IOCS routines + transfer data A
IOCS routines + transfer data B
IOCS routines + transfer data C
etc.
R
R
Processing of data A
R
Processing of data B
R
0 time

Figure 4.1. Fundamental concept of overlapping I/O with processing
With overlap, data A is processed during the physical transfer of data B. R represents an I/O request in the user's program.

The amount of overlap actually achieved (effective overlap) is governed by the problem program through the assignment of work areas and I/O areas. An I/O area is that area of virtual storage to or from which a block of data is physically transferred by the channel scheduler and the physical IOCS routines. A work area is an area in virtual storage used for processing an individual logical record from a block of data. However, logical records may also be processed without using a work area; they are then processed directly in the I/O area.

Certain combinations of I/O areas and work areas are possible:

- One I/O area with *no* work area
- One I/O area *with* a work area
- Two I/O areas with *no* work area
- Two I/O areas *with* a work area.

For spanned record processing, the only allowable combinations of I/O areas and work area are:

- One I/O area *with* a work area
- Two I/O areas *with* a work area.

The occurrence of normal overlap in spanned record processing is difficult to predict because, in most cases, the system requires multiple I/O operations to satisfy one I/O request (very long logical records).

Using two I/O areas and/or a work area usually increases the processing speed. In some cases, however, a larger blocking factor may improve processing speed more than the use of either two I/O areas or a work area. Moreover, certain devices are buffered, increasing the potential amount of overlap. The following text discusses combinations of I/O areas, and work areas, as they may be used for buffered devices, unbuffered devices, blocked tape, chained diskette records, and blocked DASD. The Figures presented reflect the general principle of overlapped processing and are not intended to indicate the exact amount of overlap possible with any specific I/O device, or with any specific IBM-supplied program.

Record Format (Blocked or Unblocked)	Number of I/O Areas	Separate Work Area	Amount of Effective Overlap
Unblocked	1	no	Overlap of the device operation only for buffered devices such as 1403, 1443, 2540. No overlap of magnetic tape, 1017,1018,1442,2311,2314, 3330,3333,3340,2671,1287.
		yes	Overlap processing of each record.
	2	no	Overlap processing of each record.
		yes	Overlap processing of each record. (No advantage to a work area.)
Blocked	1	no	No overlap.
		yes	Overlap processing of full block.
	2	no	Overlap processing of full block.
		yes	Overlap processing of full block. (No advantage to a work area.)

Note: Overlap given is the maximum achievable.

Figure 4.2. Summary of achievable overlap of processing and Input/Output
Overlapping of processing and I/O may, or may not be achieved, depending on the device used, or the number of I/O areas applied.

Figure 4.2 summarizes all possible combinations of I/O areas and work area; Figure 4.4 indicates for each combination the amount of processing overlap that may be achieved. It should be noted that a certain combination may or may not be implied in specific high-level programming language. The assembler language programmer has direct control of all possible combinations when using the GET/PUT macro instructions. Whenever a programming language allows a programmer to choose a proper combination, this choice is normally made when the files are defined in the program.

In some parts of figure 4.4, the action 'P' is shown. This indicates an IOCS action when a pointer to a next logical record is updated in LIOCS. This may be a switch from one I/O area to another, or a setting of the pointer to the next logical record in a block, or both.

Note: The terms 'READ REQUEST' and 'WRITE REQUEST' as used in the following figures, do not refer to any type of I/O instruction in any programming language. They indicate the point in a user's program at which the user wants to have a next logical record made available for processing (READ) or for output (WRITE).

Throughout this manual the concept of a block of data is used; a block being a physical entity which is read or written as a *unit* on the external device. To take full advantage of information contained in this manual and apply it correctly to diskette files, one must know the following. The user can take advantage of I/O area and work area overlap processing by allowing LIOCS to read and write multiple records each time the diskette is accessed. Physically each logical data record that the user reads or writes is a separate record on the diskette. But by allowing LIOCS to *chain* I/O operations to the device on input, the user-provided I/O area will be filled each time the device is accessed. In turn, on output LIOCS waits until the I/O area provided is full before writing individual logical records on the diskette.

So, for diskette files the user can logically "block" records in the I/O areas provided by chaining I/O operations; however, each record on the diskette remains a physically separate entity.

Example:
If the user:
- has two I/O areas, each 160 bytes in length;
- wants to process 80 byte records;
- indicates chaining of 2 records.

For input, when a record is requested the following data transfer occurs:

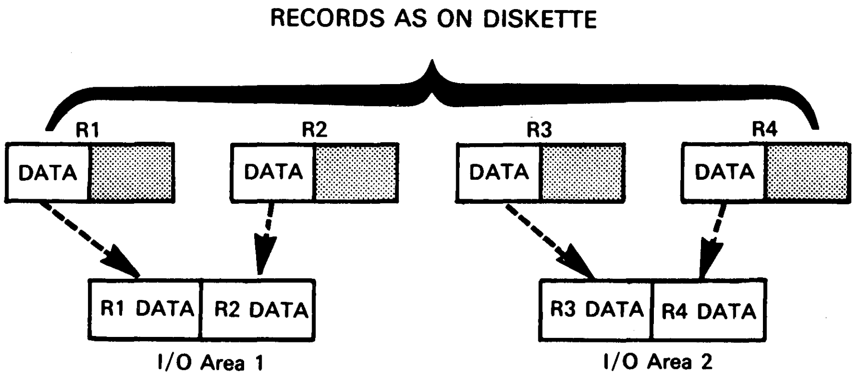


Figure 4.3A Diskette data transfer (input)
Physically the diskette records are always 128 bytes in length. Because only 80 bytes are desired, only the first 80 bytes of each physical record are placed in the I/O areas.

Diskette Data Transfer

Physically the diskette records are a constant 128 bytes in length. Because only 80 bytes are desired, only the first 80 bytes of each physical record are placed in the I/O areas. For output, when an I/O area is full, it is written on the diskette as follows:

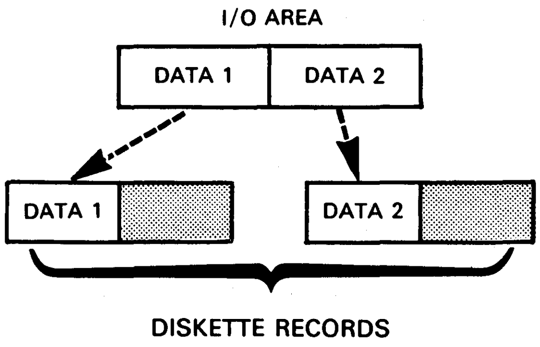


Figure 4.3B Diskette data transfer (output)
The 80 bytes comprising each data record are written in the first 80 bytes of the physical diskette record. The device itself places binary zeros in the remaining unused bytes of the physical record (shaded area).

Buffered I/O Devices

One I/O area and no work area (Figure 4.4A).

The maximum achievable overlap is the device time only. The transfer time between I/O area and buffer is not overlapped by processing.

If a next read or write request is issued before device end, the data transfer between I/O area and buffer does not take place until device end is reached.

One I/O area and one work area (Figure 4.4B).

In this combination, the maximum achievable overlap is the device time plus the transfer time between I/O area and buffer.

If a next read or write request is issued after channel end but before device end, the transfer of data between I/O area and buffer can take place, even though the transfer between buffer and device cannot start until device end is reached.

Two I/O areas and no work area (Figure 4.4C).

In this combination, the maximum achievable overlap is the device time plus the transfer time between one of the I/O areas and the buffer. Thus, the overap is the same as when using one I/O area and one work area. The difference is that, when using two I/O areas, the data in one area is processed while the other I/O area is used for actual input/output from or to the buffer. For each next request the system switches between the two I/O areas.

If the next read or write request is issued after channel end and before device end, only I/O area switching occurs. Control then returns to the problem program, and the device-buffer transfer is started after device end is reached.

Two I/O areas and one work area (Figure 4.4D).

As in the previous two combinations (Figures 4.4B and C), the maximum achievable overlap is the total transfer time: the device time plus the transfer time between I/O area and buffer. However, there is a disadvantage to this combination when compared with the previous two combinations, because this one requires extra virtual storage.

If the next read or write request is issued after channel end but before device end, the data transfer between the I/O area and the work area can take place. Control then returns to the problem program, even though the device is not started until device end is reached. If the next request is issued before channel end, IOCS must wait.

Unbuffered I/O Devices, Unblocked Records

One I/O area and no work area (Figure 4.4E).

In this combination, no overlap is possible at all. After each I/O request, processing must wait until all data has been transferred between device and I/O area.

One I/O area and one work area (Figure 4.4F).

The maximum achievable overlap is the transfer time between device and I/O area. When a next I/O request is issued before channel/device end is reached, IOCS must wait.

In this combination, the work area replaces the buffer of buffered devices. Therefore, the total effect is more or less the same as in Figure 4.4A.

Two I/O areas and no work area (Figure 4.4G).

As in the previous combination, the maximum achievable overlap is the transfer/time between device and I/O area. Processing is done in one I/O area while data is tranferred between the other I/O area and the device. This combination does not require a move of the data to or from a work area; instead, IOCS points alternately to the I/O area in process.

Two I/O areas and one work area (Figure 4.4H).

This combination also allows for a maximum overlap of the transfer time between device and I/O area. However, when compared with the previous two combinations, it has the disadvantage of requiring extra virtual storage.

Generally speaking, the best choice is either one I/O area with a work area, or two I/O areas and no work area. Which of these combinations is chosen depends on the application. For example, when processing an input file whose records are written to another file after some modification (updating), it may be useful to use a work area for both files. The work area may then even be the same area, allowing the system to read records into the work area, process them, and write them to the second file from the same area. In other cases the best choice is two I/O areas and no work area.

Unbuffered I/O Devices, Blocked Records

In all following combinations, there is overlap only if actual I/O is involved. The Figures present only these situations. I/O requests that do not result in actual I/O cause, in LIOCS, a pointer to the next logical record in the existing block to be updated, so that this new logical record is made available for processing. In combinations where a work area is used, such requests also result in data transfer between work area and I/O area.

One I/O area and no work area (Figure 4.4I).

This combination has no overlap of processing with input or output. The I/O time per record depends on the blocking factor. Therefore, the I/O time per *record* can be reduced by increasing the blocking factor.

Actual I/O is performed when the first record of a new block must be obtained, or when the last record of a block is written.

One I/O area and one work area (Figure 4.4J).

The maximum achievable overlap is the time for data transfer between I/O area and device. I/O requests for any record, except the last in a block, involve only a data transfer between I/O area and work area. For the last logical record in a block, the data transfer between work area and I/O area is followed by a data transfer between I/O area and device which is overlapped by processing. Channel end must occur before the first record of the next block can be processed by LIOCS.

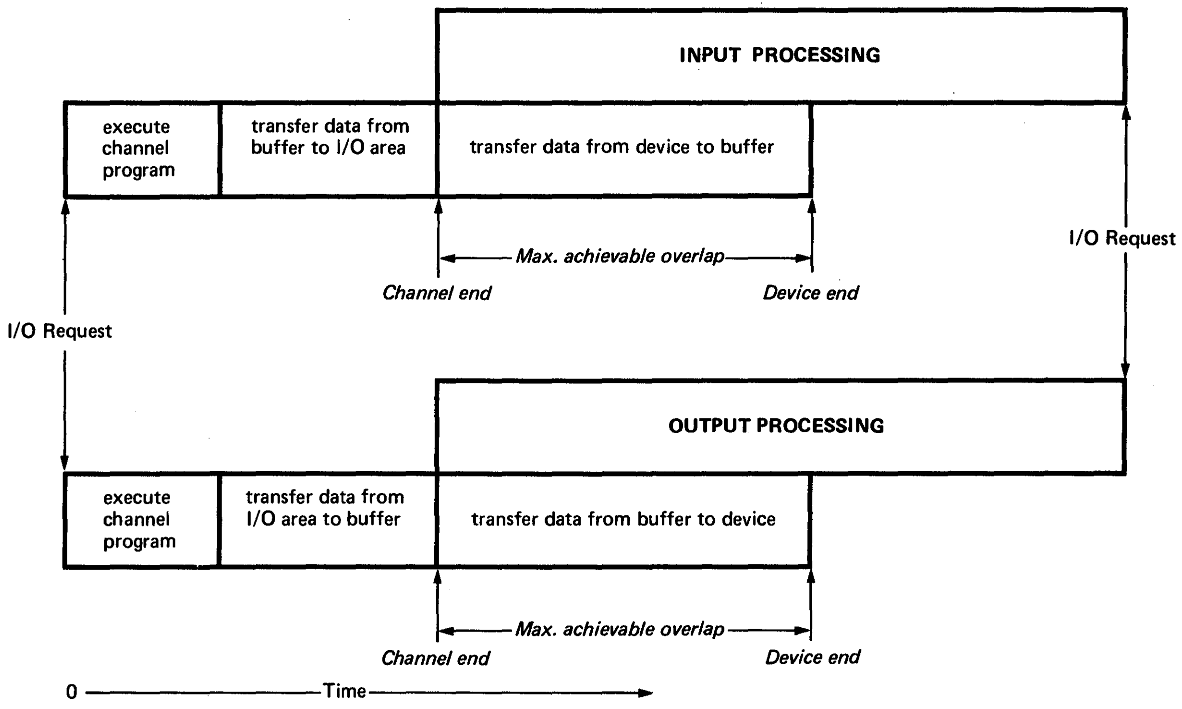
Two I/O areas and no work area (Figure 4.4K).

In this combination, the maximum achievable overlap is the time for transfer of data between I/O area and device. I/O requests for all but the first record of a block take time only for pointing to the next record. Input requests for the first logical record of a block must wait for channel end of the data transfer to the alternate area. Pointing to the first record and returning control to the program is then overlapped with the next device transfer. Output requests work the same way, except that the wait occurs with the last logical record of a block.

Two I/O areas and a work area (Figure 4.4L).

There is a disadvantage to this combination compared with the two previous ones because it requires extra virtual storage; the maximum achievable overlap is the time for transfer of data between I/O area and device.

A. BUFFERED DEVICES, One I/O area and NO work area.



B. BUFFERED DEVICES, One I/O area and one work area.

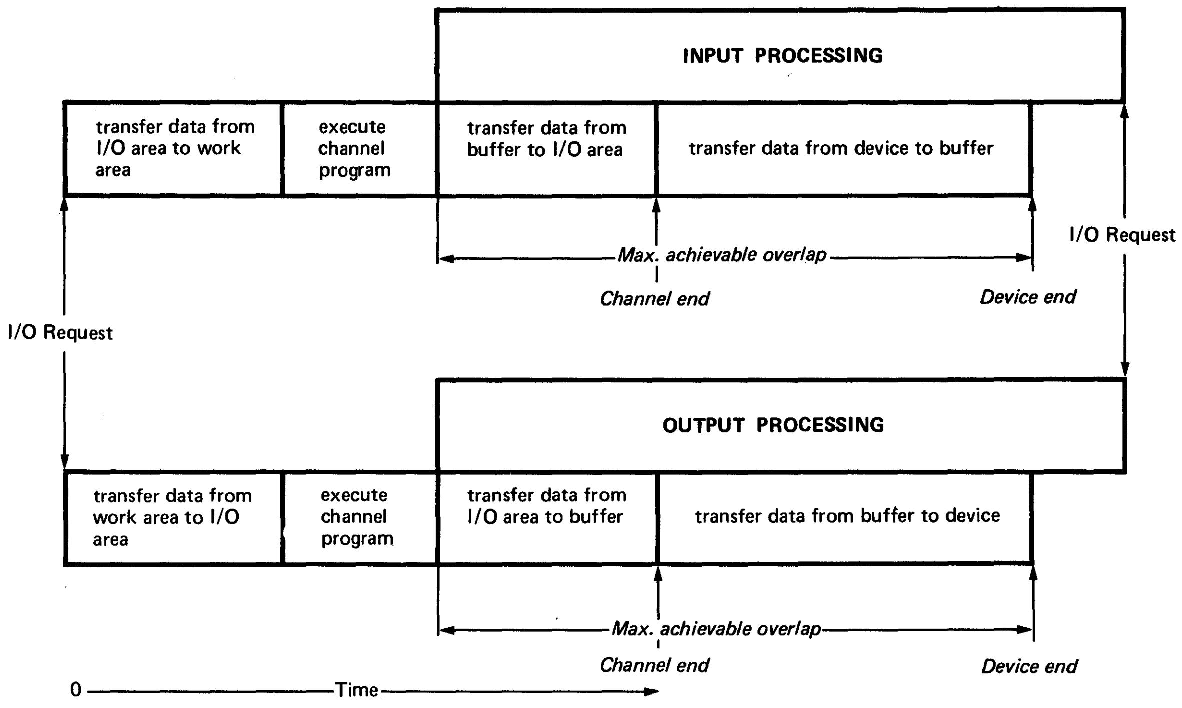
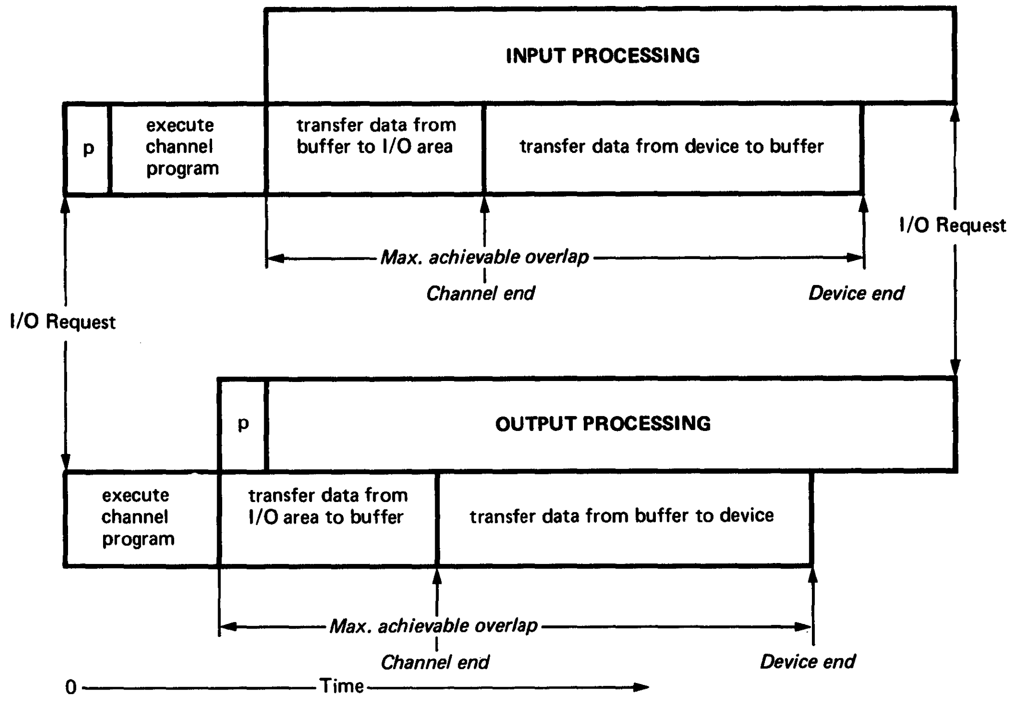

INPUT PROCESSING
transfer data from I/O area to work area
execute channel program
transfer data from buffer to I/O area
transfer data from device to buffer
Max. achievable overlap
Channel end
Device end
I/O Request
I/O Request
OUTPUT PROCESSING
transfer data from work area to I/O area
execute channel program
transfer data from I/O area to buffer
transfer data from buffer to device
Max. achievable overlap
Channel end
Device end
0
Time

Figure 4.4. Storage areas and effective overlap (1 of 6)

C. BUFFERED DEVICES. Two I/O areas and NO work area.



D. BUFFERED DEVICES. Two I/O areas and one work area.

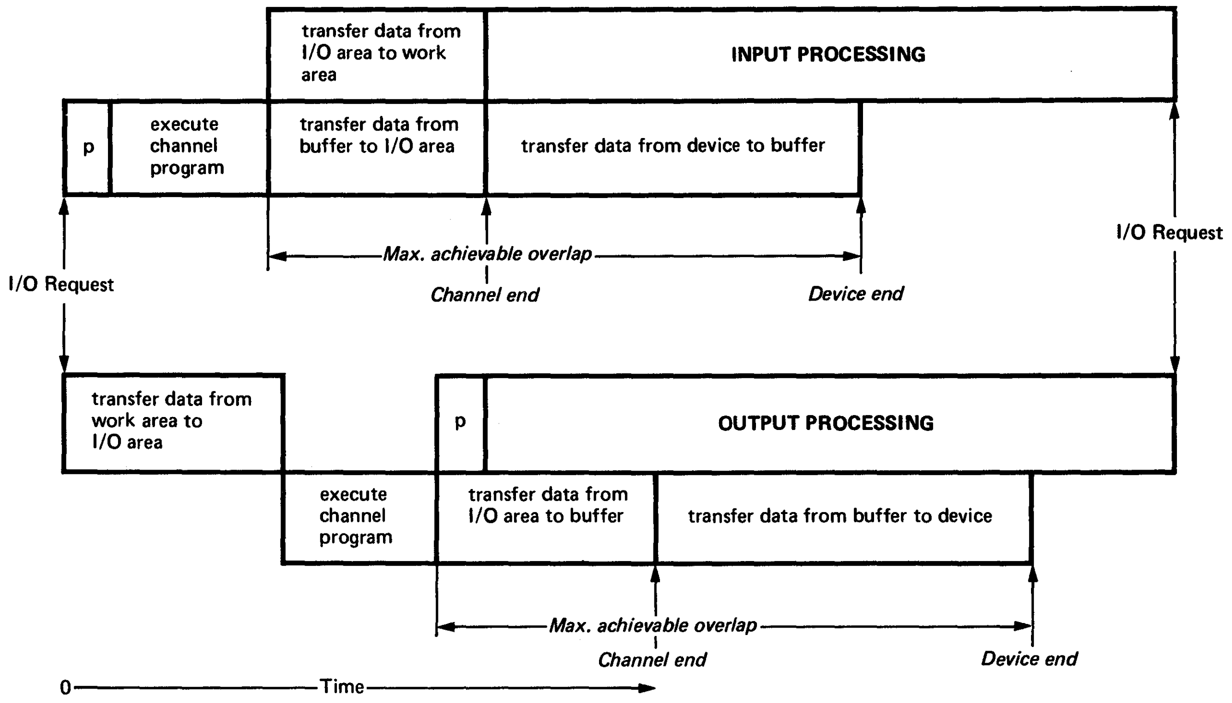
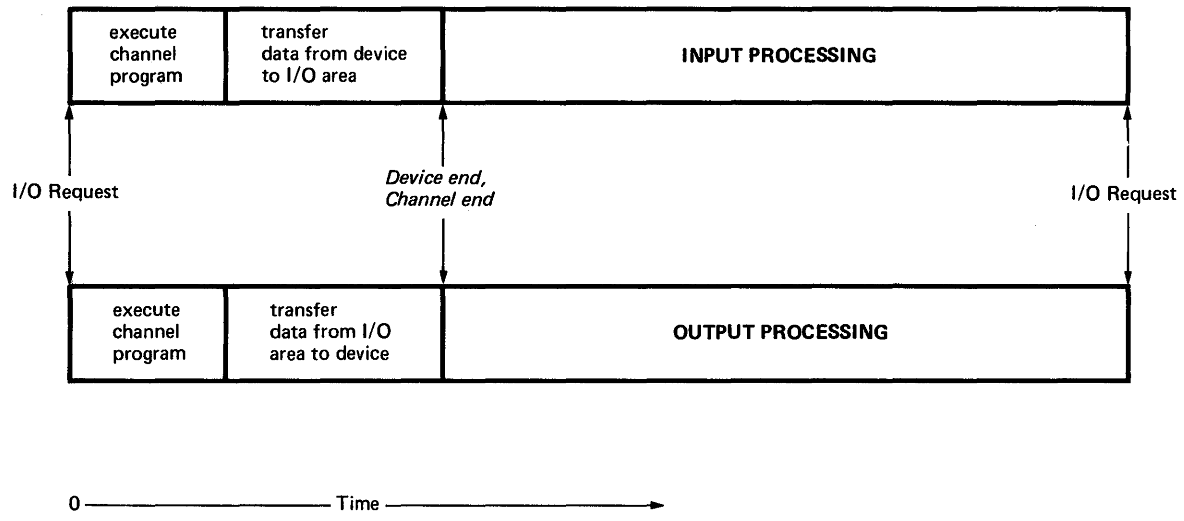

transfer data from I/O area to work area
INPUT PROCESSING
p
execute channel program
transfer data from buffer to I/O area
transfer data from device to buffer
I/O Request
Max. achievable overlap
Channel end
Device end
I/O Request
transfer data from work area to I/O area
p
OUTPUT PROCESSING
execute channel program
transfer data from I/O area to buffer
transfer data from buffer to device
Max. achievable overlap
Channel end
Device end
0
Time

Figure 4.4. Storage areas and effective overlap (2 of 6)

E. UNBLOCKED RECORDS. One I/O area and NO work area.



F. UNBLOCKED RECORDS. One I/O area and one work area.

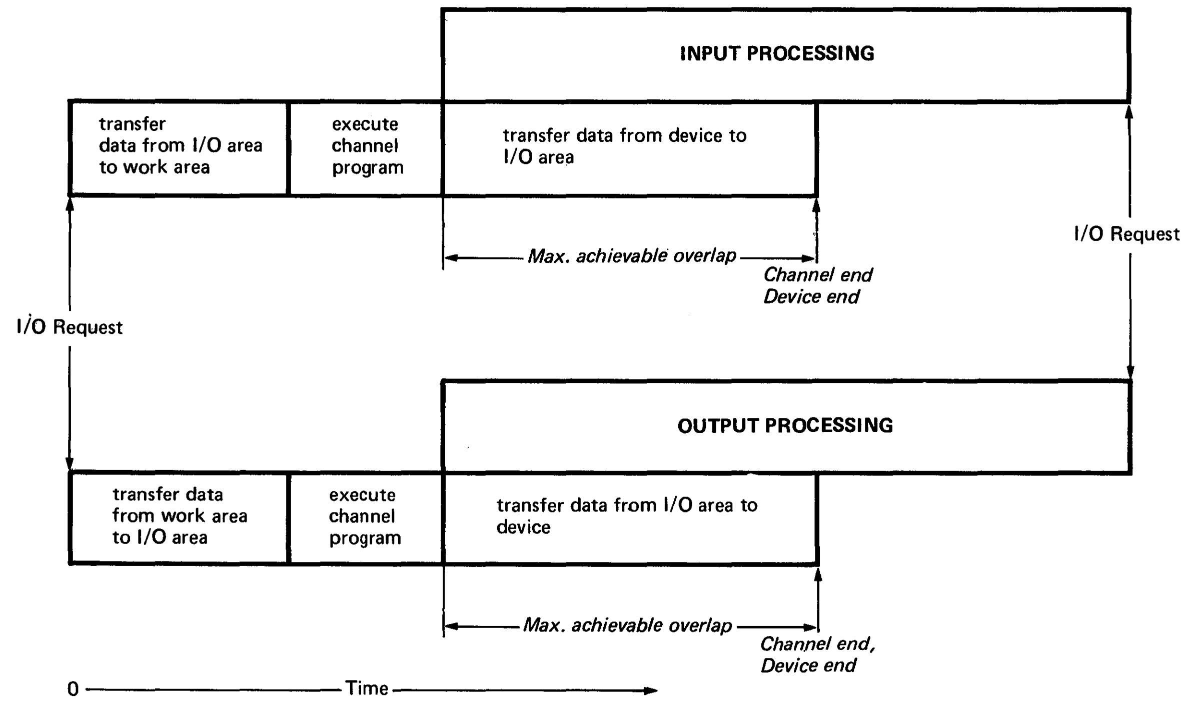
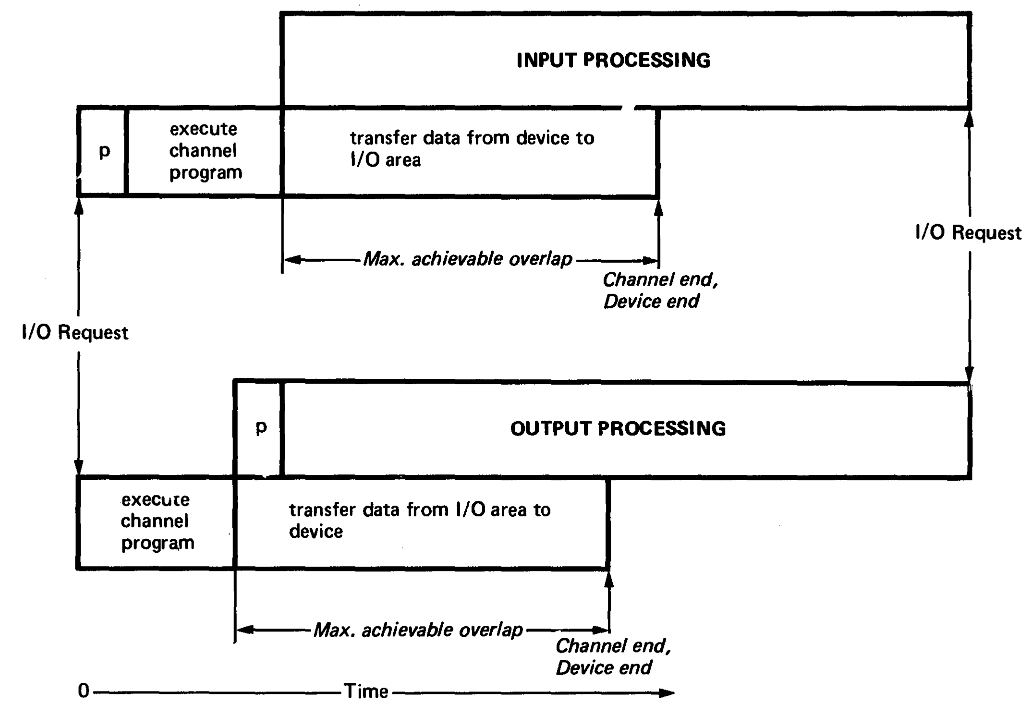


Figure 4.4. Storage areas and effective overlap (3 of 6)

G. UNBLOCKED RECORDS. Two I/O areas and NO work area.



H. UNBLOCKED RECORDS. Two I/O areas and one work area.

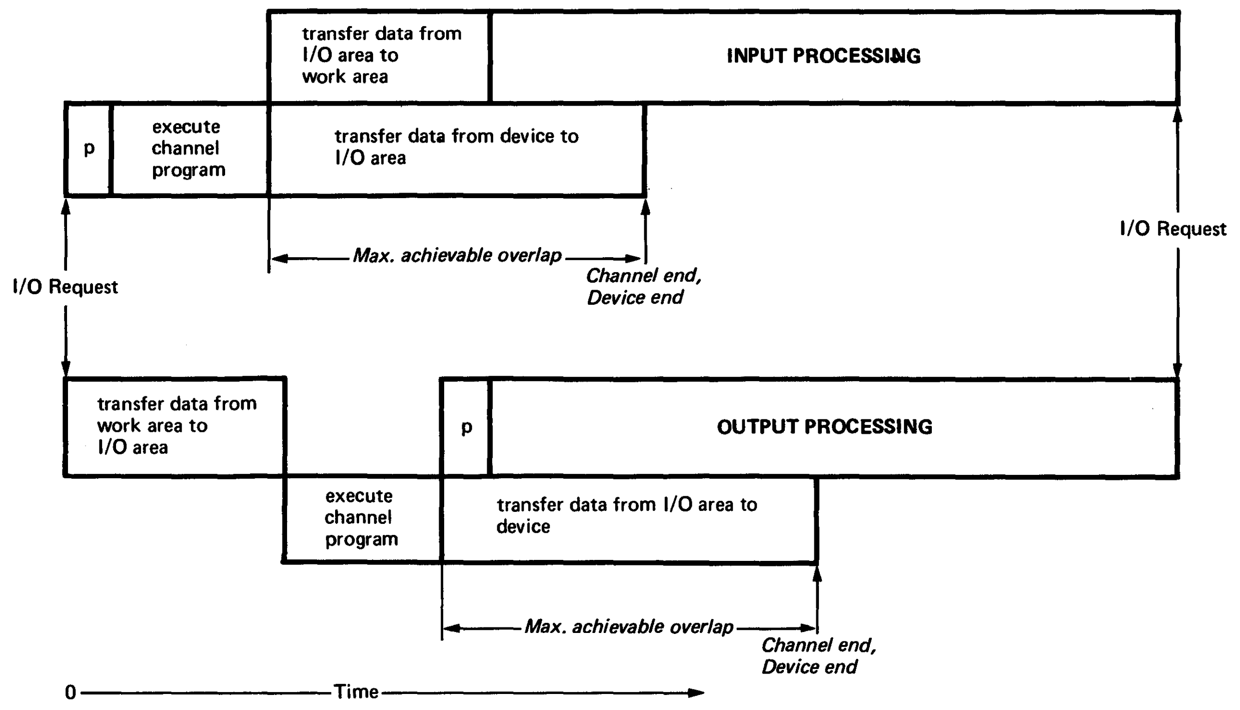
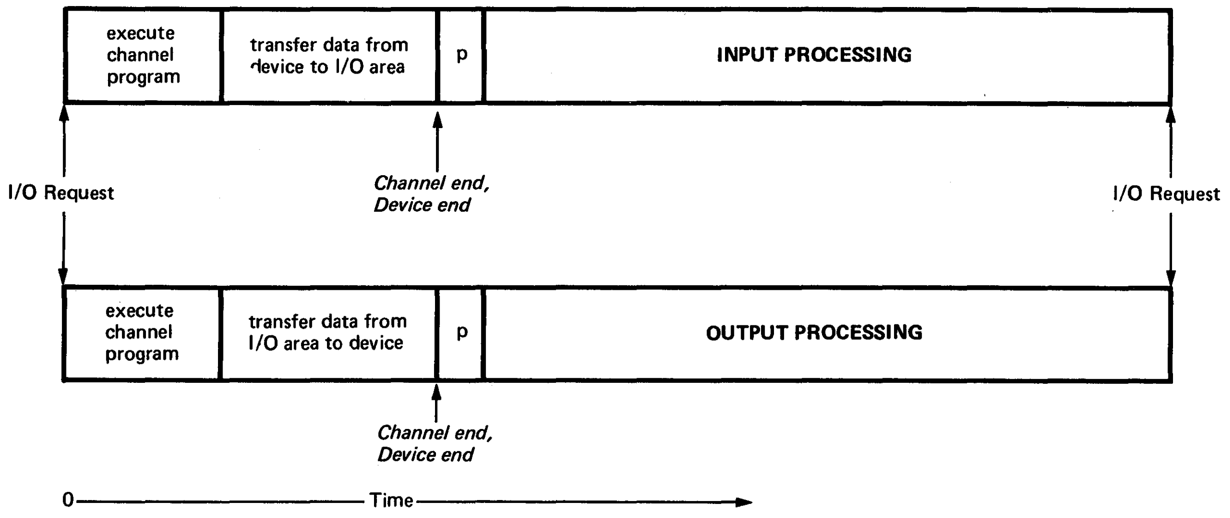


Figure 4.4. Storage areas and effective overlap (4 of 6)

I. BLOCKED RECORDS. One I/O area and NO work area.



J. BLOCKED RECORDS. One I/O area and one work area.

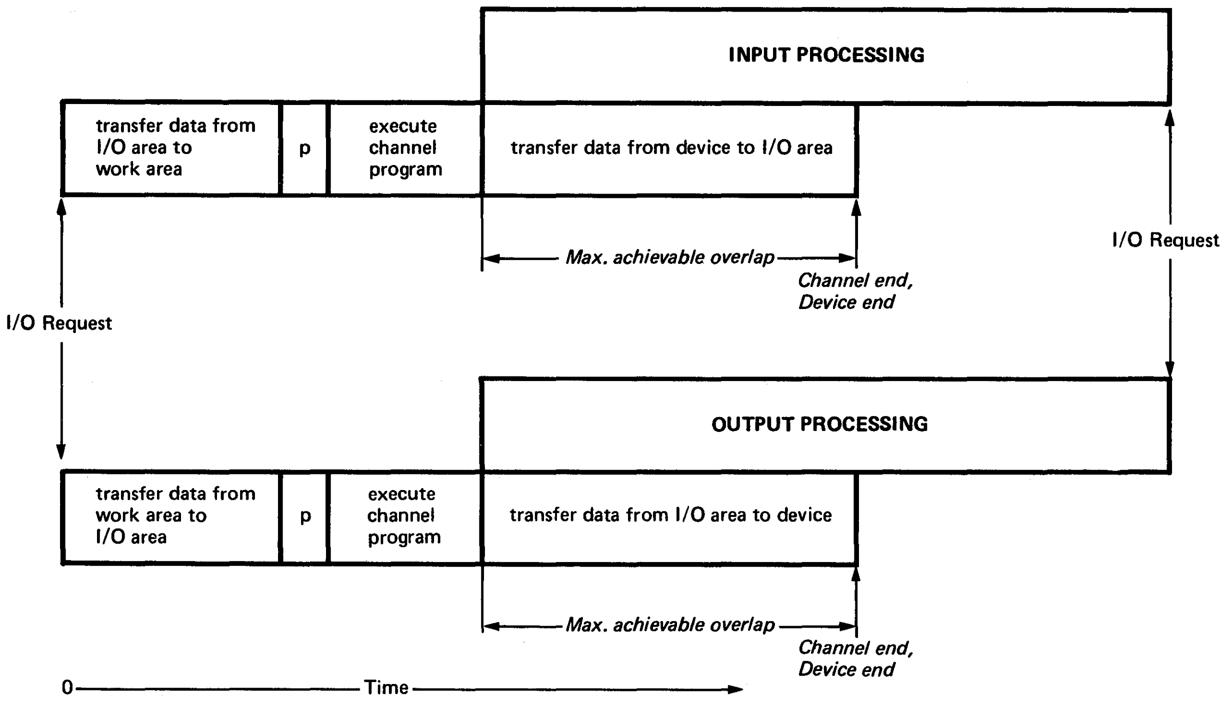
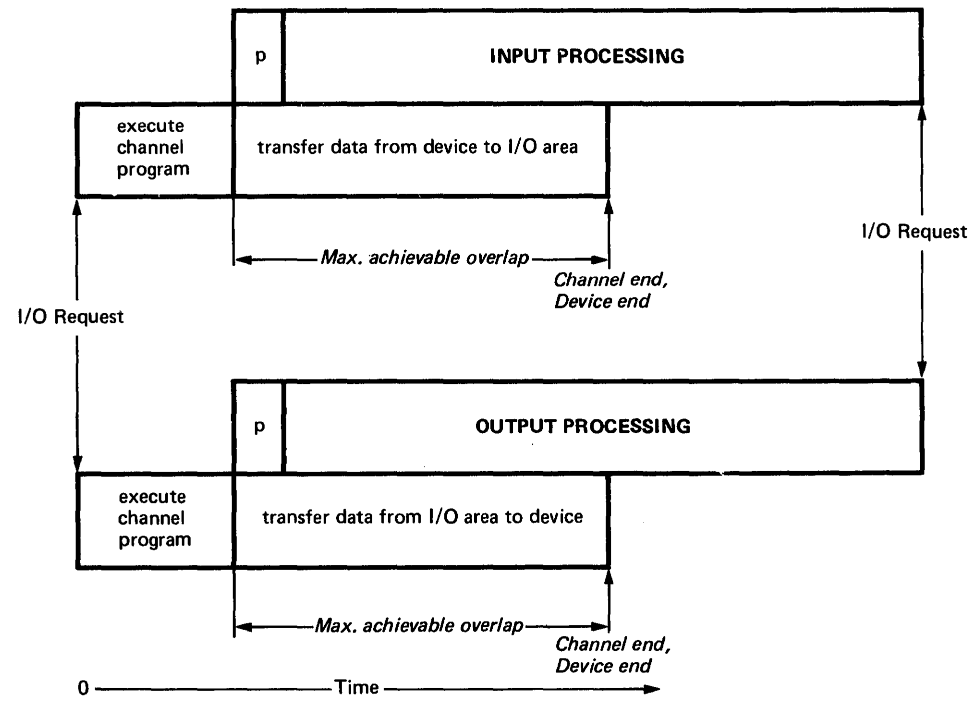


Figure 4.4. Storage areas and effective overlap (5 of 6)

K. BLOCKED RECORDS. Two I/O areas and NO work area.



L. BLOCKED RECORDS. Two I/O areas and one work area.

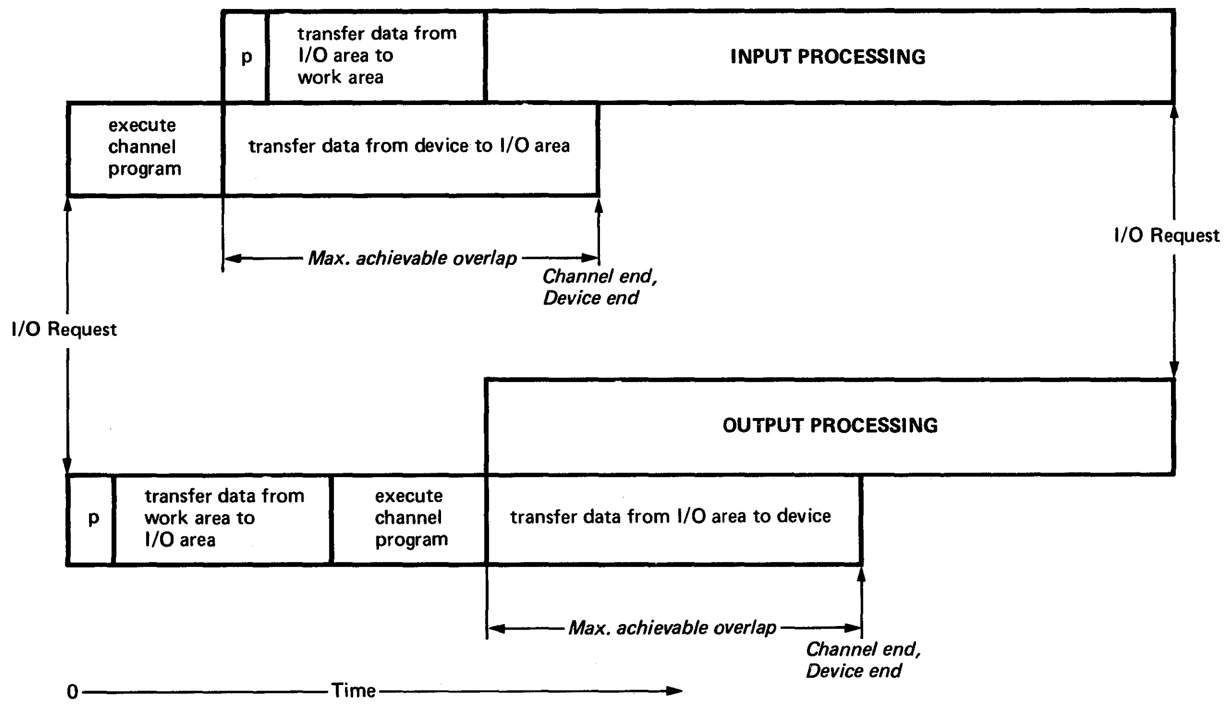


Figure 4.4. Storage areas and effective overlap (6 of 6)

Assembler Language Considerations

The macro instructions provided for the sequential access method allow the programmer to process a file sequentially with a minimum of effort.

For input/output, two levels of processing are available:

- GET/PUT level sequential access
- READ/WRITE level sequential access.

The GET/PUT level macro instructions permit the programmer to store and retrieve logical records of a file without the need for coding blocking/deblocking routines: these functions are performed automatically by LIOCS whenever necessary. For example, each time a programmer issues a GET macro instruction in his program, a next logical record is made available for processing. Actual input/output is performed only when a next logical record must be obtained from a next block.

Another major feature of this level of sequential processing is the choice of using either one or two I/O areas, and of processing a logical record either in one of the I/O areas or in a work area. This is discussed and illustrated under "Storage areas and Effective I/O overlap".

The READ/WRITE level macro instructions provide the programmer with an efficient and flexible means for storing and retrieving the blocks of a sequentially organized magnetic tape or DASD file. The macro instructions provided with this level of the Sequential Access Method allow a file to be treated alternately as input or output. It is particularly effective in applications where blocks are alternately read from and written to a file used as a temporary extension of virtual storage.

Blocking and deblocking is *not* included in this level of I/O processing: the programmer is responsible for writing his own routines if blocking/deblocking is desired. The READ and WRITE macro instructions operate on the same basis as the physical IOCS EXCP macro instruction, except that the CCW chain is provided by the system.

To a certain extent, even random processing is possible with this level of I/O processing, by means of the POINT macro instruction.

Logical Record Processing

GET	The GET macro instruction is used to obtain logical records in physical sequential order from a file on any device. Automatic record deblocking is included. As required, the system schedules the filling of input areas, deblocks records, and directs error recovery procedures. The system also checks for end-of-volume condition, and initiates automatic volume switching if an input file extends over more than one volume. When a file occupies more than one discontinuous area on a DASD volume, automatic switching from one extent to the next is also performed.
PUT	The PUT macro instruction releases logical records to the system for output, in physical sequential order. Automatic record blocking is included.

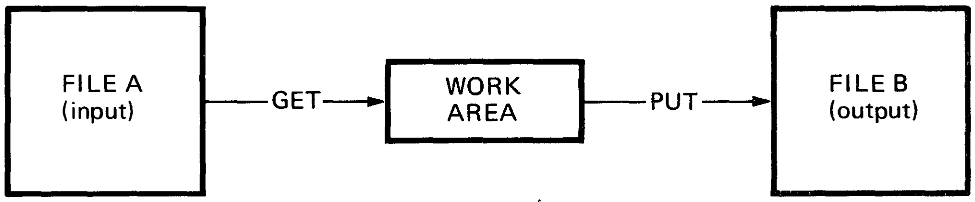
As required, the system blocks records, schedules the emptying of output areas, and handles output error correction procedures, where possible. The system checks for end-of-volume condition and performs automatic volume switching and label creation. References to non-continuous DASD extents are resolved.

Both the GET and PUT macro instructions can be used in either of two ways:

- In *move mode* where logical reocrds are moved from an input area to a work area (GET), or from a work area to an output area (PUT).
- In *locate mode* where logical records are processed in the I/O area. A register then contains the address of the first byte of the logical record in the I/O area.

Move Mode GET and PUT Macro Instructions
When operating in the move mode, the GET and PUT macro instructions transfer logical records between the I/O area and a work area.

For an input file whose logical records are to be updated and written to another output file, a work area is useful. Both the input file and the output file may share the same work area. The size of a work area of an input file may be larger than the input records, so that the records may be extended during processing.

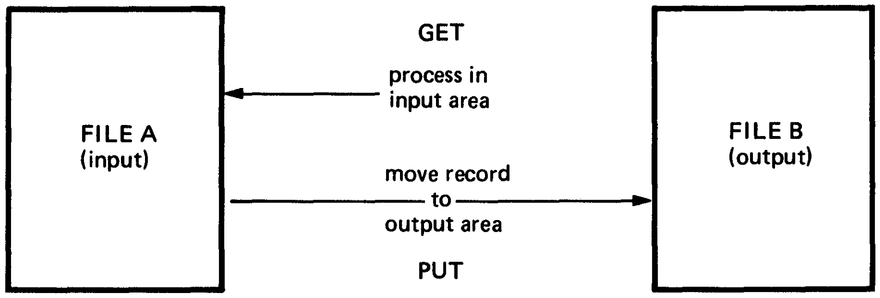


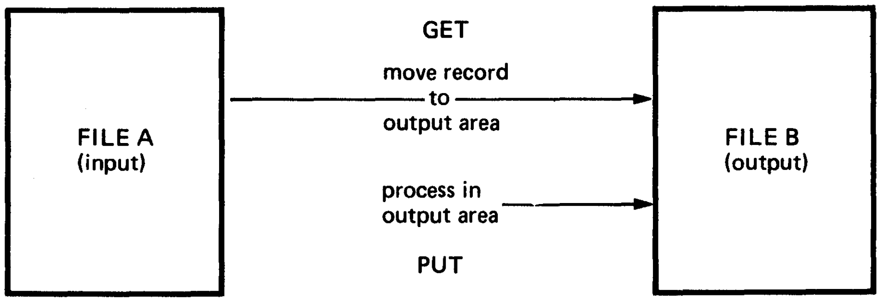
If the move mode is used for the GET and PUT macro instructions, the following operands need to be specified:

- The name of the file.
- The address of the programmer's work area, or a register which contains this address. The register option should be used for self-relocating programs.

Locate Mode GET and PUT Macro Instructions
When operating in the locate mode, logical records are processed in the I/O area. After each GET, a register points to an input area (segment) where the logical record has been made available for processing. Here the programmer is not able to extend the size of the logical record during processing. On the other hand, there is no need for an additional transfer of data to a work area. After each PUT, a register points to the output area (segment) where the next following logical record may be built.

When records are to be updated and written to a separate output file, the programmer may either process the record in the input area and move it to the output area afterwards, or move the record from the input area to the output area and process the record there.





If the locate mode is used for the GET and PUT macro instructions, the only operand required is the name of the file.

Processing Blocked Logical Records
In normal cases, a program will process a file, starting with the first logical record, until end-of-file is signalled by IOCS. In these cases there is no difference between the processing of unblocked records and the processing of blocked records, since the blocking and deblocking of logical records is performed automatically with the GET and PUT macro instructions.

A special type of processing, however, may require the system to obtain a next logical record from the next block, ignoring any remaining logical records that follow the one in the same block that is being processed. Or, for output, it may require the system to place the next logical record as the first record of a new block. A special case is spanned record processing, where a program may wish to obtain a next record and bypass all segments of the current record being processed.

RELSE	The RELSE macro instruction causes the following GET macro instruction to obtain the first logical record from the next following block and to ignore any logical record remaining in the current block. When spanned records are processed, the RELSE macro instruction causes the following GET to skip to the next first segment of the next logical record.
TRUNC	The TRUNC macro instruction causes the following PUT macro instruction to regard an output area as full, and subsequently to place the next logical record in the next block. Thus, just as intput areas may be released by the RELSE macro instruction, output areas may be

	truncated for writing short blocks. IOCS provides for reading truncated blocks, so that a short block will not necessarily result in an error condition on input. The CLOSE macro instruction effectively truncates the last block of a file.

For both the RELSE and TRUNC macro instructions, the name of the file must be specified as an operand.

End-of-Volume Conditions
Both the GET and PUT macro instructions check for end-of-volume condition. If such a condition occurs, the system performs automatic volume switching.

The programmer may decide to stop processing a file on one volume, and to resume processing the same file on the next volume.

FEOV	The FEOV macro instruction forces the system to assume an end-of-volume condition on either an input or output *magnetic tape* file, thereby causing automatic volume switching. When FEOV is issued on an input file, trailer labels are not checked. The header labels of the next volume, however, are verified. When FEOV is issued on an output file, trailer labels are created as required.
FEOVD	The FEOVD macro instruction forces the system to assume an end-of-volume condition on either an input or output *DASD* file, thereby causing automatic volume switching. The operation is exactly the same as for the FEOV macro instruction except that trailer labels are processed also for input, after a FEOVD is issued.

For both the FEOV and FEOVD macro instructions, the name of the file is required as an operand.

Updating
Files that are processed sequentially, are normally either read or written. For certain devices, however, it is also possible to obtain logical records and after processing, to write an updated version of the records back into their original location in the file.
Those devices are:

- All types of DASD.
- Card input on a 1442, 2520, 2560, 3525, or 5425.
- Card output in the punch feed of a 2540 equipped with the punch-feed-read special feature.

For card files on the 1442, or 2520, the user specifies TYPEFLE=COMBND in the DTFCD macro if card records must be updated.

For card files on the 2560, 3525, or 5425, the user specifies the operands ASOCFLE and FUNC if card records must be updated.

For DASD files, the user specifies UPDATE=YES in the DTFSD macro if DASD records are updated.

Records are obtained from the file as usual by a GET macro instruction. After the record has been processed, the next PUT causes the record to be returned to its original location in the file. (DASD), or punched into the same card from which it was read.

Processing is done in the input area. After processing, the records are returned to the file from the *input* area. For card devices, the records are returned to the file by PUT, for DASD the PUT sets an indicator which is used by the next GET to accomplish the transfer. Between a PUT and the next GET, the input area must not be modified.

If a work area is used for the file, the records are returned, by PUT, from the work area to the input area and then from the input area to the file.

If a particular record does not require updating, a subsequent PUT may be omitted, except for the 2540, 2560, 3525, or 5425.

End-of-File Conditions
For end-of-file conditions, see the end of this section on Assembler Language Considerations.

Special Macro Instructions for OCR and MICR

RDLNE	The RDLNE macro instruction provides selective online correction when processing journal tapes on the IBM 1287 Optical Character Reader. The macro instruction causes the reader to read a line in the online correction mode, while processing in the offline correction mode.
DISEN	The DISEN macro instruction cuases the magnetic character reader or the optical reader/sorter to stop feeding documents.
LITE	The LITE macro instruction permits any combination of pocket lights on the magnetic character reader or the optical reader/sorter to be lit after a specified number of documents has entered the pockets.

Physical Block Processing

READ	The READ macro instruction requests that a *block* be transmitted from a file to a virtual storage area. Any deblocking must be performed by the problem program, and each READ operation transfers one full block into virtual storage. To allow overlap of the input operation with processing, READ does not wait for the end of the operation, but returns control to the problem program as soon as the READ has initiated the CCW chain that performs the data transfer.

Therefore the programmer must check for completion of the input operation before processing the data. After READ has retrieved all blocks of a file, and discovers that no more data is available for processing, IOCS passes control to the user's end-of-file routine, whose address is specified in the DTFxx macro instruction.

Because the READ macro instruction has been designed for workfiles, multiple volume support is not available. A file to be processed by means of the READ macro instruction must be contained in one volume.
READ can also be used to read backwards from magnetic tape.

When used for MICR, the READ macro instruction permits the use of more than one magnetic reader per program.

WRITE

The WRITE macro instruction requests that a block of data be transferred from virtual storage to a file. WRITE operates in much the same fashion as READ, except that WRITE is in reverse. The file to be processed by WRITE must be contained in one volume.

CHECK

The CHECK macro instruction waits for the completion of an I/O operation requested by a READ or WRITE macro instruction. It also tests for errors and exceptional conditions that may have occurred during the data transfer. As required, control is passed to the appropriate exits, for error analysis and end-of-file, that are specified by the programmer in the DTFxx macro instruction for the file. After having issued a READ or WRITE, the programmer must use the CHECK macro instruction before issuing any other macro instruction for the same file, or before the contents of the input or output area in virtual storage is altered.

Both READ and WRITE operate in a strictly sequential manner, starting either at the beginning of a file, or at a point to which the file has been positioned by one of the three POINTx macro instructions (see below).

NOTE

The NOTE macro instruction places into a register the position on a volume of the last block that has been transferred to or from virtual storage by means of a READ or WRITE macro instruction. This data can be saved by the user's program and subsequently be used to reposition the file to this location (see the POINTR and POINTW macro instructions, below).

For *output* files that are written on a DASD volume, the NOTE macro instruction also places in another register the number of bytes of space remaining on the track containing the noted (= last written) block.

Before a programmer issues a NOTE macro instruction, the last I/O operation must be tested for completion by means of the CHECK macro instruction.

POINTS

This macro instruction causes the file to be positioned at the beginning. For magnetic tape files, POINTS causes a rewind of the tape to load point and the positioning the tape to the first data block; labels are bypassed to the first tape mark. A DASD file is positioned to the lower limits of the first extent of the file.

POINTR

The POINTR macro instruction is used to position the file to a specific block, prior to the reading of that block by means of a subsequent READ macro instruction.

A series of READ macro instructions following a POINTR will pick up blocks sequentially starting with the block specified in the POINTR.

The address to be specified in the POINTR macro instruction can be obtained from the result of a previously issued NOTE macro instruction (see above) or may be specified by the user himself. More specific information about the use of the POINTR macro instruction in combination with the NOTE, READ, and WRITE macro instructions, is presented in *DOS/VS Supervisor and I/O Macros,* GC33-5373.

POINTW

The POINTW macro instruction is used to position the file to a block following the one specified in the POINTW, prior to writing a block to that location with a subsequent WRITE macro instruction.
A series of WRITE macro instructions following a POINTW will write blocks sequentially, starting at a location following the block specified in the POINTW.

The address to be specified can be obtained from the result of a previously issued NOTE macro instruction (see above) or may be specified by the user himself. More specific information about the use of the POINTW macro instruction in combination with the NOTE, READ, and WRITE macro instructions, is presented in *DOS/VS Supervisor and I/O Macros,* GC33-5373.

The reader may have understood from the preceding text that the Sequential Access Method allows for random processing through the READ, WRITE, NOTE, and POINTx macro instructions. To a certain extent this is true. However, the following facts should be recognized:

1. The READ and WRITE macro instructions can be applied in SAM, only to files that are completely contained on one single volume.
2. Random processing on magnetic tape, although possible, is highly inefficient, since many blocks may have to be bypassed when going from one block to another. Furthermore, on magnetic tape, random processing is restricted to reading; writing should be done sequentially. This is because writing is performed within certain tolerances: it takes some time before actual writing starts and when actual writing stops it takes some time before the medium comes to a complete standstill. The interrecord gaps between blocks allow for this, but frequent overwriting of blocks may cause an interrecord gap to be too short or too long, and may even affect a following block.
3. For random processing on DASD through SAM, the user may specify the operand UPDATE=YES in the DTFSD macro instruction. In this case, a WRITE will be considered as a WRITE UPDATE; in other cases, it is considered as a WRITE SEQUENTIAL.
   For a WRITE UPDATE, the block that was POINTED at will be overwritten; for a WRITE SEQUENTIAL, the block following the one that was pointed at will be written and the remained of the track is erased.

Storage Areas
Files that are processed by means of the READ and WRITE macro instructions, use one single I/O area with a size equal to the block length. This area is filled or emptied each time a READ or WRITE macro instruction is issued.

The I/O area need not be fixed in location: the programmer supplies the address of the I/O area in the macro instruction itself, each time the macro instruction is issued.

Special Macro Instructions for OCR and MICR

DSPLY	The DSPLY macro instruction displays a document field on the display screen. This macro keys-in a complete field on the keyboard when a 1287 read error makes this type of correction necessary.
RESCN	The RESCN macro instruction selectively rereads a field on a document when a 1287 read error makes this type of correction necessary.
WAITF	The WAITF macro instruction, for use in document processing with the IBM 1287 optical reader, is issued to ensure that the transfer of data from reader to CPU has been compoleted without error. The WAITF macro instruction is used with magnetic character readers in a multiprogramming system to determine if any magnetic character reader has documents ready for processing. The WAITF macro instruction is used with optical reader/sorters in a multiprogramming system to determine if any optical reader/sorter has documents ready for processing.

Device Control Macro Instructions

PRTOV

The PRTOV macro tests overflow indicators for *online* printer channel overflow; it can be used only of the DTFPR macro is used, not if the DTFDI macro is used. If an overflow indicator is ON, the user's printer overflow routine, if specified, gets control. When no user's printer overflow routine is specified in the DTFPR macro, PRTOV causes an automatic skip to channel 1.
The overflow indicator to be tested (channel 9 to 12) is specified as an operand of the PRTOV macro.

Note: At system generation, a page size is inserted. The system maintains a line count and automatically skips to channel 1 if this line count becomes higher than the constant inserted at sysgen. The constant can be changed for a job at job control time, using the SETLINE job control statement.

CNTRL

The CNTRL macro provides for the following functions:

- Card readers and punches (DTFCD only):
  Stacker select control.

- Printers (DTFPR only):
  Space number of lines, before or after print.
  Skip to specified channel, before or after print.

- Magnetic tape units:
  Rewind.
  Rewind and unload.
  Erase gap (write blank tape).
  Backspace to interrecord gap or to tapemark.
  Backspace to tapemark.
  Space forward to interrecord gap.
  Space forward to tapemark.
  Space forward logical record.
  Backspace logical record.
  Logical record spacing (spanned record input only).

- DASD:
  Seek to specified track.

- Optical readers:
  Mark error lines when reading journal tapes.
  Read keyboard information when reading journal tapes.
  Eject documents.
  Stacker select documents.
  Increment documents.

The use of the PRTOV and the CNTRL macro instructions requires that the devices involved are actually online when the program is executed. In many cases, however, the output will be written to another device type than is assumed by the program. For example, print output may be written to magnetic tape when the program is executed, and a tape-to-print program may produce the printed output later. In such programs the file characteristics must be specified in the DTFDI macro, not in the DTFCD or DTFPR macro; the PRTOV and CNTRL macros cannot be applied for device control purposes if DTFDI is used.

Instead of using the CNTRL macro instruction, the user may use the optional control character in the first byte of his output records. Also, instead of using the PRTOV macro instruction, the user may maintain a line-count field in his program which is incremented by one for each line printed or spaced on a form, and is reset to zero at the first of a new form.

Using a line-count field and control characters instead of the PRTOV and CNTRL macro instructions has the following advantages:

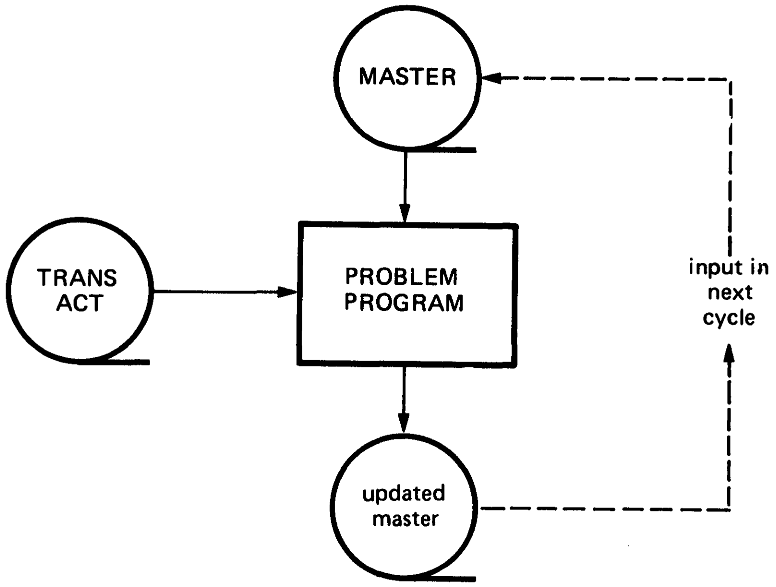
- The line-count field can be used in the program to determine the amount of lines already printed on a form.
- When all programs running on an installation use a line-count field and control characters, each program can create its own special printing lay-out, without the need for frequenly changning a carriage tape on the printer. As long as all different forms used are of equal length, one carriage tape may be designed that suits all different applications. On such a tape, channel one may be used for the top of the form, channel 9 or 12 may be used for the bottom of the form, and any other channels may be used to indicate any desired checkpoints on the form.
- As already indicated above, the use of a line-count field and control characters allows device-independent programming. Output can be written on any device available when the program is executed and can be transformed to the desired type of output at a later time. See also section 2 "Device Independence".

End-of-File Conditions
Issuing a GET or READ macro instruction on a sequential input file after the last record of that file has been processed, results in an end-of-file condition. IOCS then enters the user's end-of-file handling routine, the address of which is specified in the DTFxx macro instruction. This routine usually contains the CLOSE macro instruction to deactivate the file.

A special case is a program that processes a sequential transaction file against a sequential master file, producing an updated version of the master file. One cannot usually predict which input file will reach end-of-file first. Unexperienced programmers usually find this a problem, difficult to control. Since, however, this type of program often occurs in business data processing, this section on assembler language for SAM ends with a sample program of this type, indicating a general solution to the problem.

The sample program operates on three magnetic tape files:

1. A master file as INPUT, whose filename is MASTER. The end-of-file address specified in the DTFMT macro instruction is EOFMT. The I/O area name is MFREC.
2. A transaction file as INPUT, whose filename is TRANSACT. The end-of-file address specified in the DTFMT macro instruction for this file is EOFTA. The I/O area is TARECD.
3. An updated master file as as OUTPUT. This file will be input for the next cycle of this program, with new transaction input. For the purpose of explaining how to handle EOF conditions on input, this output file is not important, and it is not described in more detail.



Reading the input files TRANSACT and MASTER is controlled by two program switches (see Figure 4.5). If a switch is ON, reading is permitted. The switches are ON when the program begins.

As soon as a GET is issued (switch was ON) the switch for the file is set to OFF. Unless the switch is set to ON again, subsequent GETs are bypassed. The two read switches are set on by the processing routines in the program.

The identifier fields of both TARECD and MFREC are to be compared in order to find out whether both records match or not (both files are pre-sorted in ascending sequence by identifier field). The two fields are called CTA and CMF and are equal in length. They may be in the input area, or may be moved from the input area to two work fields with those names.

Depending on the result of the comparison, one processing routine out of three is selected:

1. If CTA and CMF are equal, the transaction record matches the master record so that updating can take place. After processing and writing an updated master record, new input must be obtained from both MASTER and TRANSACT. Consequently both read switches are set to ON (see *note* below).

   Note: It was assumed that only one transaction can occur on one master record. If this assumption is not true, the program must be modified in such a way that after the updating of a master record with the contents of a transaction record, only the read switch for the file TRANSACT is set to ON. As long as both files are in the proper sequence, they will always remain synchronized.

2. If the contents of CTA is lower than the contents of CMF, there is apparently a transaction on a master record which does not exist. This may be erroneous input or, depending on the application, it may mean that a new master record must be added to the output file. After whatever processing may be required, a new transaction record must be obtained, so that the read switch for the file TRANSACT is set to ON.

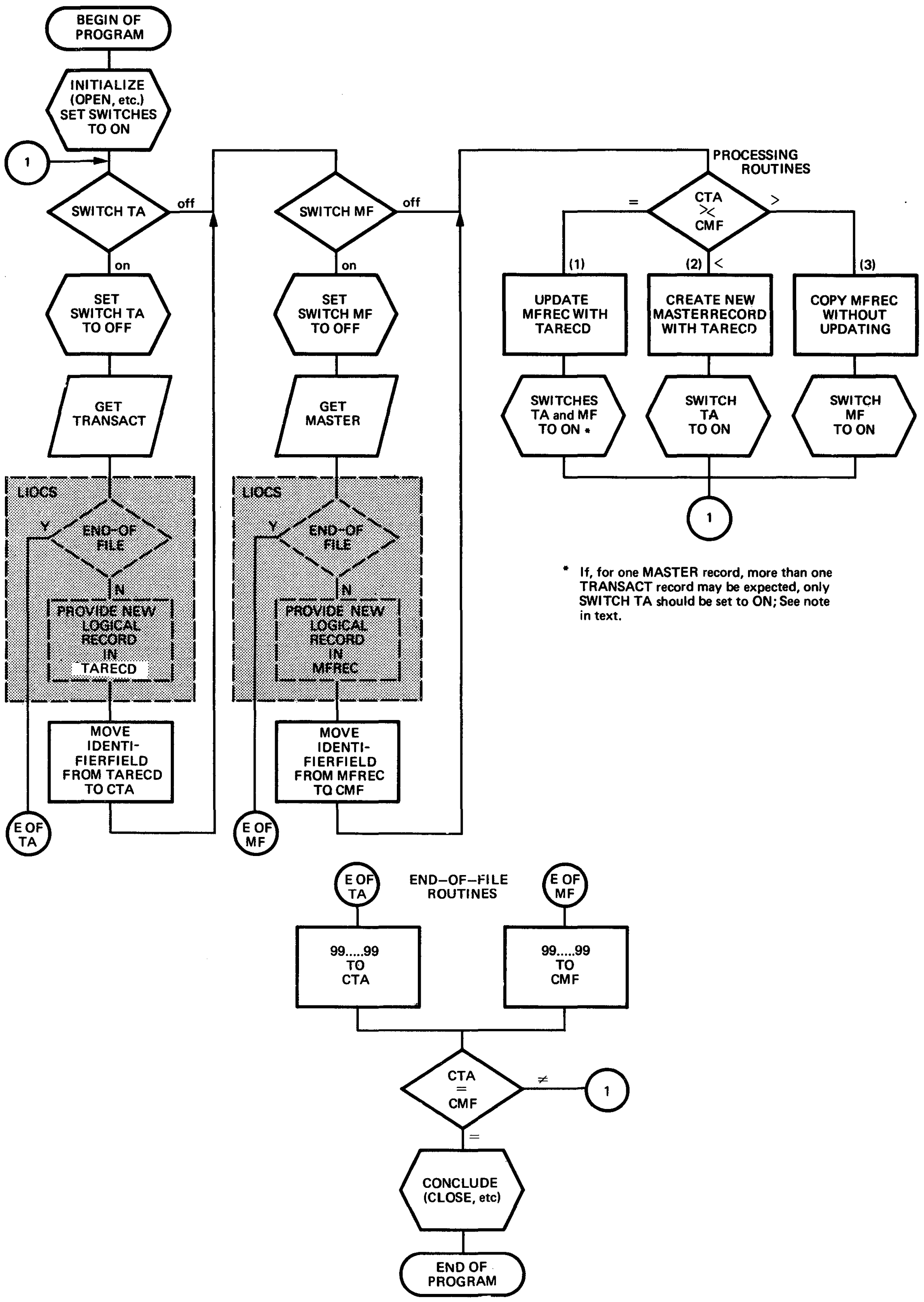

BEGIN OF PROGRAM
INITIALIZE (OPEN, etc.) SET SWITCHES TO ON
1
SWITCH TA
off
on
SET SWITCH TA TO OFF
GET TRANSACT
LIOCS
Y
END-OF FILE
N
PROVIDE NEW LOGICAL RECORD IN TARECD
MOVE IDENTI-FIERFIELD FROM TARECD TO CTA
E OF TA
SWITCH MF
off
on
SET SWITCH MF TO OFF
GET MASTER
LIOCS
Y
END-OF FILE
N
PROVIDE NEW LOGICAL RECORD IN MFREC
MOVE IDENTI-FIERFIELD FROM MFREC TO CMF
E OF MF
PROCESSING ROUTINES
CTA >< CMF
=
>
(1)
(2) <
(3)
UPDATE MFREC WITH TARECD
CREATE NEW MASTERRECORD WITH TARECD
COPY MFREC WITHOUT UPDATING
SWITCHES TA and MF TO ON *
SWITCH TA TO ON
SWITCH MF TO ON
1
* If, for one MASTER record, more than one TRANSACT record may be expected, only SWITCH TA should be set to ON; See note in text.
E OF TA
END-OF-FILE ROUTINES
E OF MF
99.....99 TO CTA
99.....99 TO CMF
CTA = CMF
≠
1
=
CONCLUDE (CLOSE, etc)
END OF PROGRAM

Figure 4.5. An updating program with end-of-file conditions on two input files
One of three processing routines is used depending on whether the ID of the transaction record (1) matches, is (2) lower than, or (3) higher than the ID of the master record.

3. If the contents of CTA is higher than the contents of CMF, there was apparently no transaction for the master record in the transaction file. The master record therefore can be written to the output file without any updating. A new master record must now be obtained, so that the read switch for the file MASTER is set to ON.

Eventually, one of the two input files will reach end-of-file. The only action that really remains to be done is to prevent this file from being read again; this is done in the end-of-file routine by setting the identifier field to the highest possible value (all nines, for example, or binary ones). As a result, the mechanism that synchronizes both files will only read the file which did not reach end-of-file. After this second file also has been processed completely, its identifier field is also set to the highest possible value; both fields are then equal, and the program can be terminated.

Note: For an input file on an IBM 5425 MFCU, the end-of-file indicator (/* or user-defined) must be followed by a blank card.

Logic Modules for Sequential Access Processing

The logic modules available with the Sequential Access Method must be assembled by the user from a source statement library, supplied by IBM. This is a one-time process. Once assembled, the logic modules can be stored in the relocatable library.

The logic modules can be linked with any problem program that requires them by the linkage editor. If preferable, however, they can also be assembled along with the user's program and included in the same output object module.

The logic module for a specific type of file in a particular problem program is assembled on a selective basis, according to the requirements for that file, specified by the user through parameters in the xxMOD macro instruction. These parameters specify the functions that the particular module is to provide. The functions provided by a logic module vary depending on the characteristics of the file, the type of device on which the file resides, and the activities to be performed on the file.

There are different xxMOD macro instructions for different device types:

• CDMOD	Card Module
• PRMOD	Printer Module
• MTMOD	Magnetic Tape Module
• SDMODxx	Sequential Disk Module
• PTMOD	Paper Tape Module
• MRMOD	Magnetic Reader Module
• ORMOD	Optical Reader Module
• DIMOD	Device Independent Module
• DUMODFx	Diskette Module

The characteristics of the file are specified as parameters of the DTFxx macro instruction, which generates a DTF table, serving as a link between the user's program and the logic module for a certain file:

• DTFCD	Card files
• DTFPR	Printer files

• DTFMT	Magnetic tape files
• DTFSD	Sequential disk files
• DTFPT	Paper tape files
• DTFOR	OCR files
• DTFMR	MICR files
• DTFDU	Diskette files
• DTFDI	Device independent files
• DTFCN	Console files
• DTFSR	Serial device files

For console and serial device files, no logic modules need to be generated, or headed at object time.

Direct Access Method

Devices and Record Formats

The Direct Access Method supports the following DASD equipment:

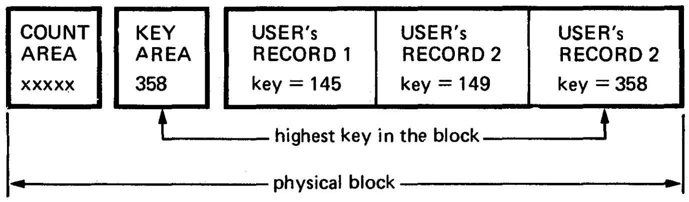
- IBM 2311 Disk Storage Drive
- IBM 2314 Direct Access Storage Facility
- IBM 2319 Disk Storage
- IBM 3330 Family of Disk Storage Devices
- IBM 3340 Disk Storage
- IBM 2321 Data Cell Drive

The equipment listed is described in Appendix 1; the data capacities of the devices are presented in Appendix 2.

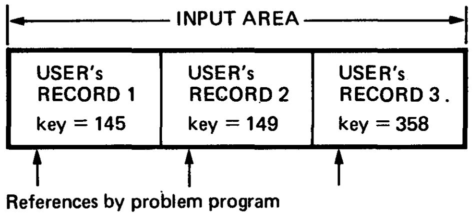
Only unblocked records, with or without a key area, are processed by the Direct Access Method. That is, a physical block is regarded as containing one logical record. If any blocking is desired, this must be done by the programmer in his problem program. Since the physical location of a logical record is determined through a randomizing algorithm it is usually impractical to have logical records blocked and, at the same time, physical blocks written with a key area. This is explained in the following example:

Assume a direct access file consisting of physical blocks with a key area, each block containing three logical records; blocking/deblocking is done by the problem program.

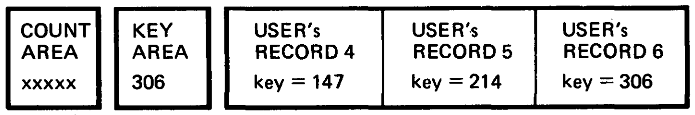
Problem 1: A physical block may have only one key; which key of the three is taken? Assume that the highest key in a block is selected and that one block in the file looks as follows:



Problem 2: Assume that an application program wants to retrieve the second logical record from the block above (key=149). This program will not be able to refer to this block on the basis of its key, since the key of the physical block is 358. Therefore the program can only refer to the block by its physical location in the file which is to be determined from key 149, through a randomizing algorithm. In the block above, the randomizing algorithm is thus supposed to calculate the same DASD address for the keys 145, 149, and 358.


INPUT AREA
USER's RECORD 1 key = 145
USER's RECORD 2 key = 149
USER's RECORD 3. key = 358
References by problem program

Problem 3: Another physical block in the same file might contain three records with the keys 147, 214, and 306; it then would look as follows:


COUNT AREA xxxxx
KEY AREA 306
USER's RECORD 4 key = 147
USER's RECORD 5 key = 214
USER's RECORD 6 key = 306

From the two blocks the real problem becomes clear: retrieval of individual records from any block, by means of a search on key on the device, can be done only if the user knows, for each individual record in the file, the key of the block in which that record occurs. In effect this means that only unblocked records with a key are practical, or in those cases where the logical records are physically kept in sequence according to their keys. In the latter case, retrieval might be possible by means of a search for a key, equal to or higher than the one supplied by the user. However, this organization is adequately treated by ISAM and VSAM, to be described later.

The following formats can be applied in DAM:

- Fixed-length format (Format F)
- Variable-length format (Format V), spanned or unspanned
- Undefined format (Format U).

If record spanning is used, the segmentation of logical records and the reassembly of logical records from a sequence of segments is performed by LIOCS routines whenever necessary.

Data can be written with or without a key area. Regardless of the record format, the length of the key area must be fixed for a file. Moreover, if spanned records are used, a key will be written only before the first segment of a record.

The various record formats, and the record structures possible on DASD, were explained earlier in this manual, under *Record formats and record structures*.

Locating Data

The Direct Access Method requires DASD addresses for all read or write operations. These addresses may be supplied in two ways:

- As an actual physical DASD address that specifies the location of a physical block within the entire system.
- As a relative track address that specifies the location of a physical block within the file.

Physical Track Addressing

An actual physical DASD address can be shown as an 8-byte binary address in the form mbbcchhr.

m identifies the volume.
A single file may be contained over more than one volume. If this is the case the physical units must be assigned (in EXTENT control cards) to a sequential set of symbolic unit numbers. The value of m is always 0 for the first volume, 1 for the second, 2 for the third, etc.

For example, a single logical file located on three volumes could be assigned to the logical unit numbers SYS002, SYS003, and SYS004. Here, m=0 refers to SYS002, m=1 refers to SYS003, and m=2 refers to SYS004.

The value of m is never actually read or written on the storage device. It references the proper element in the LUB table (see *Physical devices and symbolic device names*).

bb is a 2-byte field specifying a cell number (0-9). It is used for the 2321 data cell only; for disk bb is set to zeros.

cc is a 2-byte field that contains the cylinder number in binary form.

hh is a 2-byte field containing the head number in binary form. The first byte is reserved.

r is the record number within a track. This 1-byte field can contain a binary value of 0 to 255 to identify the physical location of a record on a track. r is not always used: it is only required when records are referenced by record ID. Records can also be referenced by record KEY, in which case r is not used.

The 8-byte DASD addresses described above are used either as a starting point for a search on record KEY (control field) or as the actual address for a read or write operation. When searching for a key, the programmer has the option of specifying that the search be only within the specified track (hh) or from track to track starting at the address given and continuing either until the record is found or until the end of the cylinder (cc) is reached.

Relative Track Addressing

The required DASD address may also be given as a relative address, which is then converted by IOCS to an actual address. Relative track addressing is more convenient to use than the actual physical address for the following reasons:

- The data in the file is treated logically as if it were located in one continuous area, although it may be physically non-continuous.
- The user needs to know only the relative position of the data within the file; its actual physical address is not required. This is especially advantageous if the user plans to move the file from one location to another. In such cases the relative addressing scheme remains the same whereas the actual addresses will be automatically converted by IOCS.

The user may specify relative addresses in either of two formats:

- Hexadecimal, in the form tttr.
- Zoned decimal, in the form ttttttttrr.

In hexadecimal, ttt represents the track number relative to the start of the file, and r represents the record number on that track. In zoned decimal, tttttttt represents the track number relative to the start of the file, and rr represents the record number on the track. The hexadecimal notation requires 4 bytes, while the zoned decimal notation requires 10 bytes.

It should be noted that the addressing techniques described above are used by the DOS/VS operating system, and may be applied in Assembler language. Addressing in a high-level programming language, such as COBOL or PL/I, may be different. Information about DASD addressing in a high-level programming language should be obtained from the appropriate reference manuals.

For certain types of operations, the system can be requested to return the actual record address (ID) of the block read or written, or of the block following the one read or written. This returned ID can be used to either read or write a new record, or to update the one just read and write it back.

For example, to delete a logical record from a direct access file that contains physical blocks with a key area, the programmer can randomize the primary key of that record to a starting location (track or cylinder address), search on key to read the block, and then use the ID returned to write a blank or zeroed block (key and data) to this location.

If a user wants to use deleted blocks again for other data, he may randomize the primary key of a new logical record to a starting location, search on key to find a blank or zeroed key (see note), and then use the ID returned to write the new record with the new key into the same location.

Note: In the examples, it was assumed that a deleted block would be written as "all blanks" or "all zeros". However, any unique identification is acceptable and the choice is entirely up to the user. See also "Loading and Processing a Direct Access File".

The format of the returned ID will be the same as the format of the DASD address that is used for locating data mbbcchhrr, tttr, or tttttttrr.

Detailed information about the returning of IDs can be found in *DOS/VS Supervisor and I/O Macros*, GC33-5373.

Capacity Record

DASD design allows the operating system to locate space on a track for writing a physical block. For this purpose, the Direct Access Method maintains a capacity record as a part of record zero on a track (see *Record structures for the various devices* in the section *Introduction to DOS/VS data management*).

When a record must be written, the system will:

- Read the data portion of record zero (=capacity record).
- Determine whether there is space on the track for the record.
- If the new record fits, write it to the track as a new last record, and update the capacity record.
- If there is not enough space on the track, notify the problem program. An overflow routine in the problem program may then become active.

This design makes a randomizing problem less critical than in the past when every single record was supposed to have its unique location. Each synonym resulting from a conversion algortihm resulted in an overflow record. Now the conversion algorithm may randomize to a track address, and more than one record may have the same address assigned by the algorithm.

The capacity record is not always used. The description of the WRITE macro instruction (under *Assembler Language Considerations*) explains when it is used.

The capacity record is updated for each record that fills empty space on a track. When a record is deleted, however, the capacity record does not show it as empty space. A deleted record can only be recognized by the user as 'free space'. This consideration has consequences for processing.

Loading and Processing a Direct Access File

The only difference between loading (creating) and processing (updating or retrieving) a direct access file is the file's initial status. In both cases, the same conversion algorithm is used for locating data blocks, and the entire file must be online.

Note: Multivolume direct access files on a 3340 cannot extend over different types of data modules.

Before creating a file, however, the user should be sure that the disk storage area is cleared of any data that may have been stored previously. IBM provides two utility programs to clear disk storage areas:

- Initialize Disk.
  This utility program operates on complete volumes only. It may write only a preformatted VTOC, or it may also clear the entire volume so

that each track contains a home address and a record zero with zero capacity. The preformatted VTOC contains empty file labels. Although the initialize disk program cannot clear a portion of a volume, the user can do so by writing a complete file consisting of erased tracks preceded by R0 with the desired contents.

- Clear Disk.
  This utility program operates on logical files. It is used to preformat a disk storage area with dummy blocks of fixed length format. It can be used either on a new pack after the Initialize Disk program, or on a used pack to clear data areas for a new file.
  Preformatting by means of the clear disk program is necessary for a file of fixed length data blocks.

Additional information about these two utility programs is presented in the section *DASD Initialization and Maintenance*.

File Organization

A file that is organized randomly usually is designed with two types of data areas:

- Prime data areas
- Overflow areas.

Overflow areas are used for data blocks that cannot be placed in the prime data areas. Overflow areas can be spread over the file (for example in the last tracks in each cylinder), or as one continuous area for the entire file (for example in one or more complete cylinders outside the file, even on a different volume).

The choice of a good conversion algorithm is very important. A poor algorithm will produce many synonyms so that overflow areas may prove to be too small and the space in the prime data area is used ineffectively. This means a waste of space. If a good algorithm is used, however, the random access technique is undoubtedly the most flexible method of all.

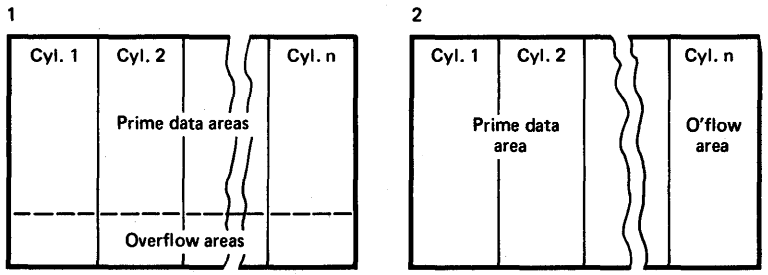


Figure 4.6. Two examples of overflow area organization
Overflow areas may be in the last tracks of each cylinder, or in separate cylinders.

The strict requirement for a good randomizing formula is lessened by the fact that the design of the IBM direct access devices allow, in certain cases, randomizing to a track address instead of down to the record address; in some cases, even a cylinder address may be sufficient. This is explained in the following text. It is important, however, to keep in mind that this manual describes the data management features offered by DOS/VS in a context of assembler programming language; it is quite possible that a high-level programming language does not offer the same features. Those languages are subject to standards that are based on minimum processing requirements throughout the industry and do not apply to all features of specific devices. The user, therefore, must consult the manuals of the language processor he is using for a description of the facilities offered.

Prime Data Organization

Different methods can be used for randomizing. The choice depends on the record structure (with or without key) and the record format (fixed or variable length).

Data Without a Key Area

If records are written without a key, the location of a data block can be uniquely identified by the randomizing algorithm only. The device has no means of identifying a data block other than by the record address as specified by the user.

For data without a key, the most practical method of randomizing is to establish a converstion algorithm that calculates a cylinder, track, *and record address.* This implies that fixed-length records must be used, and that the file will be preformatted by means of the clear disk utility program, before being loaded. Variable-length blocks cannot be processed randomly without a key on the basis of a unique DASD *record* address, since writing to a record address requires an existing block at that address, and the size of that block cannot be predicted before the size of the new data block to be inserted is known.

All of this makes the conversion algorithm for data without a key more critical, since each synonymous record becomes an overflow record. The procedure to be followed is complex. Generally speaking, the following procedure must be used for adding a new record:

1. Compute a DASD record address by means of the randomizing algorithm.
2. Check whether the block on the address computed contains current data or not. This requires an input operation.
3. If the block contains no current data, write the new record of the computed address. Clear the overflow pointer (see Figure 60).
4. If the block does contain current data, find a proper place for the new record in the overflow area. This problem is discussed under "Overflow organization".
5. The block, read in step 2, may already have a synonym in the overflow area, so that the overflow pointer (Figure 60) will be filled. If the overflow pointer is filled, save the contents and replace these by the address of the new record, as found in step 4. Then restore this block to its original location.
6. The contents of the saved overflow pointer (step 5) are now put in the overflow chain pointer which is part of the new data block to be inserted. As a result, a new synonym becomes the new first overflow record in a chain, as shown in Figure 60. Finally, write the new data block into to the overflow area, at the address found in step 4.

A result of this method may be that a chain of records must be searched before the right record is found. Also, a requirement of this method is that the pointers must be adjusted, if a record is deleted.

There is an alternative method that can be used for records without a key. In this method, the randomizing algorithm calculates a cylinder and track address only. The user must then check to see whether the track can accommodate the new record. For fixed-length records this means that a record-by-record scan must be performed until a record is found that contains no current data. For variable-length blocks this method is not practical: it may impose serious retrieval problems. Most likely more than one block must be read until the right one is retrieved. Also the overflow area must be used if the track is full. Since overflow records are now chained by track, the overflow chains may be much longer than when randomizing down to a record address. As a result, this procedure will probably be rather time consuming, and is therefore not very attractive.

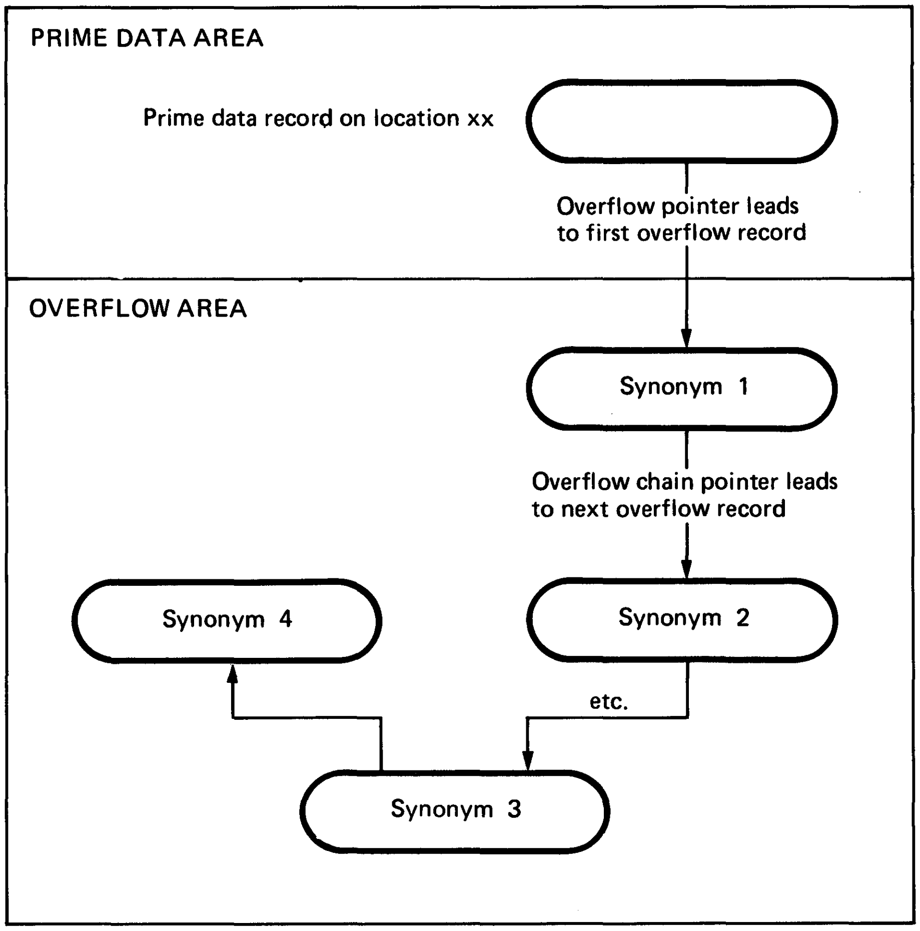

PRIME DATA AREA
Prime data record on location xx
Overflow pointer leads to first overflow record
OVERFLOW AREA
Synonym 1
Overflow chain pointer leads to next overflow record
Synonym 4
Synonym 2
etc.
Synonym 3

Figure 4.7. Prime data record and overflow records
The last new record inserted in an overflow chain becomes the first overflow record (synonym 1).

Data With a Key Area
If records are written with a key, certain functions can be performed by the device. In the following text, a distinction is made between fixed-length data blocks, and variable-length data blocks.

Fixed-Length Blocks, With a Key Area
Files should be preformatted by means of the clear disk utility program. The file then contains dummy records of fixed length specified by the user. If the user specifies the same contents for both dummy records and deleted records, he can use the same procedure for both creating and for updating the file.

The key of a block distinguishes between a current data block and a dummy block. Keys for dummy blocks have the same content; keys of current data blocks are unique, each key identifying a particular data record.

A dummy key identifies an empty location. The user has two options for adding a new record. He can:

1. Randomize to a CYLINDER address only.
   The user should specify that he wants the option to search multiple tracks. This allows him to search for the first dummy record in a cylinder, starting at the beginning of the cylinder that is specified by the address obtained from the randomizing algorithm.
   In this case, he may use one or more separate cylinders as an overflow area if the search for a dummy record is not successful.
2. Randomize to a TRACK address (includes a cylinder address).
   The user may or may not specify that he wants to use the search multiple tracks option. If he uses this option, the procedure is the same as above, except that the search begins at the track specified instead of at the beginning of a cylinder. If he does not specify the option, the user may for a dummy record on a specific track. The search continues until either a dummy block is found or the end of the track is reached.
   In this case, the last few tracks in each cylinder may be used as overflow tracks for that cylinder.

In either case, the system will return control to the problem program. It will return with a record address unless the search was unsuccesful. If an address is supplied, it can be used directly to write the new data record if the search multiple tracks option is used. If this option is not used, the address supplied reflects the block following the one that was searched for.

If no address is supplied, the system communicates 'no record found' to user. The overflow procedure must then become active, as will be discussed under *Overflow organization.*

Variable-Length Blocks, With a Key Area
The files should not be preformatted with the clear disk utility program. The initialize disk program can be used to clear a pack completely, or the user must clear a particular area on a pack himself (Assembler language: WRITE filename, RZERO), track by track.

On each track of a file that contains variable-length blocks, record zero contains a count field that states the amount of free space at the end of that track. Deleted records are not taken into account. (Unlike fixed-length blocks, deleted variable-length blocks cannot be re-used for other data records.)

The user should always establish a randomizing algorithm that delivers a TRACK address (implies a cylinder address). The system checks the contents of record zero (capacity record) to determine if the track can accomodate the new block. If so, the new block is written after the last block on that track. If there is not enough space left, this is communicated to the user; he must then direct the new record to the overflow area. (Assembler language: WRITE filename, AFTER).

Space occupied by a deleted record cannot be used for another new record. This means that if a file contains variable-length blocks and is frequently updated it may need to be reorganized from time to time. This can be done by reading the file track after track, clearing each track separately (Assembler language: WRITE filename, RZERO) and then restoring each current data block back as if it were new. Since deleted records are not restored, free space is again concentrated at the end of the tracks. After the prime data tracks have been reorganized, the overflow area may then be processed, and an attempt made to write overflow records to the prime data area. Overflow records that cannot be moved to the prime data area are moved back into the overflow tracks, omitting deleted records.

Retrieving Records With a Key Area
Records may be retrieved by a search on key. If the option for a search on multiple tracks is specified, a record can be found on a cylinder, as long as the user specifies the start of the search at, or before the record address. Thus, the same conversion algorithm that is applied for writing a record can be used for retrieving it.

Summary
A brief summary of the techniques discussed above is presented in Figure 4.8.

LOADING AND PROCESSING RADOM FILES.

1. RECORDS WITHOUT A KEY. (Fixed length data blocks)

Storage:	Randomize to a RECORD address. Each synonym becomes an overflow record, to be inserted logically in an overflow chain.
Retrieval:	Randomize to the RECORD address. Read record and check if it is the one desired. If not, search the overflow chain.

2. RECORDS WITH A KEY.

   Fixed length blocks.
   File is preformatted by the clear disk program.
   Randomize to TRACK address only, or to CYLINDER address only.
   A record will be an overflow record only if the search for a dummy record is not successful.

   Variable length blocks.
   File is not preformatted with dummy records; record zero is used to determine if a track still has enough space left for a new record.
   Randomize to a TRACK address.
   Records will become overflow records, if the track specified by the randomizing algorithm has enough space left.

   Retrieval.
   The same randomizing can be used for updating and for retrieval.

Figure 4.8. Summary of randomizing methods

Overflow Organization

Whenever a file is organized randomly and a randomizing algorithm is applied, the user should include overflow areas in the file organization. The two basic ways of organizing overflow areas are:

- Overflow tracks per cylinder in a file; these tracks are the last tracks of each cylinder.
- One separate overflow area for an entire file. This area is kept separately in one or more complete cylinders, or even on a separate volume.

A combination of the two methods is also possible. The user may design an overflow area in the last tracks of each cylinder and, in addition, an independent overflow area that is used when the cylinder overflow area itself overflows. This is shown in Figure 4.9.

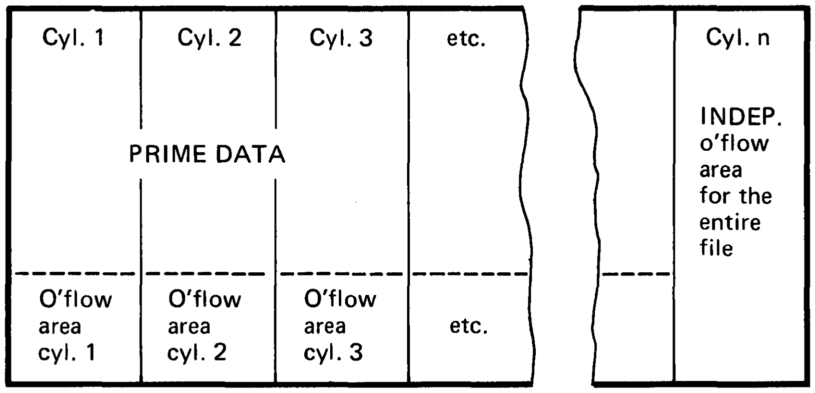


Figure 4.9. Sample overflow organization
In this case there is an additional separate overflow area to supplement the overflow area on each cylinder.

The independent overflow area may be used as an overflow area without any special structure. If a record cannot be fitted into the prime data area according to the randomizing algorithm, or the cylinder overflow area if that area is completely full, it may be placed anywhere in the independent overflow area.

The independent overflow area may also be seen as an extension of any prime data cylinder. For example, if a record must be stored in track xx of cylinder yy according to the randomizing algorithm, this record can also be placed in track xx of the independent overflow cylinder (assuming one separate cylinder) or in track xx of any cylinder of the independent overflow area (assuming multiple cylinders for the independent overflow area). This means that the user can retrieve a record in an independent overflow cylinder, by updating the randomizing algorithm, which is used to search the prime data areas, with appropriate cylinder addresses. If only an independent overflow area is used, more I/O is required than if cylinder overflow areas are also used.

Whether or not the overflow areas are efficient depends on the organization of the area in relation to the record format and the block structure.

Fixed Length Records, With a Key Area
In 'Prime data organization', techniques are explained for locating data in a prime data area. The same techniques are applicable for overflow areas.

If the search multiple tracks option is specified, a search for a dummy record in the prime data cylinder will continue until the record is found or the end of a cylinder is reached. A cylinder overflow area is not very useful here, unless the last tracks of each cylinder are excluded by the randomizing algorithm. In the latter case these tracks automatically become an overflow area. In fact, all tracks that follow a calculated track address are acting as an overflow area, as long as they are in the same cylinder.

If the search multiple tracks option is not specified, a search for a dummy block will not extend beyond the specified track. If the record is not found, the user must issue a search on each subsequent track, until the record is found. This method is probably more time consuming, but it gives the user more direct control. The user can choose between cylinder overflow areas and independent overflow areas, but is is difficult to predict which method will be most efficient. If the prime data area and the independent overflow area reside on the same volume, a switch to and from the overflow cylinders requires a movement of the read/write mechanism, which can be avoided if cylinder overflow areas are used.

Variable-Length Records, With a Key Area
As explained under 'Prime data organization', a cylinder search for a specific key is only practical for retrieving data blocks. If a new record is to be added, the randomizing algorithm must specify a track address. Using the contents of the capacity record in R0, the system determines whether or not the specified track can hold the new record. If not, the user can perform this inquiry in the overflow area in exactly the same way, track by track, until a track is found that can contain the new record. It is useful to design cylinder overflow areas in each cylinder as well as a separate independent overflow area. In case a prime data track overflows, the user should first try to store the record in the cylinder overflow area of the cylinder with the prime data track. If this is not possible, he should then store the record in the independent overflow area. If the record can be stored in the cylinder overflow area, it can be retrieved automatically if the search multiple tracks option is specified by the retrieving program, If it cannot be stored in the cylinder overflow area, the user must search each independent overflow cylinder until the record is found. Since records cannot be stored in the space occupied by deleted records, the cylinder overflow areas themselves may overflow. Reorganization of the entire file will then be necessary, in order to sustain processing efficiency.

Records Without a Key Area
Since there is no key for the system to search for, each overflow record must be accessed directly by a unique record address. Since the randomizing algorithm calculates only prime data addresses, the user must establish an address in the overflow area by another method.

The user must be able to find the address of a 'free' record location without having to scan the entire overflow area. Otherwise, he will lose time in searching the overflow area, block by block. A good method is to reserve the first record of the overflow area as an 'overflow area descriptor record'. This record contains, at all times, the address of the first free block in the overflow area. This block has a pointer to the next free block. If a new record must be added to the overflow area, the 'overflow area descriptor record' gives the direct address of the block where this new record can be stored. The pointer to the next free record is then moved to the 'overflow area descriptor record'. At the same time, the new overflow record is added to a chain, as explained in "Prime data organization".

When a record is deleted from the overflow area, the address of that block is moved to the 'overflow area descriptor record', and becomes the address of the new first free record. The address that was already in the 'overflow area descriptor record' is moved to the block that just became free, becoming the pointer to the next free block. The examples in Figure 4.10 illustrate this process.

In Figure 4.10, block 1 in the overflow area is the 'Overflow Area Descriptor Record'. The data portion of this block may contain any information a user requires, in addition to a pointer that points to the first 'free' block that can be used for a new overflow record. In the top diagram it points, as an example, to block 3. Block 3, in turn, points to block 5 as the next 'free' block, etc. Thus, starting in block 1, a user can easily locate all blocks that are free.

In the prime data area, each block has a pointer to the overflow area. If no synonyms are present for a certain prime data record, this pointer will be empty. If synonyms are present, however, this pointer will point to the first synonym in the overflow area. In the top diagram of Figure 4.10, for example, block 2 in the prime data area points to block 6 in the overflow area as being the first synonym for prime data block 2. This synonym location, in turn, points to a next synonym, if any, and so on. Thus, overflow blocks 6 and 7 form the beginning of an overflow chain for prime data block 2.

If, for example, a new record must be placed on prime data location 2 (according to some conversion algorithm), this new record must be placed into the overflow area since prime data block 2 already contains current data. In this situation, the new record can be written into overflow block 3, which is the first 'free' block, and added to the overflow chain that already exists. The central diagram in Figure 4.10 shows the situation after the new record has been added. Note that the new record becomes the first overflow block in the overflow chain, and that block 1 in the overflow area now points to another first 'free'overflow block.

When a block must be deleted from the overflow area, the user obviously must locate it properly following the overflow chain. The deleted record becomes the first 'free' overflow block. The bottom diagram in Figure 4.10 shows the situation after deleting block 2 from the overflow area.

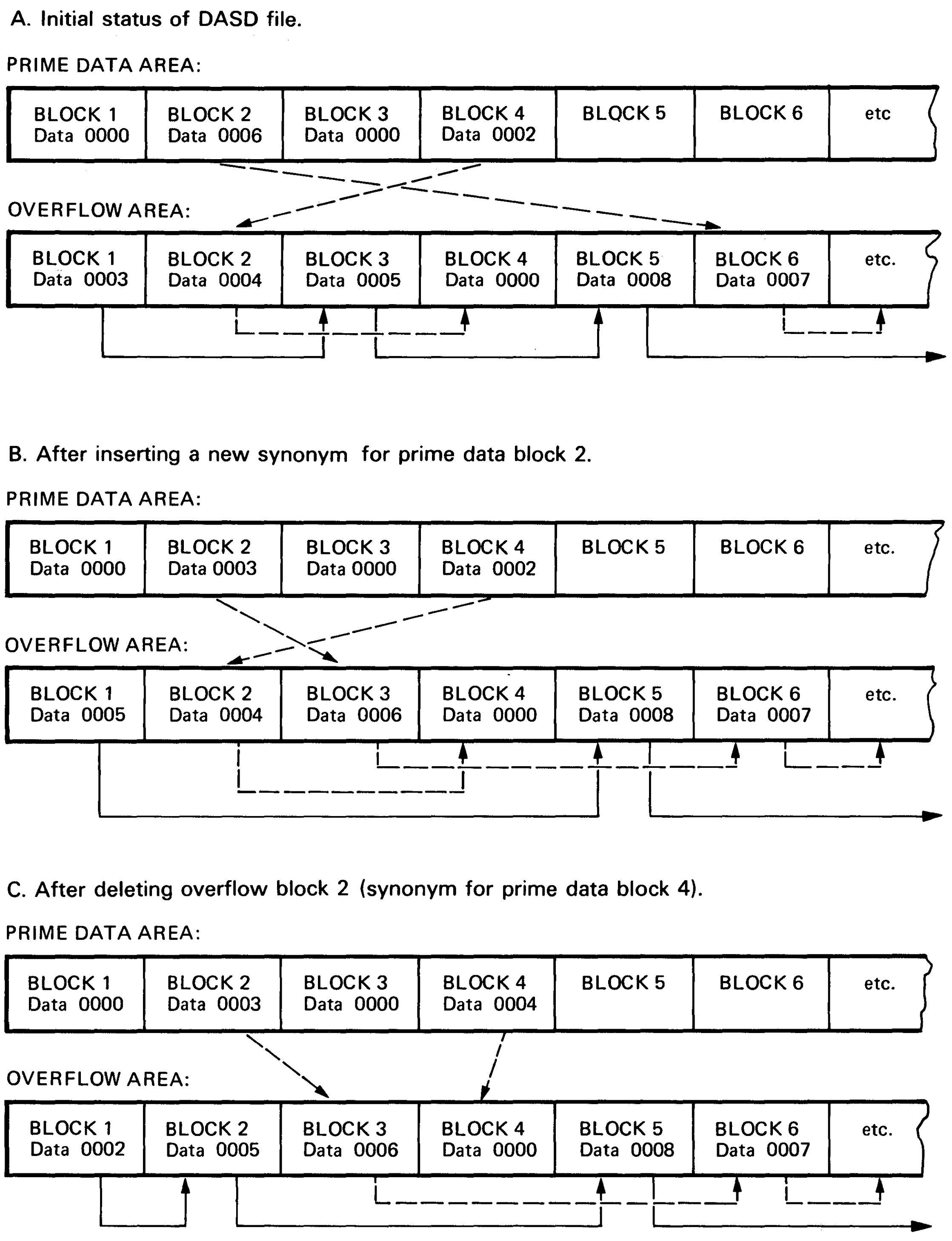


Figure 4.10. Sample overflow organization
Note: The 'Free-record pointers' in the overflow area must be written by the user after the file is preformatted by the Clear-Disk program, and before the file is loaded for the first time.

Assembler Language Considerations

The DOS/VS DAM macro instructions the programmer to take advantage of the flexibility of the direct access devices with a minimum of effort. Reading and writing of data blocks (by READ and WRITE macro instructions) is performed in basically the same way as with the physical IOCS EXCP macro instruction, except that the system provides the CCW chains.

Macro Instructions for Random DASD Processing

The following macro instructions are provided for processing a file by means of DAM:

DTFDA	Define The File for Direct Access
READ	Read a block of data, with or without the KEY
WRITE	Write a block of data, with or without the KEY, or: Erase a track, resetting R0 to maximum capacity, or: Write EOF (End Of File)
CNTRL	CoNTRoL: perform a SEEK on DASD
WAITF	Wait for completion of a READ or WRITE operation

These macro instructions are explained in the following sections in the context of particular functions such as reading and writing.

Reading Blocks of Data

Data blocks can be referenced by KEY or by ID (record location). If referencing by KEY, the programmer supplies the key field of the record to be read and the track address at which the search is to begin. If referencing by ID, the programmer supplies the track *and record address* of the record to be read.

For reference by KEY, the macro format is: READ filename, KEY. If the search-multiple-track option is specified in the DTFDA macro instruction (SRCHM=YES), the system searches until it locates the desired record, or to the end of the cylinder. If the search-multiple-track option is not specified, the system searches only the specified track.

For reference by ID, the macro format is: READ filename, ID. The system searches on this ID and then reads the key and data fields, or the data portion only if the key field is not used.

LIOCS can be requested to return the ID of records after reading. The user must specify the name of the field that contains returned IDs in the IDLOC=name parameter of the DTFDA macro instruction. The ID returned is:

- For reference by key:
  search-multiple-track specified: the ID of the record read.
  search-multiple-track not specified: the ID of the record following the one read.
- For reference by ID:
  The ID of the record following the one read.

The READ macro instruction returns control to the problem program after requesting PIOCS to execute a CCW chain. The programmer can perform any processing desired and then issue a WAITF macro instruction to check for completion of the read operation.

Writing Blocks of Data
Data blocks can be written as new records, or as updates for existing records. If a new record is written over a dummy record, this is also treated as an update. The system can assign completely new records to a record location by means of the capacity record (part of R0). For overwriting existing records, such as updating or making use of dummy records for actual data, a reference is made either by ID or by KEY.

Writing New Records
This is done with a WRITE macro instruction with the format: *WRITE filename, AFTER*. The programmer supplies the track address. The system reads the capacity record of that track to see whether there is enough space left for the new record. If there is not, the problem program is notified. If there is enough space, the system searches R0 for the ID of the last record on the track, and then issues a write count, key, and data operaton for the new record. The new record is written directly following the record currently last on the track. The capacity record is then updated with the adjusted count of remaining bytes and the ID of the new record, and is restored. This WRITE macro instruction format cannot return any ID field.

Overwriting Existing Records
If reference is by ID, the macro instruction format is: *WRITE filename, ID*. The programmer supplies the track *and record address* of the record to be written. The system searches for this ID and starts a write key and data (or write data only) operation. If an ID must be returned, this will be the ID of the next record in the file.
If reference is by KEY, the macro instruction format is: *WRITE filename, KEY*. The programmer supplies the KEY of the record to be located and the address of the track on which the record resides. The system then searches that track for the key or, if the search-multiple-track option is specified in the DTFDA macro instruction, searches through the cylinder, starting with the track specified, until the key requested is located. When this key is found, a write data only operation is performed. If the DTFDA macro instruction specifies that an ID must be returned, this ID will be:

- search-multiple-track specified: the ID of this record written.
- search-multiple-track not specified: the ID of the record following the one written.

Write Verification
If the user specifies in the DTFDA macro instruction that write operations must be verified, a read command is issued after a write operation with any of the options ID, KEY, or AFTER. This is done without actually transferring data to virtual storage. The system then checks if the data, as it was recorded, is valid.

A special WRITE macro format causes an EOF record to be written after the last record on the track specified. This macro instruction format is: *WRITE filename, AFTER, EOF.*

Clearing a Track
The user can cause the contents of a track to be erased by specifying a WRITE macro instruction with the format: *WRITE filename, RZERO,*

and supplying the track address. The system searches for this track, restores the maximum capacity of the track in R0, and erases the remainder of the track after R0.

In all cases, the WRITE macro instruction returns control to the problem program after requesting services from PIOCS. The programmer can perform any processing desired, and then issue a WAITF macro instruction to check for completion of the write operation.

Seeks
The READ and WRITE macro instructions do *not* have to be preceded by a CNTRL-seek macro instruction. They automatically seek to the correct cylinder by means of the track address that is supplied by the programmer. However, it may improve processing speed to issue a seek in order to position the access mechanism to the correct cylinder before the actual ID or KEY required for a read or write operation is available. It should be kept in mind, however, that such a preliminary seek operation may be canceled if more than one problem program is operating on the same volume (not necessarily the same file) at the same time. A seek issued by one program can be destroyed after completion by another program that issues an I/O request on the same volume. The SEEK macro instruction returns control to the problem program as soon as the operation is initiated.

Completion of Read or Write Operations
The programmer must issue a WAITF macro instruction to check if a read or write operation has been completed. This macro instruction tests for errors and exceptional conditions. Any exceptional condition discovered is passed to a special two-byte field, the name of which is specified in the DTFDA macro instruction. This field must be defined in the problem program.

Logic Modules for Direct Access Processing
The logic modules available with the Direct Access Method must be assembled by the user from a source statement library supplied by IBM. This is a one-time process. Once assembled, they can be stored in the relocatable library.

Logic modules can be linke automatically with any problem program that requires them. If prefereable, however, they can also be assembled along with the user's problem program and included in the same output object module. The logic module for a specific problem program is assembled on a selective basis, according to requirements of the parameters in the DAMOD macro instruction supplied by the user. These parameters specify the functions that the particular module is to provide.

The characteristics of the file are defined by means of the DTFDA macro instruction. This macro instruction generates a DTF table which serves as an entry to the logic module generated by the DAMOD macro instruction.

Indexed Sequential Access Method

Devices and Record Formats

The Indexed Sequential Access Method supports the following devices:

- IBM 2311 Disk Storage Drive
- IBM 2314 Direct Access Storage Facility
- IBM 2319 Disk Storage
- IBM 3330 Family of Disk Storage Devices
- IBM 3340 Disk Storage
- IBM 2321 Data Cell Drive

The equipment listed is described in Appendix 1; capacities of the devices are presented in Appendix 2. Later in this chapter, space formulas for ISAM files are given.

Only fixed length record formats (Format F) can be processed by the Indexed Sequential Access Method. Logical records may be blocked or unblocked.

Data must be written with a key area. For blocked records this means that the key of the last logical record is taken as the key for the block. As a matter of fact, ISAM is the only access method that can process blocked records with a key area.

A separate chapter in this section describes the different physical block structures as they may occur in an ISAM file.

Indexes

The ability to read and write records from or to anywhere in a file with indexed sequential organization is made possible by indexes that are part of the file itself.

There are three types of indexes:

- One track index for each cylinder of a file
- One cylinder index for the entire file
- One master index for the entire file (optional).

Track Index

Each entry in a track index contains the highest key that is present on the track associated with that entry. As Figure 4.11 shows, a search through the sample track index for a record with a key 039 will find track 5 as its track address. This means that track 5 contains records with a key higher than 030 (highest key on track 4) and up to 040. If a record with a key 039 is present on track 5, it will be retrieved by a search on key for that track.

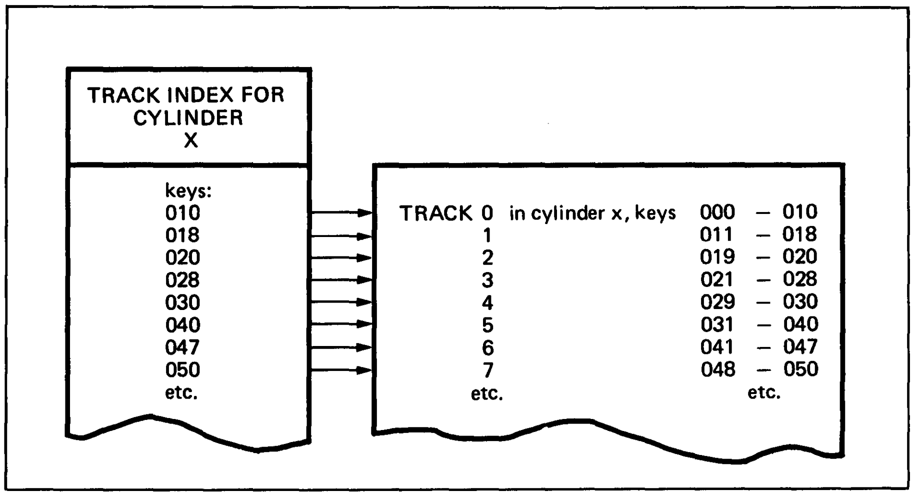


Figure 4.11. Track index
The record with key 47, for example, has the highest key on track 6.

Cylinder Index

Each entry in the cylinder index contains the highest key in the cylinder associated with that entry. As such it also covers the track index for that cylinder. So, from an entry in the cylinder index, an entry point for a specific track index is found. This is shown in Figure 4.12.

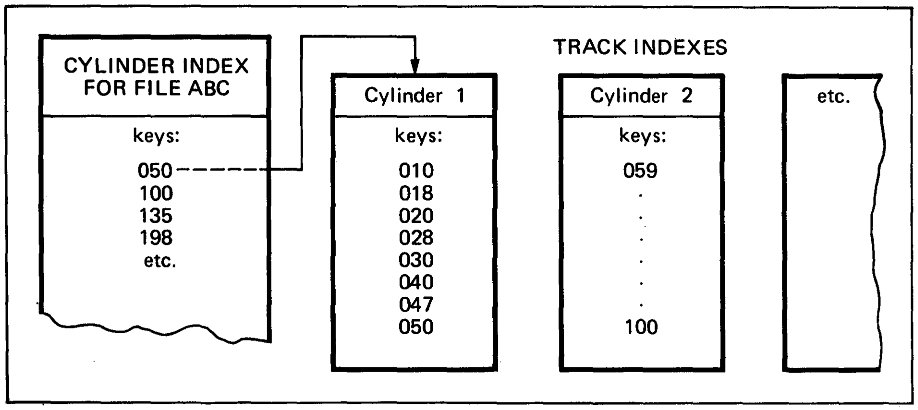


Figure 4.12. Cylinder index
The record with key 100, for example, has the highest key on cylinder 2.

Since it is necessary to search through cylinder index and track index in order to locate individual records, it is useful to have the cylinder index resident in virtual storage. This option speeds up the random retrieval and add functions when the number of records to be processed is significant. Since it is then not necessary to read the cylinder index, the processing time per record is decreased. See: *Resident cylinder index*.

Resident Cylinder Index

It is possible to have all, or part, of the cylinder index reside in virtual storage. If part of the cylinder index is kept in virtual storage it is advisable to pre-sort the input transactions to take full advantage of the resident portion of the cylinder index. If in such a situation the input is not pre-sorted, a certain portion of the index system will probably have to be read several times, so that the resident cylinder index option is not fully utilized. If all of the cylinder index can reside in virtual storage at the same time, there is obviously no need for pre-sorting the input transactions.

When the resident cylinder index option is used, there is no need at all for searching the master index (if any); the option is selected by specifying CORINDX=YES in the DTFIS macro instruction.

Master Index

If desired, a third index can be created: the master index. If a file occupies many cylinders, the cylinder index will be rather long and a search for a key is slow. A master index of an order higher than the cylinder index effectively reduces the length to be searched in the cylinder index is inefficient. Each entry of the master index points to a track of the cylinder index.

It is advisable to use a master index if the cylinder index occupies more than four tracks, and if the cylinder index is not kept permanently in virtual storage. The index structure with a master index is shown in Figure 4.13.

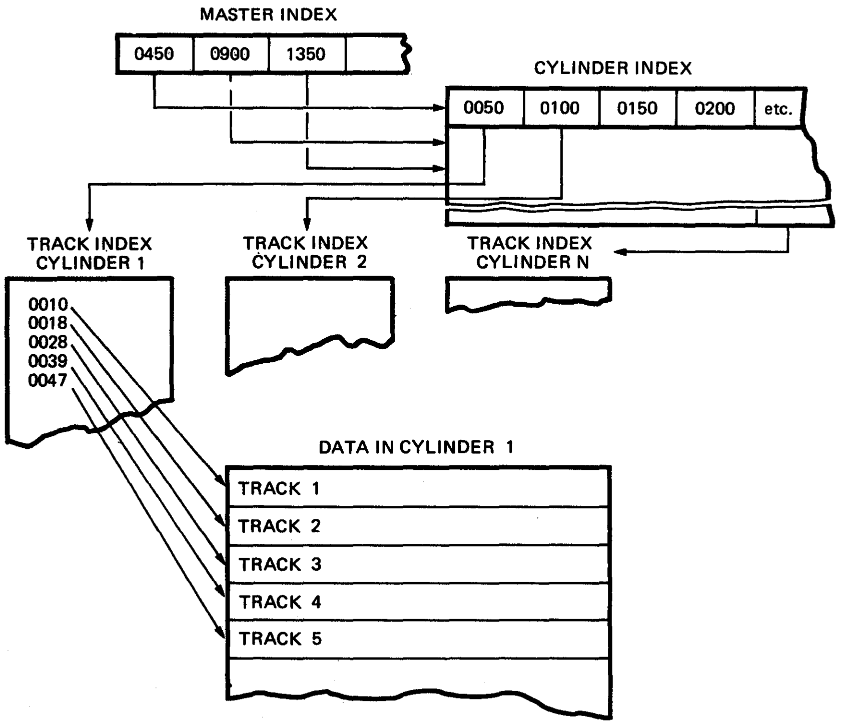


Figure 4.13. Basic index structure for an ISAM file
The master index points to the cylinder index, the cylincer index points to the track index, and the track index points to the data.

Prime Data Area

When creating an indexed sequential file, data blocks are written in the prime data area. This must be one continuous area which begins in the first track (track 0) of a cylinder, and ends in the last track of the same or another cylinder. For a multivolume indexed sequential file, the prime data area must continue from the *last* track of one volume to the *first* track of *cylinder 1* on the next volume, so that the prime data area is considered as continuous by ISAM. (Cylinder 0 of a volume is reserved for labels.)

Note: The prime data area of a multivolume ISAM file on a 3340 cannot extend over different types of data modules.

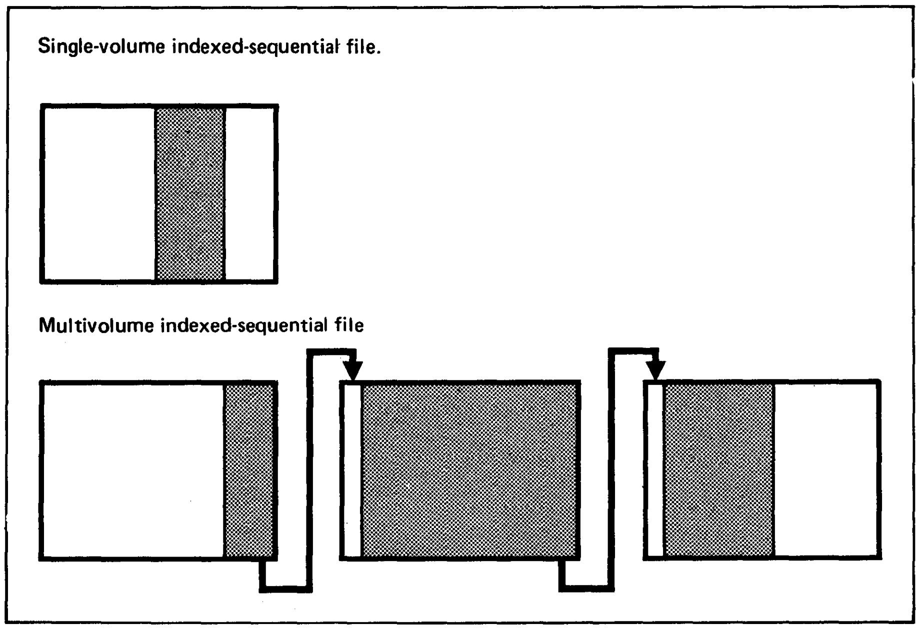
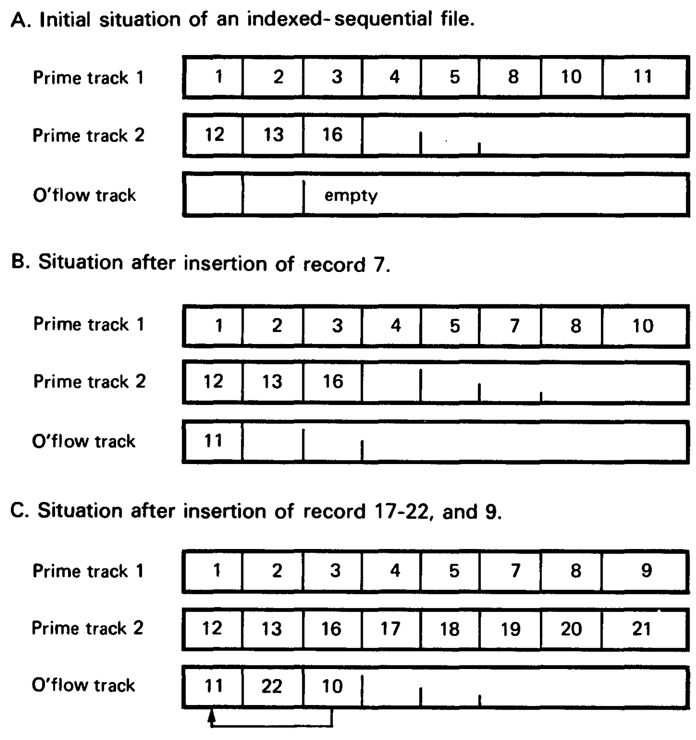

Single-volume indexed-sequential file.
Multivolume indexed-sequential file

Figure 4.14. The prime data area of an ISAM file
Prime data areas are continuous, except for the first cylinders on multivolume files.

Overflow Area

In addition to the prime data area, where the original records of an indexed sequential file were stored, an overflow area is provided for records that are forced off their original tracks by the insertion of new records. A new record added to an indexed sequential file is placed into a location on a track determined by the value of its key field. This implies that existing records with a higher key must be shifted. Shifting all of the records of an entire file would take too much time. Therefore, only the records on a track are shifted: only the track that is affected by the insertion of the new record, and only the records that contain a key higher than that of the new record, are shifted. A record that is shifted out of the track in this way is put into the overflow area. The track index is also adjusted in the process.

Figure 4.15 shows how records are added to an ISAM file. Figure 4.15A shows the file before records are added. Figure 4.15B shows the file after adding a record with key 7. In this picture, the record with key 11 has been moved to the overflow area. Figure 4.15C shows the situation after a series of records with keys 9, 17, 18, 19, 20, 21, and 22 have been added. The insertion of the record with key 9 causes the record with key 10 to be moved to the overflow area. The records with keys 17 ... 22 can be put at the end of the file since their keys are higher than any key in the file currently. The last track is filled up completely. Remaining records, in this case the record with key 22, are moved to the overflow area.



(In this example it has been assumed that records 17–22 were inserted before record 9. Otherwise, record 22 would have been placed after record 10 in the overflow area.)

For Prime track 1, the track index would contain the following information:

. Highest key on prime data track: 9.
. Address of first record on the track: address of rec. 1.
. Highest key in overflow area: 11.
. Address of lowest key in chain: address of record 10.

Figure 4.15. Addition of records to an existing, 1-cylinder. 3-track ISAM file
Any record shifted out of a track can be located by way of the track index and the overflow chain.

If a record is shifted out of a track, it still remains associated with that track. The track index contains, as was explained before, an entry for each track. This entry contains information about the prime data track as well as the overflow records that have been shifted out of this track:

- A normal entry that specifies:
  - The highest key on a prime data track
  - The address of the first record on that track.
- An overflow entry that specifies:
  - The highest key in the overflow area which is associated with the prime data track
  - The address of the lowest key in the overflow chain.

The normal entry specifies the prime data track, the overflow entry specifies all overflow records that have shifted out of that prime data track into the overflow area. All overflow records associated with a particular prime data track are chained to one another, and the overflow entry in the track index specifies the beginning of that chain. In Figure 4.15C, the records with keys 10 and 11 are chained since they come from the same prime data track. The address of the record with key 10 is given by the track index in the overflow entry. In this entry, the record with key 11 is noted as the highest key in this chain.

Types of Overflow Areas

Programmers may request two types of overflow areas:

- A cylinder overflow area for each cylinder, which provides a certain amount of tracks on each cylinder to hold the overflow records of that cylinder.
- An independent overflow area for the entire file, which provides a certain amount of tracks, independently of the file, perhaps even on another volume.

The independent overflow area can be used in addition to, or without, cylinder overflow areas. If in addition to cylinder overflow areas, it is used whenever one of the cylinder overflow areas is filled.

Structure of the Physical Blocks

All data blocks in an indexed sequential file are written with a count area, a key area, and a data area. For prime data blocks, the logical records are as defined by the user. For overflow blocks, the logical record is expanded with a link field, which is used to chain overflow blocks together. In the overflow area, logical records are written in an unblocked format, though the records may be written blocked in the prime data area. Figure 4.16 shows the various record structures that can be used.

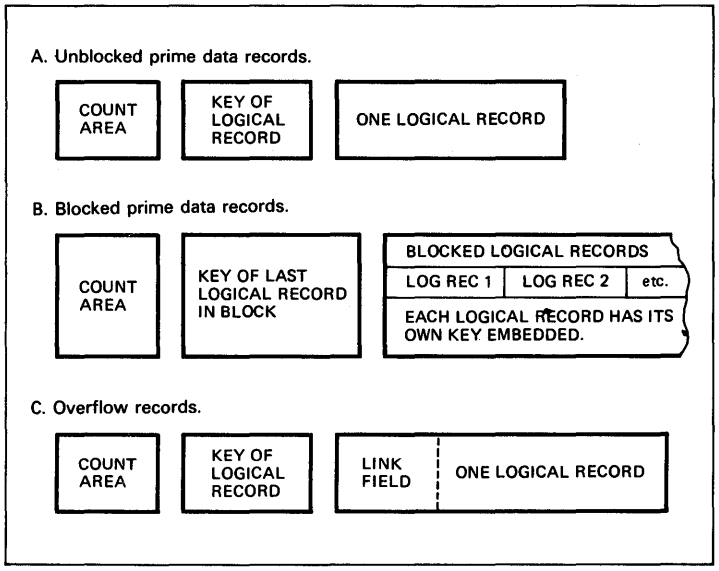


Figure 4.16. ISAM record structures
All physical blocks have three areas whose contents depend on the record type.

The count area specifies the sequence number of the physical block on the track, the length of the key area which must be constant throughout the file, and the length of the data area which also is constant except for the link field in overflow records. The link field itself is 10 characters long.

ISAM Procedures

This section describes in general how the various functions of ISAM are performed. How the user chooses and controls specific functions is described separately under *Assembler language considerations*, later in this chapter.

The functions that are described, are:

- Creating (loading) an Indexed Sequential file
- Adding new records to an existing Indexed Sequential file
- Sequential retrieval and update
- Random retrieval and update
- Reorganizing an Indexed Sequential file

Creating an ISAM File

An Indexed Sequential file is created by a load routine. The file is created according to the characteristics of the file as they are defined by the programmer in his program. It should be noted that those characteristics describe only the output ISAM file. The input may be in a quite different format and is described separately as another file, or as even more than one file. The problem program constructs an ISAM record from the input and presents this record to the ISAM load routine. Therefore, input may be provided by any access method, from any device, in any format suitable to that access method or device. The only requirement for the load routine is that the ISAM records be presented in ascending order by key.

The Format 2 DASD label used for an ISAM file contains pre-recorded information about the record format, such as record length, block length, and key length. This implies that those sizes are fixed for a specific file.

Each logical record is presented to ISAM separately. If the program requests blocking, this is done by the load routine. The load routine also creates indexes.

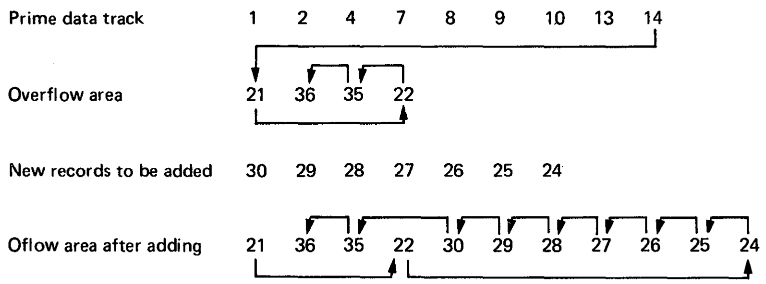
Extending an ISAM File

The same program that initially sets up an ISAM file can also be used to extend that file, provided that the key of each additional record is sequentially higher than the last current record in the file.

Adding Records to an Existing ISAM File

A separate routine is used to write new records to an ISAM file and place those records in the proper sequence. The indexes are searched to locate the proper place in the file for the new record. The block on that location is read, the new record is inserted, and all records on the same track are shifted one record location, changing records from one block to the next when necessary. The last record previously on the track is written to the overflow area. (Except for the last track in the file, which may be followed by usable free space.)

In the overflow area, all records that originally came from the same prime data track are chained in key sequence. The beginning of such a chain is addressed in the index for that prime data track, another index entry also contains the highest (last) key of the chain. When records must be added to an overflow chain, the entire chain is searched, starting with the first record, until a record is found with a key higher than the key of the new record. Therefore, when a string of new records must be added, and each new record has a next higher key, it may be advisable to insert those records in descending sequence. This keeps the update time for each new record in the string constant. Otherwise the update time for each new record is somewhat longer than for the previous one.



In order to reduce search time, the resident cylinder index operation should be used also.

Sequential Retrieval and Update

A sequential processing routine retrieves all records in ascending sequence by key, starting with a specified record somewhere in the file, and continuing to the point where the program decides to break the sequence. Then the program can choose another starting point to process another set of records sequentially by key.

When unblocked records are processed, their keys are read along with the records. These keys therefore need not be embedded in the records. When blocked records are processed, the keys are not read. In sequential processing, information regarding record, block, and key sizes is obtained from the Format 2 DASD label.

Updating is possible, since each record that has been retrieved by the sequential processing routine can, after processing, be returned to its original location in the file.

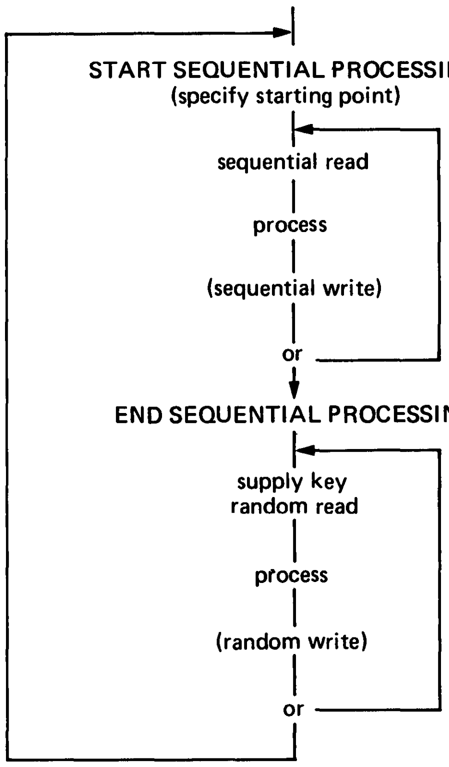
Random Retrieval and Update

Random processing of an Indexed Sequential file is done by a random processing routine. This routine accepts the key of a desired record, searches the indexes, reads the record, and presents it to the problem program. For both blocked and unblocked records, only the data portion is read; the key area is not read from the file. Information regarding record, block, and key sizes is obtained from the Format 2 DASD label.

After the retrieved record has been processed it can be returned to its original location in the file.

Mixed Functions

It is possible to perform both sequential and random processing in one problem program. For example, while processing sequentially, it may be necessary to update some other record randomly. In this case, the sequential process is delimited at its beginning by the specified starting record and at its end by an action in the program before the random process.



New records may also be added to an ISAM file in the same program that processes the file sequentially, provided that sequential processing is separated from any random addition process in a way as indicated above.

It is not possible to combine the load function (creating the file) with another function.

Reorganizing an ISAM File

As new records are added to an Indexed Sequential file, existing records are placed in overflow areas. The access time for retrieving records from an overflow area is greater than that for retrieving prime data records. This is because prime data records are located by a device search over a track, whereas overflow records are found by scanning, record-by-record, through a chain of records. Therefore, many overflow records reduce input/output performance. For this reason, the programmer should reorganize indexed sequential files as soon as he recognizes the need for it.

The system maintains statistics to assist the programmer in determining when reorganization is required. These statistics, are maintained in the Format 2 DASD label recorded with the file:

- Prime record count
  A 4-byte count of the number of records in the prime data area, in binary.
- Overflow record count
  A 2-byte count of the number of records in the overflow area, in binary.
- Available independent overflow tracks
  A 2-byte count of the number of tracks remaining in the independent overflow area (if used), in binary.

- Cylinder overflow areas full
  A 2-byte count of the number of cylinder overflow areas that are full, necessitating use of the independent overflow area, in binary.
- Nonfirst overflow reference
  A 4-byte count of the number of times a random reference (retrieve) is made to records that are the second or higher links in an overflow record chain.

The fields mentioned above are maintained automatically during processing by ISAM. In addition, there is another field that can contain statistics: *Tag Deletion Count*. This field, however is *not* maintained by the DOS/VS ISAM routines; it can be maintained by the problem program. The contents of this field is retrieved by the OPEN routines of ISAM and placed in virtual storage. After processing, the CLOSE routines of ISAM return this field to the Format 2 DASD label. Before ISAM returns the field, the programmer may use it for counting the number of records that he tags for deletion.

DOS/VS provides no facility for actually deleting records. A programmer, however, may tag records for deletion by any method desired, as long as the keys of the records are not changed in a way that alters the sequence in the file. The data portion of a record may, for example, be overwritten with character zeroes, or a special field in a record may indicate that this record is deleted. Records that are tagged for deletion can be eliminated when reorganizing the file. *If a user plans to interchange ISAM data between DOS/VS and OS/VS, it is recommended that deleted records be tagged with the hexadecimal value X'FF' in the first byte of the record.*

Reorganization is accomplished by creating a new version of the file, using the existing version as input. Two ISAM files are defined: the existing file as (sequential) input, and a new version as (load) output. As far as the system is concerned, there is no relationship between those two files. It is the problem program that establishes a relationship by reading the existing version, and loading a new version. Therefore, the records of the new version may have quite another format, if desired, provided that the problem programs that process the new version are so designed that the new format is also defined.

Depending on the capacity of the system, the reorganization may be done in one step, or in two steps. If the reorganization is done in one step, the capacity of the system must be large enough to hold both files online. Otherwise the reorganization must be done in two steps, the first of which reads the existing version in key sequence and writes it to magnetic tape, and the second of which reads the magnetic tape and creates a new version. In either of the two steps, records that are tagged for deletion may be eliminated.

Figure 4.17 indicates how the reorganization process can be accomplished, in either one or two steps. When in two steps, an intermediate file is created in step 1, and processed as input in step 2. This file is processed in both steps by means of the Sequential Access Method. It may be written on magnetic tape or on DASD (since this file need not be completely online, a single DASD or tape drive is sufficient).

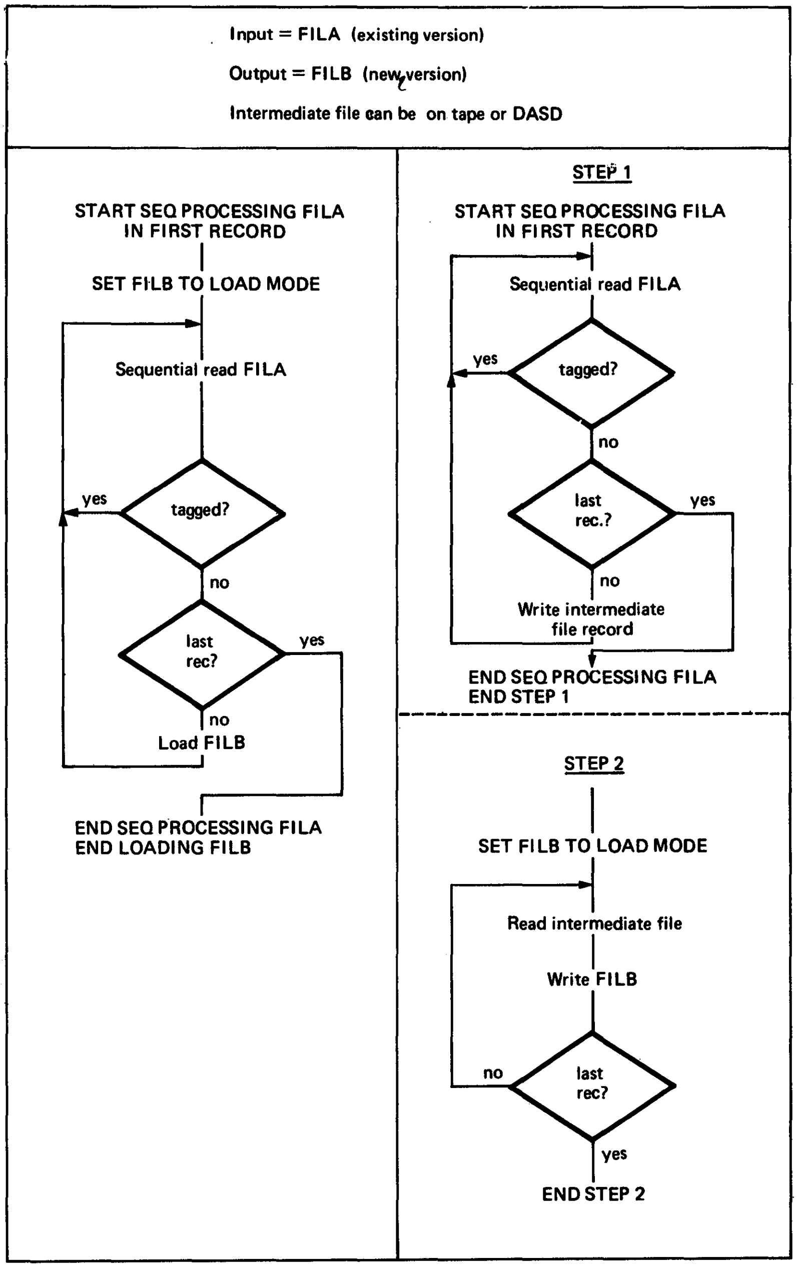


Figure 4.17. Reorganizing an ISAM file

If the online capacity of the system is not large enough to hold both FILA and FILB (left diagram), reorganization must be done in two steps (right diagram).

INDEXED SEQUENTIAL DISK STORAGE SPACE FORMULAS FOR THE IBM 2311

Three formulas compute IBM 2311 disk storage requirements for an ISAM file. The known quantities for the computations given are:

D = Data Length
K = Key Length
B = Block Length (Data Length x Number of Records)
X = Number of prime data tracks per cylinder
L = Number of bytes (10) for overflow link information.

I. TO CALCULATE THE NUMBER OF PRIME DATA RECORDS PER CYLINDER (Npr)

Let: A = Number of prime data records on a shared track
B = Number of records on a nonshared track.

(Notes: These values must be whole numbers. A shared track is one in which prime data records occupy unused space on a track index.)

The last track of the prime data area cannot be used during a load or an extension of a data file. The programmer should issue the ENDFL macro and perform a load extend on the file.

Then: a. Determine the size of the track index in bytes (T_1).

$$T_1 = [2X+1]\,[91.49+1.049(K_L)]$$

b. Determine the number of bytes remaining on a track for prime records (T_2).

$$T_2 = 3625 - T_1$$

c. Determine the size of the last prime record on a track (T_3)

$$T_3 = 20 + K_L + B_L$$

d. Determine the number of prime data records on a shared track (A).

$$T_4 = T_2 - T_3$$

if the result (T_4) is negative, set $A=0$,
if the result (T_4) is zero, set $A=1$,
if the result (T_4) is positive, set

$$A = 1 + \frac{T_4}{81+1.049(K_L+B_L)}$$

e. Determine the number of records on a non-shared track (B).

$$B = 1 + \frac{3605-(K_L+B_L)}{81+1.049\,(K_L+B_L)}$$

Compute the number of prime records per cylinder (Npr) by substituting for A, B and X in

$$Npr = A+B\,(X-1)$$

II. TO DETERMINE THE NUMBER OF OVERFLOW RECORDS PER TRACK (Nor)

Compute:

$$Nor = 1 + \frac{3605-(K_L+D_L+L)}{81+1.049\,(K_L+D_L+L)}$$

III. TO DETERMINE THE NUMBER OF CYLINDER OR MASTER INDEX RECORDS PER TRACK (Nir)

Compute:

$$Nir = 1 + \frac{3595-K_L}{91.49+1.049\,(K_L)}$$

(Note: Allow for a dummy record.)

INDEXED SEQUENTIAL DISK STORAGE SPACE FORMULAS FOR THE IBM 2314/2319

Three formulas compute IBM 2314/2319 disk storage requirements for an ISAM file. The known quantities for the computations given are:

D	=	Data Length
K	=	Key Length
B	=	Block Length (Data Length x Number of Records)
X	=	Number of prime data tracks per cylinder
L	=	Number of bytes (10) for overflow link information.

I. TO CALCULATE THE NUMBER OF PRIME DATA RECORDS PER CYLINDER (Npr)

Let: Let: A = Number of prime data records on a shared track
B = Number of records on a nonshared track.

(Notes: These values must be whole numbers. A shared track is one in which prime data records occupy unused space on a track index.)

The last track of the prime data area cannot be used during a load or an extension of a data file. The programmer should issue the ENDFL macro and perform a load extend on the file.

Then: a. Determine the size of the track index in bytes (T_1),

$$T_1 = [2X+1]\,[156.43+1.043\,(K_L)]$$

b. Determine the number of bytes remaining on a track for prime records (T_2),

$$T_2 = 7294 - T_1$$

c. Determine the size of the last prime record on a track (T_3),

$$T_3 = 45+K_L+B_L$$

d. Determine the number of prime data records on a shared track (A),

$$T_4 = T_2 - T_3$$

if the result (T_4) is negative, set A = 0,

if the result (T_4) is zero, set A = 1,

if the result (T_4) is positive, set

$$A = 1 + \frac{T_4}{146+1.043\,(K_L+B_L)}$$

e. Determine the number of records on a non-shared track (B),

$$B = 1 + \frac{7249-(K_L+B_L)}{146+1.043\,(K_L+B_L)}$$

Compute the number of prime records per cylinder (Npr) by substituting for A, B and X in.

$$Npr = A+B\,(X-1)$$

II. TO DETERMINE THE NUMBER OF OVERFLOW RECORDS PER TRACK (Nor)

Compute:

$$Nor = 1 + \frac{7249-(K_L+D_L+L)}{146+1.043\,(K_L+D_L+L)}$$

III. TO DETERMINE THE NUMBER OF CYLINDER OR MASTER INDEX RECORDS PER TRACK (Nir)

Compute:

$$Nir = 1 + \frac{7239-K_L}{156.43+1.043\,(K_L)}$$

(Note: Allow for a dummy record.)

INDEXED SEQUENTIAL DISK STORAGE SPACE FORMULAS FOR THE IBM 3330 FAMILY

Three formulas compute IBM 3330 disk storage requirements for an ISAM file. The known quantities for the computations given are·

D = Data Length
K = Key Length
c = constant: $c = 0$, if $K = 0$
$c = 56$, if $K \neq 0$.
B = Block Length (data length x number of records)
X = Number of prime data tracks per cylinder
L = Number of bytes (10) for overflow link information.

I. TO CALCULATE THE NUMBER OF PRIME DATA RECORDS PER CYLINDER (Npr)

Let: A = Number of prime data records on a shared track
C = Number of records on a non-shared track.

(Notes: These values must be whole numbers. A shared track is one in which prime data records occupy unused space on a track index.)

The last track of the prime data area is never used for prime data records. If the file is completely filled, the last track contains an EOF record only.

Then: a. Determine the size of the track index in bytes (T1),
$T1=(2X+1)(135+c+K)$

b. Determine the number of bytes remaining on the track for prime records (T2),
$T2=13{,}165-T1$

c. Determine the number of prime data records on a shared track (A),
if T2 is negative, set $A=0$,
if T2 is zero, set $A = 1$,
if T2 is positive, set $A= 1+\frac{T2}{135+c+K+B}$

d. Determine the number of records on a non-shared track (C),
$C= 1+\frac{13{,}165-(K+B)}{135+c+K+B}$

Compute the number of prime records per cylinder (Npr) by substituting for A, C, and X in
$Npr = A+C(X-1)$

II. TO DETERMINE THE NUMBER OF OVERFLOW RECORDS PER TRACK (Nor)

Compute: $Nor= 1+\frac{13{,}165 - (K+D+L)}{135+c+K+D+L}$

III. TO DETERMINE THE NUMBER OF CYLINDER OR MASTER INDEX RECORDS PER TRACK (Nir)

Compute: $Nir= 1+\frac{13{,}165 - (K+10)}{135+c+(K+10)}$

INDEXED SEQUENTIAL DISK STORAGE SPACE FORMULAS FOR THE IBM 3340

Three formulas compute IBM 3340 disk storage requirements for an ISAM file. The known quantities for the computations given are:

D = Data Length
K = Key Length
c = constant $c = 0$, if $K = 0$
$c = 75$ if $K \neq 0$.

B = Block Length (data length x number of records)
X = Number of prime data tracks, that is, shared and non-shared data tracks, per cylinder.
L = Number of bytes (1∅) for overflow link information,

I. TO CALCULATE THE NUMBER OF PRIME DATA RECORDS PER CYLINDER (Npr)

Let: A = Number of prime data records on a shared track
C = Number of records on a non-shared track.

(Notes: These values must be whole numbers. A shared track is one in which prime data records occupy unused space on a track index.)

The last track of the whole prime data area is never used for prime data records. If the file is completely filled, the last track contains an EOF record only.

Then: a. Determine the size of the track index in bytes (T1),
$T1 = (2X + 1) \leftarrow (167 + c + K)$

If $T1 \leqslant 8535$ the track index requires at most one track. Otherwise track ∅ and part of track 1 is required. The size T1' of the track index on track 1 is calculated as follows:

a1. Number of track index entries on track ∅:

$$N = \frac{8535}{167 + c + K}$$ N rounded to next whole number smaller or equal.

a2. $T1' = T1 - N(167 + c + K)$

For $T1 > 8535$, use T1' instead of T1 in b.

b. Determine the number of bytes remaining on the track for prime records (T2),
$T2 = 8535 - T1$

c. Determine the number of prime data records on a shared track (A),
if T2 is zero, set A = ∅,
if T2 is positive, set $A = \frac{T2}{167 + c + K + B}$

d. Determine the number of records on a non-shared track (C),

$$C = \frac{8535}{167 + c + K + B}$$

Compute the number of prime records per cylinder (Npr) by substituting for A, C, and X in

$Npr = A + C(X - 1)$ in case of shared data tracks ($A \neq \emptyset$)
$Npr = CX$ in case of non-shared data tracks ($A = \emptyset$)

II. TO DETERMINE THE NUMBER OF OVERFLOW RECORDS PER TRACK (Nor)

Compute: $$Nor = \frac{8535}{167 + c + K + D + L}$$

III. TO DETERMINE THE NUMBER OF CYLINDER OR MASTER INDEX RECORDS PER TRACK (Nir)

Compute: $$Nir = \frac{8535}{167 + c + (K + L)}$$

Assembler Language Considerations

Macro Instructions for Indexed Sequential Processing (Figure 4.18)

Both GET-PUT and READ-WRITE input/output macro instructions are provided for indexed sequential processing. The purpose and the effect of the macro instructions vary, depending on the logic modules used and the conditions preset by other macro instructions.

The following macro instructions are provided:

DTFIS	Define The File for Indexed Sequential
GET	Obtain a record (sequential retrieval)
PUT	Write a record (sequential update)
READ	Read a record (random retrieval)
WRITE	Write a record (load, add, or random update)
WAITF	Wait for completion of a READ or WRITE
SETL	SET Lower limit for sequential retrieval
ESETL	End of sequential retrieval
SETFL	SET file load mode
ENDFL	End file load mode

These macro instructions are explained in the following text in the context of a particular function.

Creating an ISAM File
The DTFIS macro instruction calls for a load routine. Before actual loading begins, a SETFL macro instruction is issued to initialize the indexes. Each record is presented to the load routine separately by means of a WRITE macro instructions. When all records have been loaded, the loading process is terminated by an ENDFL macro instructions. This instruction writes the last block of records, followed by an end-of-file record. The indexes are completed with any end-of-file record needed, and dummy index entries are written for the rest of the specified prime data extent.

Adding Records to an ISAM File
New records are added to an existing indexed sequential file by means of the WRITE macro instruction. This process is not preceded by a SETFL macro instruction. A WAITF macro instruction is used at the point in the program where processing must be suspended until the WRITE macro function has been completed (for example, before the next WRITE).

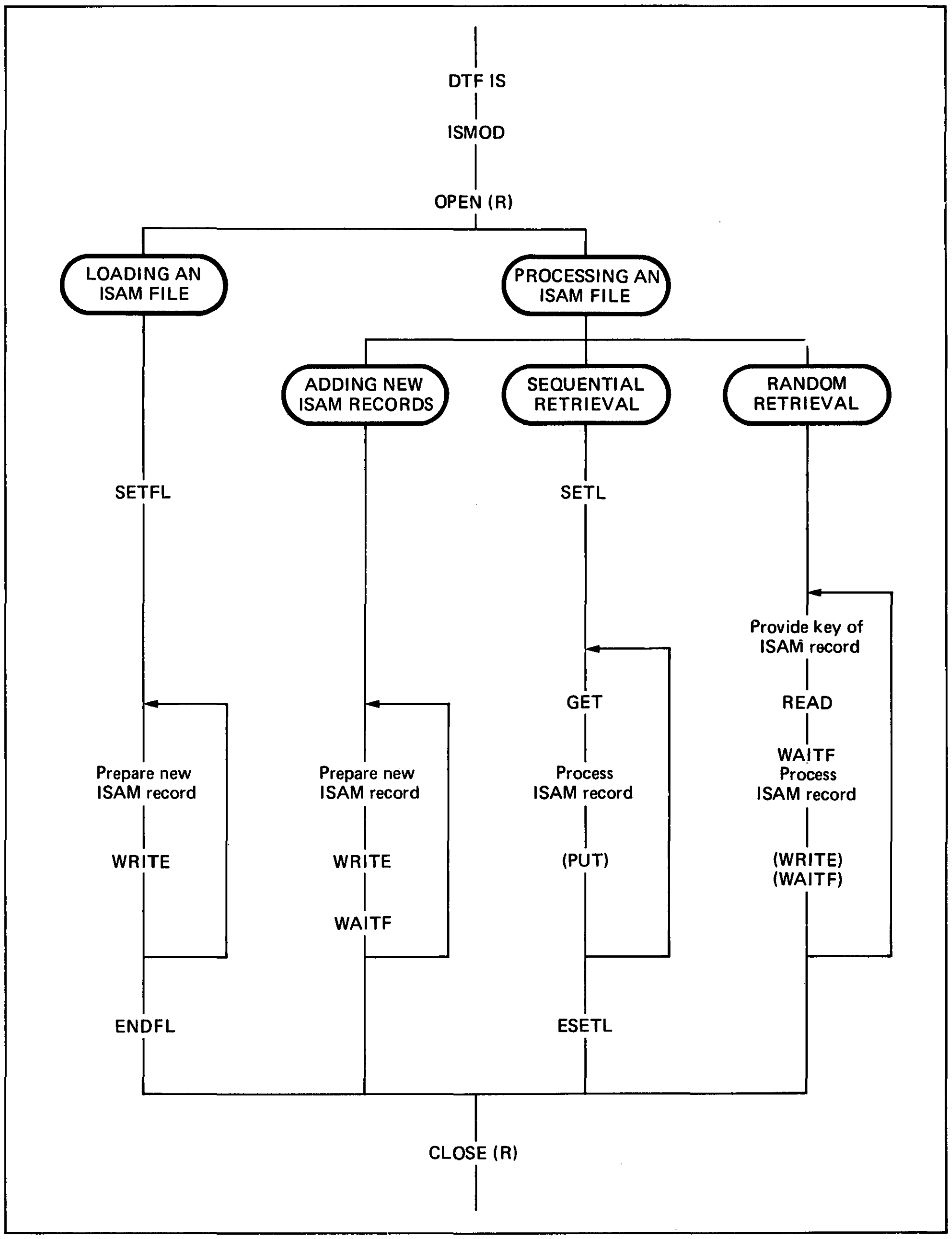


Figure 4.18. ISAM functions, and how to control them in Assembler language
Sequential retrieval and random retrieval functions are used in this example for updating records.

Sequential Retrieval and Update

The SETL macro instruction specifies the location of the record that must be retrieved first, as a starting location. Following records are retrieved in logical sequential order (sequentially by key) by means of a GET macro instruction. After processing a record obtained by a GET macro instruction, the programmer can issue a PUT macro instruction to restore the record to its original location. Obviously this is not needed for records that have not been changed. The complete block is written back if, and only if, a PUT is issued for any record in the block. The ESETL macro instruction terminates the sequential processing of a string of records.

Processing may be resumed at another starting point in the file by issuing another SETL.

When processing an indexed sequential file sequentially, new records can be added to this file, provided that the WRITE macro instruction that performs this function is preceded by a ESETL macro instruction. After the addition of one or more records, sequential processing can be resumed by a new SETL macro instruction.

Random Retrieval and Update

For reading a record, the programmer places the key of the desired record in a special field, and issues a READ macro instruction. If it is in the file, the record is then presented to the program. After the record has been processed, a WRITE macro instruction may be issued to restore the record to its original location. In order to allow overlap between processing and input/output, the READ and WRITE macro instructions return to the problem program before input/output has been completed. The WAITF macro instruction must be used to suspend processing until the I/O operation is complete.

Logic Modules for ISAM Processing

The LIOCS routines provided for indexed sequential file processing are much more than just an access method. Several routines are available to provide complete file management for direct access storage files. The complete facility is called the Indexed Sequential Access Method. The ISAM routines can be retireved from the relocatable library by the linkage editor just like the SAM routines. The user must assembler the routines before placing them in the relocatable library. This is normally a one-time operation, performed as part of normal total system generation. The routines generated are tailored to provide specific functions but remain generalized in regard to specific file and data attributes. Four basic types of routines are available:

- LOAD
  To load (create) a new indexed sequential file.
- ADD
  To add new records to an existing indexed sequential file in logical sequence.
- SEQUENTIAL RETRIEVAL
  To retrieve records in logical sequence from an indexed sequential file.
- RANDOM RETRIEVAL
  To retrieve individual records called for by key from any point in the file.

The load routine is always separate. No other functions can be performed on an output file that is being loaded. In other situations, the add and retrieve functions can be used in any combination. Furthermore, the retrieval routines are assembled with updating capability, allowing records to be written back into their original location in the file.

The assembled routines of ISAM are called logic modules. They are selected from a master source routine in accordance with parameters in a special macro instruction: ISMOD (Indexed Sequential MODule). The assembled modules are completely file-independent and can be used for all indexed sequential files. If desired, the logic modules can be assembled along with the user's problem program and included in the output object module.

For each indexed sequential file to be processed, a program must include a DTFIS macro instruction. Some of the fields within the DTF table generated are not specified until the file is opened during execution of the object program. Many of the fields are kept with the file in a special format (Format 2) of the standard DASD file label.

In addition to the parameters that describe the file to be processed, the DTFIS macro instruction includes certain parameters identical to those used in the ISMOD macro instruction.

If the DTFIS and ISMOD macro instruction specify that two I/O areas are used, overlap of the physical transfer of data with processing can be achieved for load and sequential retrieval.

In addition to the macro instruction mentioned in the preceding section, the CLOSE(R) and OPEN(R) macro instructions are used to connect the indexed sequential file to the program, to process labels, and to fill the DTF table with items from the Format 2 DASD label.

Virtual Storage Access Method (VSAM)

Devices and Record Formats

The Virtual Storage Access Method (VSAM) supports the following devices:

- IBM 2314 Direct Access Storage Facility
- IBM 2319 Disk Storage
- IBM 3330 Family of Disk Storage Devices
- IBM 3340 Disk Storage

The equipment is described in Appendix 1.

Fixed-length or variable-length records can be processed by VSAM. Record blocking is completely controlled by VSAM, and so the user is not concerned with whether records are blocked or unblocked. The user specifies only the record size and, under some conditions, the size of a logical unit of a VSAM file called a control interval.

VSAM File Structures

VSAM has key-sequenced and entry-sequenced files. The primary difference between the two is the sequence in which data records are stored. This section explains the logical organization of the two types of VSAM files. It also explains the physical organization of VSAM files on direct-access volumes.

Records are stored in a *key-sequenced file* in the collating sequence of a key field, such as employee number or invoice number. Each record must have a unique value in its key field. VSAM uses an index to access records in a key-sequenced file. VSAM also allows free space to be distributed throughout the file so records can be inserted physically into the file. Therefore, separate overflow chains are not needed.

Records are stored in an *entry-sequenced file* in the physical sequence in which they are entered (loaded). New records are stored at the end of the file and records cannot be physically deleted or changed in length.

VSAM stores the records of each type of file in a logical unit called a control interval. Control intervals are grouped together in a logical unit called a control area. The following section describes how and why VSAM uses control intervals and control areas for storing data records. The index of a key-sequenced file is stored separately as described later.

Control Intervals and Control Areas

A *control interval* is a continuous area of direct access storage in which VSAM stores data records and control information describing them. It is the unit of a file that VSAM transfers to and from direct access storage and contains one or more physical blocks.

The size of a control interval can vary from one file to another, but each control interval in a given file is of the same size. It is chosen either by VSAM or by the user (within limits acceptable to VSAM). VSAM chooses the size that is best for (1) the type of direct access device used to store the file, (2) the length of the data records, and (3) the least amount of virtual storage space the processing program will provide for VSAM's I/O buffers.

A control interval contains an integral number of physical blocks. VSAM chooses the physical block size. It will be 512 bytes, 1024 bytes, 2048 bytes, 4096 bytes (3330 and 3340 only), or 8192 bytes (3340 only). Therefore, the control interval size has to be a multiple of 512 bytes. If the control interval size is larger than 8192 bytes, it must be a multiple of 2048 bytes. The maximum control interval size is 32,768 bytes.

A control interval is independent of types of direct access devices, though its size is chosen for the device on which it was created. As Figure 4.19 illustrates, a control interval that fits on a track of one type of device might extend across tracks if the file is moved to another type of device.

The records of a file are stored in control intervals; a group of control intervals makes up a *control area.* Each control area in the file has the same number of control intervals. For a key-sequenced file, the number is equal to the number of index entries in an index record in the lowest level of the index. If 50 were the number chosen, for example, the first 50 control intervals would be the first control area; the next 50 would be the second control area, and so on. Control areas are also used for distributing free space throughout a key-sequenced file, as a percentage of free control intervals per control area. The section *Key-Sequenced Files* describes the relationship of control areas to the index and free space of a key-sequenced file in more detail.

A control area is the amount of direct-access space that VSAM may preformat for data integrity as records are added to the end (loaded) of either a key-sequenced or entry-sequenced file. Whenever the space for a file is extended, it is extended by an integral number of control areas. (See the section *Data Security and Integrity.*)

A control area always occupies an integral number of tracks. It has normally the size of a cylinder on the device which contains the file, and starts on a cylinder boundary. It will never be larger than a cylinder and will be smaller only if (1) the user allocated space for the file in terms of records or tracks, and (2) VSAM could not set the control area size to one cylinder and still select a control interval size that met all requirements. If the control area is smaller than a cylinder, it can cross cylinder boundaries but will not cross extent boundaries.

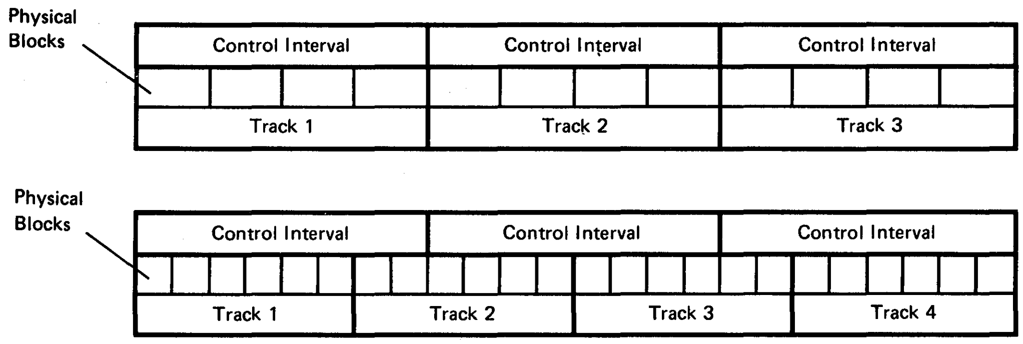


Figure 4.19. Control intervals are independent of physical block size

The Method of Storing a Record in a Control Interval

VSAM stores fixed length and variable length records in the same way. It puts control information at the end of a control interval to describe the data records stored in that control interval. The combination of a data record and its control information, though they are not physically adjacent, is called a *stored record.* Figure 4.20 shows how data records and control information are stored in a control interval.

Stored records do not extend across control intervals. When the user defines a file, he must specify enough buffer space so that the control intervals are large enough for the largest records. The maximum logical record size is 32,761 bytes.

A data record is addressed by its displacement, in bytes, from the beginning of the file. This displacement is the Relative Byte Address *(RBA)* of the record. The RBA does not depend on the location (cylinder and track) of the record on a direct-access volume. For relative byte addressing, VSAM considers the control intervals in the file to be contiguous, as though the file were stored in virtual storage beginning at address 0. In calculating RBAs, VSAM considers free space and control information in the control intervals, as well as the records themselves, as part of the address space.

Physical Organization

How do control intervals and control areas relate to the physical attributes of direct access storage?

A volume need not be assigned exclusively to VSAM. System files and files of other access methods can also be on the volume. An area to be used exclusively by VSAM is called a *data space* and is defined in the VTOC through DLBL and EXTENT statements. It can include up to 16 *extents* that need not be adjacent to one another on the volume.

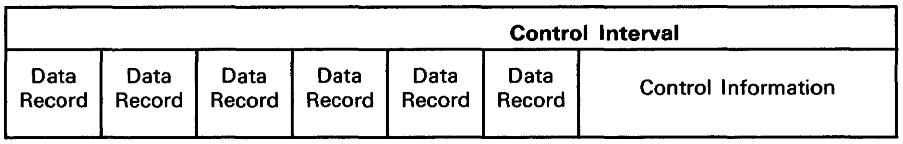
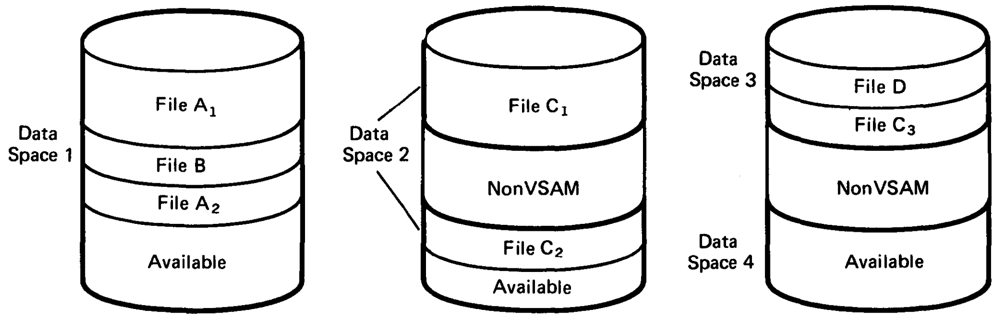


Figure 4.20. How data and control information is stored in a control interval
Data records are stored in the front of a control interval; control information in the back.



NOTE: A_1, C_1, A_2, etc. are extents of the files.

Figure 4.21. Volumes, VSAM data spaces, VSAM files, and nonVSAM files
Portions of files A and C are stored in different data spaces on different volumes.

A file is stored in a data space or data spaces on one or more direct-access volumes. All the volumes of a file must reside on the same device type, but the index of a key-sequenced file can be on a different device type than the data of the file. When a file is defined, space can be allocated at the end of the file for additional records. Otherwise, when additional space is needed, VSAM automatically extends the file if unused data space is available. The amount of space for extension is defined by the user. The file can be extended beyond its original size to include up to 123 logical extents (data spaces or portions of data spaces) or to a maximum size of 2^{32} (approximately 4.3 billion) bytes. Figure 4.21 illustrates the relationships among volumes, data spaces, and files.

Key-Sequenced Files and Entry-Sequenced Files

The purpose of this section is to describe VSAM's two types of files in detail and to explain further how VSAM uses the control interval for data storage. Figure 4.22 contrasts the two types of files by listing the attributes of each.

Key-Sequenced File	Entry-Sequenced File
Records are in collating sequence by key field	Records are in the order in which they are entered
Access is by key through an index or by RBA	Access is by RBA
A record's RBA can change	A record's RBA cannot change
Distributed free space is used to insert records and change their length in place	Space at the end of the data is used for adding records; their length cannot be changed in place
Space given up by a deleted or shortened record is reused by new or lengthened records in the same control interval	A record cannot be deleted, but its space can be reused for a record of the same length

Figure 4.22. Differences between, and characteristics of, key-sequenced files and entry-sequenced files

Key-sequenced and entry-sequenced files differ in the use of an index and free space and in the changeability of RBAs.

Key-Sequenced Files

The most distinctive features of a key-sequenced file are its index and its distributed free space. In discussing them we can cover all the important points about this type of file.

The Index for a Key-Sequenced File.
A key-sequenced file has an index which relates key values to the locations of the data records in a file. A key in the index is taken from a record's key field, and whose value cannot be altered. VSAM uses an index to locate a record for retrieval, deletion, or update and to locate the collating position for insertion of a new record.

An index has one or more levels. Each level is a set of records that contain *index entries* giving the location of the records in the next lower index level. The index records in the lowest level are collectively called the *sequence set*; their entries give the location of control intervals containing the data records. The records in all the higher levels are collectively called the *index set*; their entries give the location of lower level index records. The highest level always has only a single record. The index of a file with few enough data control intervals for a single sequence-set record has only one level: the sequence set itself. The index record (also called an index control interval) is the size of a physical block. Thus it may be 512 bytes, 1024 bytes, 2048 bytes, 4096 bytes (3330 and 3340 only), or 8192 bytes (3340 only).

Figure 4.23 illustrates the levels of an index and shows the relationship between a sequence-set index record and a control area.

An entry in an index-set record consists of the highest key that an index record in the next lower level contains, paired with a pointer to that index record. An entry in a sequence-set record consists of the highest key a data control interval will contain, paired with a pointer to that control interval. The group of control intervals pointed to by all the entries in a single sequence-set record is a control area. Not all data records have sequence-set entries, for there is only one entry for each control interval in the file.

For direct processing by key, VSAM follows *vertical pointers* from the highest index level down to the sequence set to find a pointer to the control interval containing the data record. For sequential access by key, VSAM usually refers only to the sequence set. It uses a *horizontal pointer* in a sequence-set record to get from that sequence-set record to the one containing the next key in collating sequence in order to find a pointer to the control interval containing the data record. Figure 4.23 shows both vertical pointers and horizontal pointers.

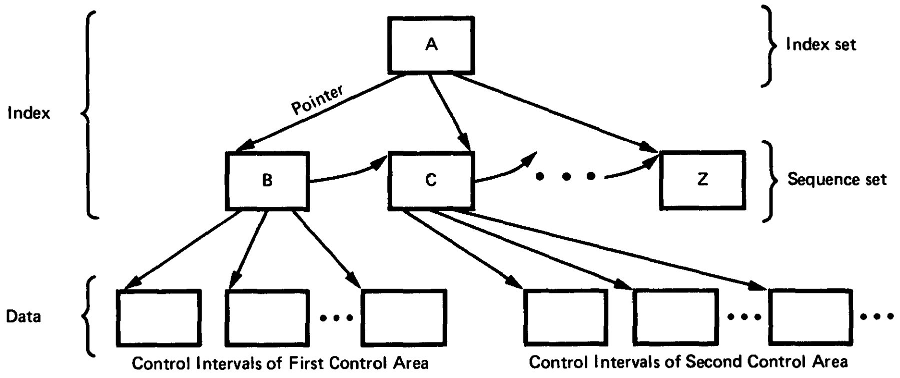

Index
Data
A
Pointer
Index set
B
C
Z
Sequence set
Control Intervals of First Control Area
Control Intervals of Second Control Area

Figure 4.23. VSAM's index structure for key-sequenced files
The highest level index record (A) controls the entire next level (B through Z); each sequence-set index record controls a control area.

VSAM increases the number of entries that an index record can hold by *key compression.* It eliminates from the front and the back of a key those characters that are not necessary to distinguish it from the adjacent keys. Compression helps achieve a physically smaller index by reducing the size of keys in index entries. For example, a two-level index, the size of whose records is 2048 bytes with a key field of 10 bytes, and the size of whose entries (including compressed key and pointer) is 8 bytes on the average, can control approximately 62,500 control intervals, each of which may contain numerous data records.

Index entries vary in size because of key compression. Thus a binary search of the index record cannot be made. Therefore, the entries in an index record are grouped into sections, the number of sections depending on the number of entries in a record. A binary search is made of the high key of each section. The entries of the desired section are then searched for the desired key.

The number of control intervals in a control area equals the number of entries in a sequence-set index record. This equality has important uses in:

- Placing the sequence-set index record adjacent to the control area on a single cylinder (see the section *Optimizing the Performance and Storage of VSAM*).
- Distributing free space throughout a file as a percent of free control intervals in each control area.

Distributed Free Space for File Growth.
When the user defines a key-sequenced file, he can specify that free space is to be distributed (1) by leaving some space at the end of all the used control intervals and (2) by leaving some control intervals in each control area completely empty. The amount of free space in a used control interval and the number of free control intervals in a control area are independent of each other. The user can specify that the file is to contain one type of free space or both. Figure 4.24 shows how free space might be set aside in each control area. The sequence-set record for a control area contains an entry for each free control interval as well as an entry for each control interval that contains data.

When records in a key-sequenced file are shortened or deleted, their space is automatically reclaimed by VSAM and added to the free space of the control interval.

Reclaiming space and using distributed free space may cause RBAs of some records to change. As Figure 4.24 illustrates, free space within a used control interval is between the data in the front (low RBA) and the control information in the back (high RBA). If a record is deleted or shortened, any succeeding records in the control interval are moved to the left and their RBAs are changed so that the space vacated can be combined with the free space already in the control interval. Conversely, if a record is inserted or lengthened, any succeeding records in the control interval are moved to the right into free space and their RBAs are changed. However, the index entries in the sequence set and the index set do not change because the sequence-set entries point to control intervals and not to each record. This keeps the records physically in key sequence within the control interval.

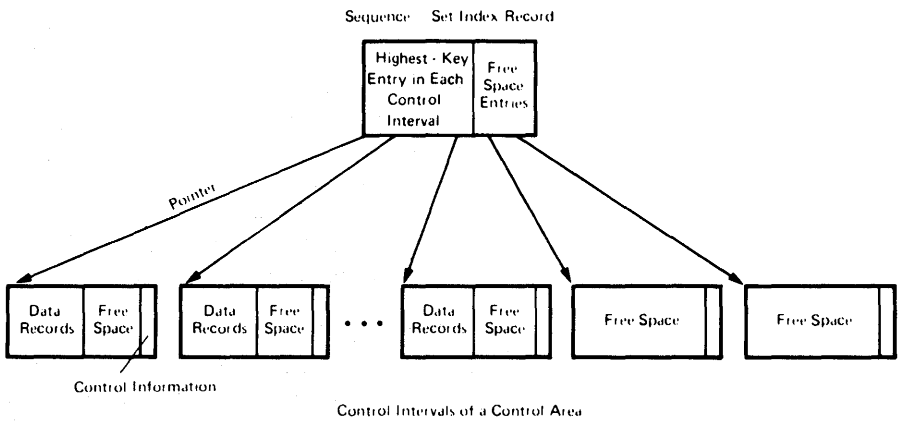


Figure 4.24. Distributed free space in a key-sequenced VSAM file

There are two kinds of distributed free space; space left in used control intervals and empty control intervals.

The preceding discussion assumes that there is enough free space in the control interval for a new record or a lengthened record. If the record to be inserted will not fit in the control interval, there is a *control interval split.* VSAM moves approximately half of the stored records in the control interval to an empty control interval in the same control area, and inserts the new record in its proper key sequence.

Figure 4.25 illustrates a control interval split and shows the resulting free space available in the two affected control intervals. Because the number of records in the first control interval is reduced, subsequent new or lengthened records will occupy the newly created free space in the two control intervals.

If the control intervals involved in a split are not adjacent, the physical sequence of data records is no longer the same as their key sequence. In Figure 4.25, the physical sequence of the records in the last three control intervals on the right is: 58, 59, 60, 61, 55, 56, 57. But the sequence set index record reflects the key sequence, so that, for keyed sequential requests, the data records are retrieved in the order: 55, 56, 57, 58, 59, 60, 61.

Should there not be a free control interval in the control area, an insertion requiring a free control interval causes a *control area split.* VSAM establishes a new control area at the end of the file, either by using space already allocated or by extending the file (if the user provided for extensions when the file was defined). VSAM moves the contents of approximately half of the control intervals in the full control area to free control intervals in the new control area and inserts a new record into one of the two control areas, as its key dictates. Since about half of the control intervals of each of these control areas are now free, subsequent insertions will not require control area splitting. Splitting should be an infrequent occurrence for files with sufficient distributed free space; splitting a control area does make it possible, however, to insert records into a key-sequenced file without previously distributed free space.

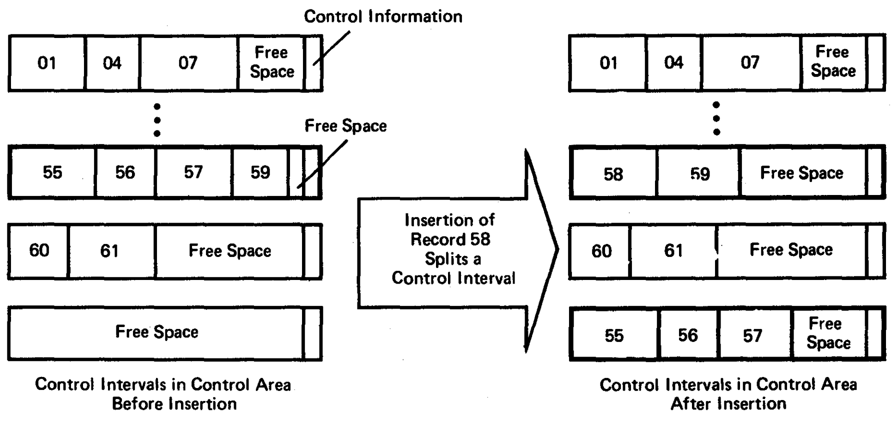

Control Information
01
04
07
Free Space
Free Space
55
56
57
59
60
61
Free Space
Free Space
Control Intervals in Control Area Before Insertion
Insertion of Record 58 Splits a Control Interval
01
04
07
Free Space
58
59
Free Space
60
61
Free Space
55
56
57
Free Space
Control Intervals in Control Area After Insertion

Figure 4.25. Example of a control interval split

A control-interval split: new record is inserted in key sequence and some of the records in the control interval that is too full for insertion are moved to the free control interval.

Entry-Sequenced Files

The records in an entry-sequenced file are stored in the physical sequence in which they are loaded. New records are added to the end of the file and existing records cannot be physically deleted or changed in length. An entry-sequenced file does not have an index. When a record is loaded or subsequently added, VSAM indicates its RBA to the user. He must keep track of the RBAs of the records to gain access to them directly.

Sequential processing with an entry-sequenced file is similar to that of SAM with direct access devices. Direct processing is accomplished by supplying the RBA of a record to be processed.

Direct processing with an entry-sequenced file can also be done in a way similar to DAM by loading the file with dummy records (filled with blanks, for instance). To store a record initially, the user determines an RBA by using an application-oriented algorithm, retrieves the dummy record at that RBA, and stores the new record back at that RBA. The user's algorithm must have a procedure for determining an alternate RBA when two or more keys are converted to the same RBA. To retrieve a record, the user applies the same algorithm to determine the RBA.

VSAM Processing Procedures

This section describes the procedures used to create and process VSAM files.

Creating VSAM Files

There are two or three steps to creating a VSAM file, depending on whether the first two steps are combined or not. The first step is to allocate an area of direct access storage to VSAM. This area, called a VSAM data space, is owned by VSAM and is available for one or more VSAM files that will be created later. A data space consists of one or more extents on a volume and is described in the VTOC as well as in the VSAM catalog. The user allocates VSAM data spaces through job control and Access Method Services. No user-written routines are required.

The second step is to sub-allocate a VSAM data space to a file, and to enter information about the file's characteristics in the VSAM catalog. This is called *defining* the file and is also done through job control and Access Method Services.

The first and second steps can be combined; the data space and the file can be *defined* at the same time. In this case, the file is called Unique and occupies the data space created with it. No other files can occupy that space. A Unique file, like those of other access methods, occupies the direct-access extents specified in the VTOC. However, unlike other files, Unique files cannot be allocated new extents later.

The third step is to load records into the file. This can be done by using Access Method Services to copy an ISAM or SAM file or by writing the records with the user's processing program.

The section *VSAM Catalog, Service Programs, and Job Control* has more information on how to create VSAM files. The format of Access Method Services commands and examples of their use are in *DOS/VS Utilities: Access Method Services*, GC33-5382.

Processing VSAM Files

For a key-sequenced file, the primary form of access is keyed access, using the index; for an entry-sequenced file, the only form of access is addressed access, using the RBA determined for a record when it was stored in the file. Addressed access can also be used to process a key-sequenced file, but previous keyed insertion, deletion, or update can change the RBAs of records. Therefore, the user may have to keep track of RBA changes if he wants to use addressed access. (VSAM passes back the RBA of each record retrieved, updated, added, or deleted.)

VSAM allows both sequential and direct processing for each of its two types of files. Sequential processing of a record depends on the position, with respect to the key or the address, of the previously processed record; direct processing does not. With sequential access, records retrieved by key are in key sequence; records retrieved by address are in entry (ascending RBA) sequence. To retrieve or store records after initial position, the user does not need to specify a key or an RBA. VSAM automatically retrieves or stores the next record in order - either next in key sequence or next in entry sequence, depending on whether processing is by key or by address. In sequential processing, initial positioning must be established by (1) pointing to the desired record, (2) inserting a record into the file (keyed access only), or (3) using direct processing to retrieve a record for update.

With direct processing, the retrieval or storage of a record is not dependent on the key or the address of any previously retrieved record. The record to be retrieved or stored must be identified by key or by RBA.

VSAM allows a processing program or its subtasks to process a file with multiple concurrent sequential and/or direct requests, each requiring that VSAM keep track of a position in the file, with a single opening of the file. Access can be to the same part of the file or to different parts.

For processing a subset of records in sequence in a key-sequenced file, the processing program can specify *skip sequential processing*. When the program indicates the key of the next record to be retrieved, VSAM skips to its index entry by using horizontal pointers in the sequence set to get to the appropriate sequence-set index record to scan its entries. However, the key of the next record must always be higher in sequence than the key of the last record.

When the processing program retrieves a record, VSAM reads the entire control interval in which it is stored. VSAM deblocks the records. VSAM places the record in the processing program's work area (move mode) or leaves the record in VSAM's I/O buffer and provides a fullword pointer to the record in the work area (locate mode). VSAM indicates the length of the record in both move mode and locate mode. The user need not be concerned with any physical attributes of stored records.

VSAM provides programmers of utilities and systems with *control interval access.* They retrieve and store the contents of a control interval, rather than a single record, by specifying control interval access in the macros and (for direct processing) giving the RBA of the control interval. They are responsible for maintaining the control information at the back of the control interval. The format of this information may change in future releases of VSAM. Figure 4.26 summarizes the types of access for VSAM files.

Type of File	Type of Access	Type of Processing	Retrieve Records	Add Records	Update Records	Delete Records
key sequenced	keyed	sequential	yes	yes	yes	yes
		skip sequential	yes	yes	yes	yes
		direct	yes	yes	yes	yes
	addr	sequential	yes	no	yes*	yes
		direct	yes	no	yes*	yes
entry sequenced	addr	sequential	yes	to end	yes*	no
		direct	yes	to end	yes*	no

*** The length of the records cannot be changed.**

Figure 4.26. Summary of Access to VSAM Files

Keyed Access for Key-Sequenced Files
Keyed access is only for a key-sequenced file. An entry-sequenced file has no index and thus cannot be processed by keyed access.

Keyed access provides for retrieval, update (including lengthening or shortening a record, as well as altering its contents except for the key), insertion, addition, and deletion. Each of these actions can be sequential, skip sequential, or direct.

Keyed Retrieval
Keyed-sequential access depends on where the previous macro request positioned VSAM with respect to the key sequence defined by the index. When the processing program opens the file for keyed access, VSAM is positioned at the first record in the file in key sequence to begin keyed-sequential processing. The POINT macro instruction positions VSAM for keyed-sequential processing at the record whose key is specified. If the key specified is a generic key (leading portion of the key field), the record positioned to is the first of the records having the same generic key. A subsequent GET macro retrieves the record VSAM is positioned at. The GET then positions VSAM at the next record in key sequence. The POINT macro can position either forward or backward in the file.

With keyed direct processing, the user can optionally specify that GET keep VSAM positioned at the next record in key sequence. His program can then process the following records sequentially.

Keyed direct retrieval does not use previous positioning; VSAM searches the index from the highest level down to the sequence set to retrieve a record. The record to be retrieved can be specified by:

- The exact key of the record
- An approximate key less than or equal to the key field of the record
- A leading portion of the key, or *generic key*, of the record.

Approximate specification can be used when the exact key is unknown. If a record actually has the key specified, VSAM retrieves it; otherwise, it retrieves the record with the next higher key. Generic-key specification for direct processing causes VSAM to retrieve the first record having that generic key. If all the records with the generic key are to be retrieved, the processing program should shift to sequential access to retrieve the rest of the records.

Keyed Deletion.
An ERASE macro instruction following a GET for update deletes the record that the GET retrieved. A record is physically erased in the file when it is deleted. The space the record occupied is then available as free space.

Keyed Storage.
A PUT macro instruction stores a record. A PUT for update following a GET for update stores the record that the GET retrieved. To update a record, the user must previously have retrieved it for update.

When VSAM detects sequential insertion of two or more records in sequence into a collating position in a file, VSAM uses a technique called *mass sequential insertion* to buffer the records being inserted. This reduces I/O operations. Using sequential instead of direct processing to insert two or more records in sequence between two records in a file enables the user to take advantage of this technique. The file can be extended (loading can be resumed) by using mass sequential insertion to add records beyond the highest key. In this case, the percentage of free space specified when the file was defined is maintained in the new control intervals.

Skip sequential processing or direct processing can be used to store records in sequence throughout a file. With skip sequential processing, VSAM skips to the next collating position by scanning the sequence set of the index; with direct processing, it finds the next collating position by searching the index from top to bottom.

VSAM uses free space for efficient insertion and lengthening of records in a key-sequenced file and automatically combines the space that is given up by deletion or shortening of records with any free space already in the affected control interval. If there is not enough free space, a control interval or control area split occurs.

Addressed Access for Both Types of Files
Addressed access can be either sequential or direct with key-sequenced and entry-sequenced files, but the processing allowed for a key-sequenced file is different from that allowed for an entry-sequenced file. With a key-sequenced file, addressed access can be used to retrieve records, update their contents, and delete records. (The length of a record and the contents of its key field cannot be changed.) Records cannot be added because VSAM will not allow changes to the file which could cause the index to change. With an entry-sequenced file, addressed access can be used to retrieve records, update their contents (but not change their length), and add new records to the end of the file. Records cannot be physically deleted because that would change the entry sequence of the file (RBAs of the records).

The discussions of free space in a key-sequenced file pointed out that keyed insertion, deletion, or update (length changing) of records can change their RBAs. Therefore, to use addressed access to process a key-sequenced file, the user may have to keep track of RBA changes. VSAM passes back the RBA of each record retrieved, added, updated, or deleted.

Addressed Retrieval.
Positioning for addressed-sequential retrieval is done by RBA rather than by key. When a processing program opens a file for addressed access, VSAM is positioned at the first record in the file in entry sequence to begin addressed-sequential processing. A POINT positions VSAM for sequential access beginning at the record whose RBA is indicated. A sequential GET causes VSAM to retrieve the data record at which it is positioned and positions VSAM at the next record in entry sequence.

With direct processing, the user may optionally specify that GET keep VSAM positioned at the next record in entry sequence. His program can then process the following records sequentially.

Addressed-sequential access retrieves records in entry sequence. If addressed-sequential retrieval is used for a key-sequenced file, records will not be in their key sequence if there have been control interval or control area splits.

Addressed-direct retrieval requires that the RBA of each individual record be specified, since previous positioning is not applicable. The address specified for a GET or a POINT must correspond to the beginning of a data record; otherwise, the request is invalid.

Addressed Deletion.
The ERASE macro can be used only with a key-sequenced file to delete a record that the user has previously retrieved for update.

With an entry-sequenced file, the user is responsible for marking a record he wants to delete. In other words, as far as VSAM is concerned, the record is not deleted. The space occupied by a record marked for deletion can be reused by retrieving the record for update and storing in its place a new record of the same length.

Addressed Storage.
VSAM does not insert new records into an entry-sequenced file, but adds them at the end. With addressed access of a key-sequenced file, VSAM does not insert or add new records.

A PUT macro instruction stores a record. A PUT for update following a GET for update stores the record that the GET retrieved. To update a record, the user must previously have retrieved it for update. He can update the contents of a record with addressed access, but he cannot alter the record's length. Neither can he alter the key field of a record in a key-sequenced file.

To change the length of a record in an entry-sequenced file, the user must store it either at the end of the file (as a new record) or in the place of a deleted record of the new length (as an update). He is responsible for marking the old version of the record as deleted.

VSAM Catalog, Service Programs, and Job Control

This section discusses creation of VSAM files and data spaces, through the VSAM catalog, by using the commands of Access Method Services and by using job control language.

The VSAM Catalog

VSAM is the only DOS/VS access method which can dynamically allocate direct access storage space for files. This is made possible by a central file, the VSAM catalog, which brings together extensive information about files and storage space. The catalog, which is connected to the system at IPL by a logical unit named SYSCAT, must be on the system whenever VSAM files are defined or processed.

The Catalog's Use in Data and Space Management
The VSAM catalog is a central information point for all VSAM files and the direct access storage volumes containing them. The catalog provides VSAM with the information to allocate data space for files, verify that the user is authorized to access them, compile usage statistics on them, and relate RBAs to physical locations. Consequently, the management of files is less dependent on job control or processing programs.

All VSAM files and indexes must be cataloged in the VSAM catalog. That is, a file's name and characteristics must be entered in the catalog when the file is defined.

There can be more than one VSAM catalog for a DOS/VS system. However, only one catalog at a time can be connected to the system. The catalog is connected to DOS/VS during IPL by the CAT command.

Access Method Services is used to define VSAM files in the catalog and to allocate space for them. The DLBL and EXTENT statements of job control are used to allocate VSAM data spaces on each volume that will contain VSAM files; they are not used either to catalog files or to allocate space for them from existing VSAM data spaces. (However, DLBL and EXTENT statements are used to allocate space for Unique files.) See *Job Control and VSAM.*

Information Contained in the Entries of a Catalog
The VSAM catalog is a key-sequenced file. The data of the catalog consists of entries describing files and entries describing direct access volumes in terms of the allocation of data spaces and the location of available space. The index of the catalog allows VSAM to find the file entry through its 44-byte name (file-ID) or the volume entry through the volume serial number. Except for Unique files, VSAM can allocate and deallocate space for files on cataloged volumes that are not mounted. However, if there is not enough unused data space to contain the file, the user must allocate new data space or assign other volumes that contain unused data space to the file.

Information in a File Entry.
File entries contain the information VSAM requires to translate a record's RBA to its physical location on a direct access volume. Besides the type of storage device and a list of volume serial numbers, the catalog keeps other file information, including:

- A pointer to the location of each extent of the file. These do not necessarily correspond to extents of the data space(s), specified on EXTENT statements, which contains the file, unless the file is Unique. They may be sub-allocated within the data space(s).
- Statistics on file processing, such as the number of records inserted or deleted, and the amount of free space.
- Attributes determined when the file was defined, such as control interval size, physical block size, logical record length, number of control intervals in a control area, and (for a key-sequenced file) location of the key field.
- Password protection information.
- The 44-byte name of the file (File-ID) as well as an indication of the connection between the data and the index of a key-sequenced file.
- Information used to determine whether only the data or only the index of a key-sequenced file has been processed.
- Indication of whether or not the file is shared.

Information in a Volume Entry.
Volume entries contain the information required to keep track of data spaces and files on the volumes. Each volume entry contains the following information:

- The volume serial number and device type of each volume.
- The location of data spaces on a volume.
- The location of files within data spaces on a volume
- The location and size of free data spaces available for sub-allocation to files.

Transporting Files Between Systems
Since all VSAM files must be cataloged, moving a file from one system to another requires that catalog information be moved along with it or that the copy of the file being moved be cataloged in the receiving system. The entire catalog and all the VSAM files of a system can be moved to another DOS/VS system or to an OS/VS system. Thus a VSAM volume or volumes can be made portable between systems. To use a VSAM volume from another DOS/VS system or an OS/VS system, the user need only assign its catalog by use of the CAT command during IPL.

He can also move individual files from one DOS/VS system to another or to an OS/VS system by using the EXPORT and IMPORT commands of Access Method Services.

Access Method Services

Access Method Services is a set of utility programs that are used with VSAM. A very brief description of Access Method Services follows. For a complete description see *DOS/VS Utilities: Access Method Services*, GC33-5382. The user tells Access Method Services what to do by giving commands and descriptive parameters through an input job stream or by calling it in a processing program and passing to it command statements. A set of conditional statements (IF, ELSE, DO, END, SET) allows the user to alter the sequence of execution of a series of commands.

There are Access Method Services functional commands for:

- Defining and maintaining files.
- Copying files, listing files, and listing catalog entries.
- Moving files from one operating system to another.
- Protecting data from loss.

Defining and Maintaining Files

The DEFINE command must be used to define all VSAM data spaces and files. It is also used to define the VSAM catalog. The ALTER command is used to change one or more of the attributes of a file after it has been defined. The DELETE command is used to remove a file from the catalog and free its data space for other VSAM files or to remove unused data space from VSAM's control and make it available for other DOS/VS files. The VERIFY command helps the user check and maintain file integrity.

DEFINE: Defining an Object and Allocating Space

To define a data space, entry-sequenced file, key-sequenced file, or catalog, the user specifies the DEFINE command and the object to be defined.

Defining Data Spaces.

The extents of a data space are specified in the DLBL and EXTENT statements of job control. Only a cross-reference to the DLBL and EXTENT statements (filename) is taken from the DEFINE command.

Defining Files.

For a key-sequenced file, a single DEFINE command defines its data and its index and their relationship. An entry-sequenced file or the data of a key-sequenced file together with its index is called a *cluster* by Access Method Services. The 44-byte name (File-ID) of the file (cluster) is specified in the DEFINE command. Separate names are given, by VSAM or by the user, to the data and the index of a key-sequenced file. These names enable the user to process the data and the index separately with addressed or control interval access.

The attributes of the file are specified in the DEFINE command. There are parameters for specifying whether data in a deleted file will be erased, whether passwords will be required to access the file, size and other attributes of data records, minimum amount of virtual-storage space for I/O buffers, percentages of free space in control intervals and in control areas of a key-sequenced file and other performance options, retention period, identification of the owner, whether a file can be shared, file preformatting options, and whether write operations are to be verified.

The amount of direct access storage space for a sub-allocated file is specified as the number of data records that it is to contain or as a number of physical units, such as tracks or cylinders. Specifying the number of records, independent of type of storage device, leaves the calculation of the number of physical units of space up to VSAM. It calculates the size of the control interval and control area to be used. The user may specify the control interval size, and VSAM will use it if it falls within the acceptable limits that VSAM calculates.

The volumes to contain the file must be specified; the order in which they will be used can also be specified. The space for a key-sequenced file can be allocated on volumes according to ranges of key values. The space for each range is extended separately when additional space is required. Examples of the use of space allocation parameters in the DEFINE statement are described under *Space Allocation for Sub-Allocated Files.*

For convenience, an existing catalog entry can be specified as a model for a new entry, if they are of the same type (entry-sequenced file or key-sequenced file). The information in the model will be used in the new entry unless the user overrides it.

ALTER: Modifying a Catalog Entry

Many of the attributes, which were defined explicitly or by default when a catalog entry was created, may be modified subsequently by the ALTER command, most of whose parameters are the same as the DEFINE parameters. The user can change the name of a file, the indication whether to erase the data in a deleted file, passwords, minimum amount of space for I/O buffers (which can be increased, but not decreased), percentages of free space in new control intervals and in control areas of a key-sequenced file, retention period, name of the owner, the indication whether to share a file, and the indication whether to verify write operations.

Certain attributes of the file cannot be modified, such as control interval size and placement of the index in direct access storage relative to the data of a key-sequenced file. Changing these attributes amounts to a reorganization of the file and requires that a new file be defined and the old file be copied into it.

DELETE: Removing a Catalog Entry and Freeing Space

The DELETE command removes the entry for a previously defined object and, in effect, causes it to cease to exist. The space is freed for use when new files are added to the data space. If the erase option is specified in the entry, the deleted file is overwritten with binary zeros. If a Unique file is deleted, the data spaces it occupies are also deleted and their labels are removed from the VTOC.

VERIFY: Testing and Reestablishing a File's Accessibility

The VERIFY command protects data when a file was not closed successfully the last time it was processed. It instructs Access Method Services to investigate whether an entry-sequenced file or both the data and the index of a key-sequenced file have been properly closed. The end of the data or of the index of a file is indicated by a special field, the end-of-file indicator, and by information in the file's catalog entry. If a file is divided into key ranges, the end of each key range is indicated by an end-of-file indicator. See *Protecting Data With VSAM.* The end may be improperly indicated in the catalog if an error prevented VSAM from closing the file. Access Method Services closes the file and modifies the catalog entry, if necessary, to correspond with the file.

Copying and Listing Files

The REPRO and PRINT commands enable the user to copy and list sequential, indexed-sequential, and VSAM files. The LISTCAT command enables the user to list a catalog or individual entries of a catalog.

REPRO: Converting and Reorganizing Files

The REPRO command instructs Access Method Services to get records from a sequential, indexed sequential, or VSAM file and put them into a sequential or VSAM file. It can be used to convert an indexed sequential file to a key-sequenced file. First, define a new key-sequenced file. Then copy the indexed-sequential file into the key-sequenced file. Access Method Services converts data records to the VSAM format and builds an index.

An old file can be reorganized by copying it into a newly defined file of the same type. With key-sequenced files, the user can optionally specify different percentages of distributed free space and different performance options for the new file. Copying the old key-sequenced file into the new one redistributes free space, makes the entry sequence of the data records the same as their key sequence, and builds a new index.

The file into which records are copied may either be newly allocated (by way of the DEFINE command) or contain records already. Records copied into a key-sequenced file are merged with any existing records; records copied into an entry-sequenced file are added at the end. A range of records to be copied can be specified by number of records, by key in an indexed-sequential or a key-sequenced file, or by address in either type of VSAM file.

PRINT: Listing Data Records

The PRINT command instructs Access Method Services to list some or all of the records of a sequential, indexed sequential, or VSAM file in one of three formats: each byte as 2 hexadecimal digits, each byte as a single character, or a combination of these two, side-by-side. A range of records to be listed can be specified by number of records, by key in an indexed sequential or key-sequenced file, or by address in either type of VSAM file.

LISTCAT: Listing Catalog Entries

The LISTCAT command lists individual entries, all entries of a particular type, or all entries of a given catalog. The entire entry is listed, but passwords in an entry are not listed unless the master password for the file defined by the entry or the master password for the catalog itself is specified.

Moving Files from One System to Another

VSAM files can be moved between DOS/VS systems and between a DOS/VS and an OS/VS system by using the EXPORT and IMPORT commands of Access Method Services. Figure 80 compares volume and file portability. See the section *Transporting Files Between Systems* for a description of volume portability.

EXPORT: Extracting Catalog Information and Making a File Portable

The EXPORT command instructs Access Method Services to copy an entry-sequenced file or a key-sequenced file in the format of a sequential file onto a storage volume to be transported to another system. The transporting volume may be magnetic tape or disk. Access Method Services also extracts information from the catalog entry that defines the file to be transported and copies it onto the transporting volume. The information is used to define the file automatically in a VSAM catalog in the other operating system.

Exportation is either permanent or temporary. In permanent exportation, Access Method Services deletes the catalog entry and frees the storage space; in temporary exportation, both the sending and the receiving systems have a copy of the file, and the user may specify that one or both of the copies are not to be modified. A protected copy can only be read. The copy can be freed for full access with the ALTER command.

IMPORT: Loading a Portable and Its Catalog Information

The IMPORT command instructs Access Method Services to define the entry-sequenced file or the key-sequenced file on the transporting volume, using the catalog information extracted in exportation. The file itself is stored in its VSAM format in a data space that is defined in the catalog.

The EXPORT and IMPORT commands can be used to prepare a backup copy of an entry-sequenced file or a key-sequenced file and its index and their catalog entries and to load the backup copy if it is needed. When a backup copy is imported, the catalog entry is regenerated.

Space Allocation for Sub-Allocated Files

Unless a VSAM file is defined as Unique, the direct access space for it is suballocated from previously existing VSAM data spaces by using the DEFINE statement as follows.

- The volume or volumes on which the file will reside are indicated in the VOLUMES parameter. Space to be allocated initially (primary allocation) and, optionally, space to be allocated if the file must be extended (secondary allocation) are indicated in the TRACKS, CYLINDERS, or RECORDS parameters. VSAM selects which data spaces or portions of data spaces on a volume to suballocate to the file.
- If more volumes are specified than needed for the primary allocation, the additional volumes can be used when the file is extended. These volumes are described in the file's catalog entry as potential *overflow* volumes.
- The KEYRANGES parameter enables the user to put specifies parts of a file on different volumes. The amount of space specified in the primary allocation parameter is allocated to *each* key range.
- If the ORDERED parameter is specified, the space must be suballocated on the volumes in the order in which they are listed in the VOLUMES parameter.

The following examples illustrate the use of these parameters in the DEFINE statement for suballocated files:

```
VOLUMES (A B C)
ORDERED
CYLINDERS (50 5)
```

The 50-cylinder primary space allocation for the file must be made on volume A, or the request will be rejected. Volumes B and C are overflow volumes. If the file is extended, a five cylinder secondary space allocation is made on volume A if it has enough data space. Otherwise, the secondary allocation is made on volume B. If volume B does not have enough data space for a secondary allocation, the request for extension is rejected. When the file is subsequently extended, the allocation is made on volume B if it has enough data space. Otherwise, the allocation is made on volume C.

```
VOLUMES (A B C)
UNORDERED
CYLINDERS (50 5)
```

The 50-cylinder primary allocation for the file can be made on either volume A, volume B, or volume C. However, if all 50 cylinders cannot be allocated on one volume, the request is rejected. The volumes are searched in the order they are specified. If both volume A and volume B have 50 cylinders available, the allocation will be made on volume A. If the file is extended, the five cylinder secondary allocation can be made on any of the three volumes. Once again, the volumes are searched for space in the order specified.

```
VOLUMES (A B C)
KEYRANGES ( (00 30) (31 65) (66 99) )
ORDERED
CYLINDERS (100 10)
```

A primary allocation of 100 cylinders will be made for *each* key range. The first key range will be one volume A, the second on volume B, and the third on volume C. If 100 cylinders cannot be allocated on each volume, the request is rejected. A key range can be extended only on the volume it occupies or on an overflow volume. Thus, if volume D were added to the list, the first key range could be extended on volume A or (if volume A were full) on volume D. If only volumes A and B were specified, the first key range would be allocated on volume A and the second and third key ranges would be allocated on volume B.

```
VOLUMES (A B C)
KEYRANGES ( (00 30) (31 65) (66 99) )
UNORDERED
CYLINDERS (50 5)
```

A primary allocation of 50 cylinders will be made for *each* key range. VSAM will attempt to put one key range on each volume. If volume A does not have 50 cylinders available, the first key range is put on volume B and the second and third on volume C. If neither volume A nor volume B have 50 available cylinders, all three key ranges are placed on volume C. A key range will be extended first on the volume it is on, then it will be extended on any other volume in the list. However, if volume D was available as an overflow volume, each key range would be extended on volume D if no more space were available on the volume of its primary allocation. A key range can cover only two volumes—the one of primary allocation and one more.

```
VOLUMES (A B C)
ORDERED
CYLINDERS (800 10)
```

This request will be rejected because the amount of primary space allocation can never be greater than one volume. The file being defined should be split into key ranges (key-sequenced file only), or the secondary allocation parameter should be used so that the file can be extended.

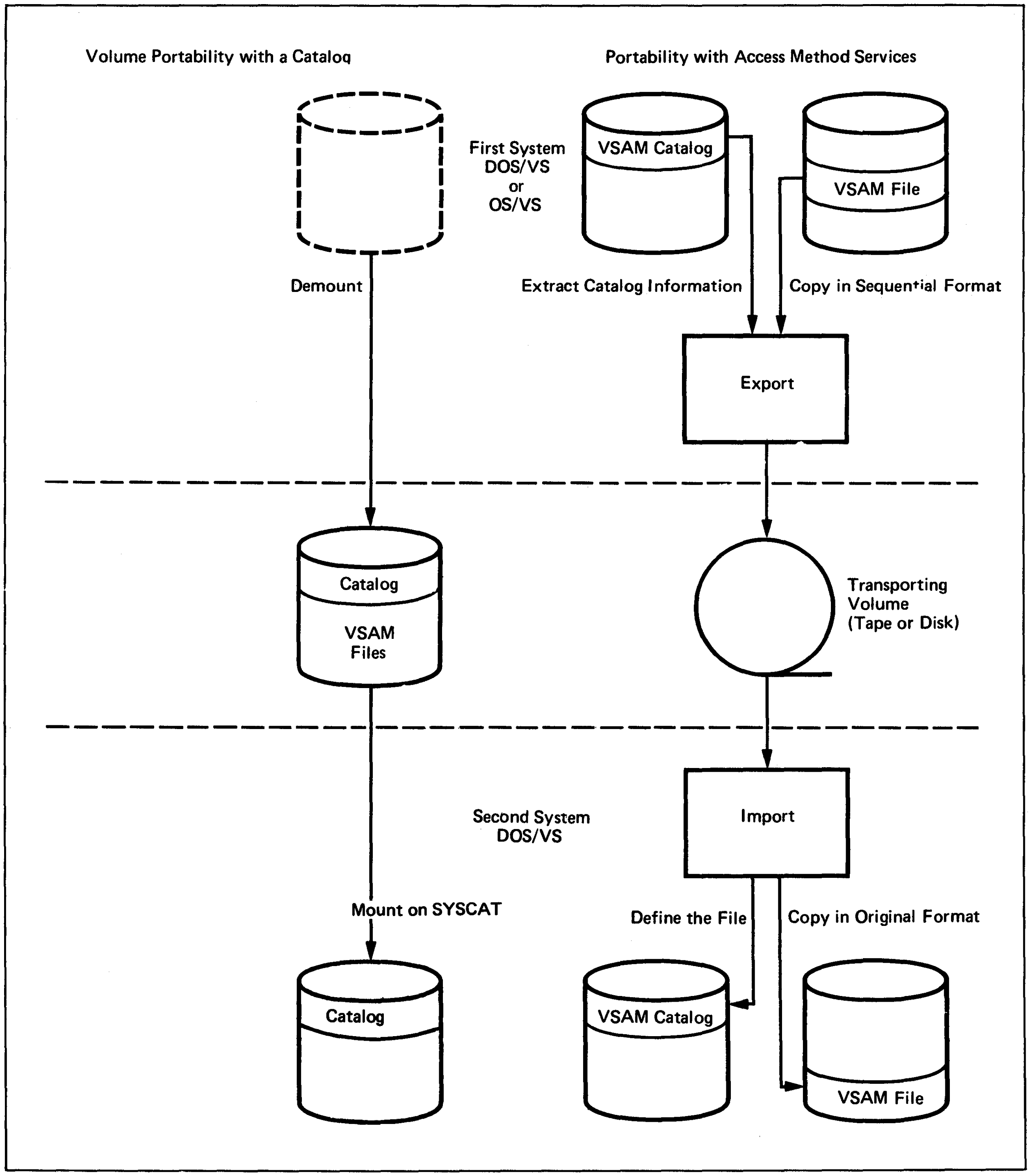

Volume Portability with a Catalog
Portability with Access Method Services
First System DOS/VS or OS/VS
VSAM Catalog
VSAM File
Demount
Extract Catalog Information
Copy in Sequential Format
Export
Catalog
VSAM Files
Transporting Volume (Tape or Disk)
Second System DOS/VS
Import
Mount on SYSCAT
Define the File
Copy in Original Format
Catalog
VSAM Catalog
VSAM File

Figure 4.27. Data Portability

File portability achieved by moving volumes or by moving individual files.

Job Control and VSAM

VSAM data spaces and files are specified to DOS/VS job control by the DLBL and EXTENT statements. DLAB, XTENT, and VOL statements cannot be used with VSAM. EXTENT statements specifying the same volume serial number must be contiguous. The VSAM catalog is connected to DOS/VS by assigning the unit that has the volume containing the catalog to the logical unit name SYSCAT. This is done at IPL by the command CAT UNIT=X'cuu'. Figure 4.28 shows the use of DOS/VS job control statements to create or process a VSAM file.

Reserving Virtual Storage for VSAM

VSAM modules are loaded into virtual storage dynamically during OPEN time. You must reserve this virtual storage using the SIZE parameter of the EXEC job control statement. If you have loaded the VSAM modules into the SVA, working storage and buffer space must still be reserved in the partition.

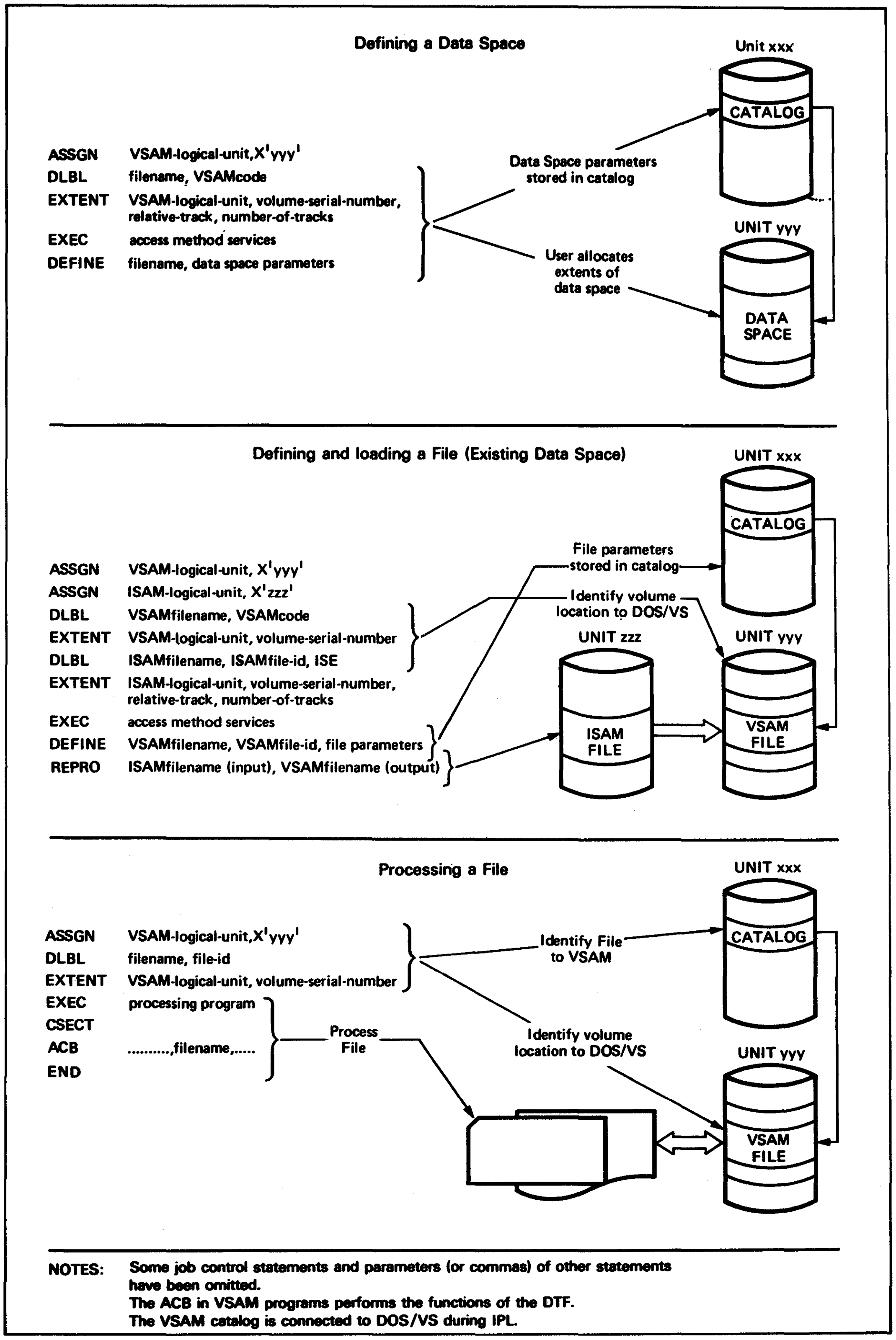
Defining a VSAM Data Space

When a data space is defined, DLBL and EXTENT statements are required. They are used by VSAM catalog and space management routines. The DEFINE command of Access Method Services specifies the data space being defined. The data space is described by standard format-1 and format-3 labels in the VTOC; it is also described in the VSAM catalog. Two examples of job control statements used to define data spaces are shown below:

```
// JOB     ALLOCATE A 3330 VOLUME TO VSAM
// ASSGN   SYS001,X'130'
// DLBL    VFILENM,,VSAM
// EXTENT  SYS001,514966,1,,001,7675
// EXEC    IDCAMS,SIZE=26K
   DEFINE  SPACE( FILE (VFILENM) TRACKS (7675) -
             VOLUME (514966) )
/*
/&
```

```
// JOB     DEFINE A VSAM DATA SPACE
// ASSGN   SYS001,X'130'
// DLBL    VFILENM,,VSAM
// EXTENT  SYS001,321942,1,,800,400
// EXTENT  SYS001,321942,1,,1800,200
// EXEC    IDCAMS,SIZE=26K
   DEFINE  SPACE( FILE(VFILENM) TRACKS (600) -
             VOLUMES (321942) )
/*
/&
```

In the DLBL statement the user must specify the filename and the code which identifies VSAM files. The filename (VFILENM) is the same as the FILE parameter and connects the job control statements to the DEFINE command. VSAM data spaces can be deleted only by using the DELETE statement of Access Method Services even though the expiration data has been reached. Therefore the user should specify an expiration date to prevent other access methods from overwriting VSAM data spaces. In the EXTENT statement he must specify the symbolic unit name, the volume serial number, and the space parameters (beginning-track and number-of-tracks). Note that the VOLUMES parameter and the space allocation parameter (CYLINDERS, TRACKS, or RECORDS) must be included in the DEFINE command, and must agree with the information in the EXTENT statements. If the CYLINDERS parameter is used, each extent must begin on a cylinder boundary.



Defining a Data Space

```
ASSGN    VSAM-logical-unit,X'yyy'
DLBL     filename, VSAMcode
EXTENT   VSAM-logical-unit, volume-serial-number,
         relative-track, number-of-tracks
EXEC     access method services
DEFINE   filename, data space parameters
```

Unit xxx
CATALOG
Data Space parameters stored in catalog
User allocates extents of data space
UNIT yyy
DATA SPACE

Defining and loading a File (Existing Data Space)

```
ASSGN    VSAM-logical-unit, X'yyy'
ASSGN    ISAM-logical-unit, X'zzz'
DLBL     VSAMfilename, VSAMcode
EXTENT   VSAM-logical-unit, volume-serial-number
DLBL     ISAMfilename, ISAMfile-id, ISE
EXTENT   ISAM-logical-unit, volume-serial-number,
         relative-track, number-of-tracks
EXEC     access method services
DEFINE   VSAMfilename, VSAMfile-id, file parameters
REPRO    ISAMfilename (input), VSAMfilename (output)
```

UNIT xxx
CATALOG
File parameters stored in catalog
Identify volume location to DOS/VS
UNIT zzz
UNIT yyy
ISAM FILE
VSAM FILE

Processing a File

```
ASSGN    VSAM-logical-unit,X'yyy'
DLBL     filename, file-id
EXTENT   VSAM-logical-unit, volume-serial-number
EXEC     processing program
CSECT
ACB      ..........,filename,.....
END
```

NOTES: Some job control statements and parameters (or commas) of other statements have been omitted.
The ACB in VSAM programs performs the functions of the DTF.
The VSAM catalog is connected to DOS/VS during IPL.

Figure 4.28. Using Job Control With VSAM

A data space can consist of up to 16 separate extents on a volume. If EXTENT statements are supplied for more than one volume, a separate data space is defined on each volume. Each extent is checked against the extent limits of each existing data space or file of other DOS/VS access methods. If a new extent overlaps any previously written extent, VSAM issues a message to the operator. He can cancel the job or, if the old extent is part of a file of another access method, delete the file.

Defining a Sub-Allocated VSAM File

When a VSAM file is defined and the space for it is sub-allocated from one or more existing data spaces, DLBL and EXTENT statements are not required and label processing is not performed because (1) information needed to set up the file is in the DEFINE command and (2) information about the data spaces to be used for the file is in the VSAM catalog. An example of job control statements used to define a sub-allocated VSAM file is shown below. Note that the DEFINE parameters needed to specify the logical attributes of the file are not shown.

```
// JOB      DEFINE A SUB-ALLOCATED VSAM FILE
// EXEC     IDCAMS,SIZE=26K
   DEFINE   CLUSTER( NAME(MSTRFILE) -
            VOLUMES (321942) TRACKS (300) -
            ...... )
/*
/&
```

Defining a VSAM file involves building a catalog entry for the file and finding free data space for it. The volume(s) on which the file is defined need not be mounted because only the catalog is used. However, they must contain previously defined data space. You can load the file in the same job step by using the REPRO command of Access Method Services. In this case, the volume(s) must be mounted.

Defining a Unique VSAM File

A file can be defined at the same time as the data spaces which will contain it. In this case, the file is called Unique and no other file can occupy its data spaces. The data and the index of a key-sequenced Unique file must occupy separate data spaces; each requires DLBL and EXTENT statements. However, an entry-sequenced file occupies only one data space. An example of job control statements used to define a key-sequenced Unique file is shown below. Note that the DEFINE parameters needed to specify the logical attributes of the data and index are not shown.

```
// JOB      DEFINING A UNIQUE VSAM FILE
// ASSGN    SYS001,X'130'
// DLBL     VDATANM,,,VSAM
// EXTENT   SYS001,321942,1,,800,400
// EXTENT   SYS001,321942,1,,1800,400
// DLBL     VINDXNM,,,VSAM
// EXTENT   SYS001,321942,1,,600,200
// EXEC     IDCAMS,SIZE=26K
   DEFINE   CLUSTER ( NAME (MSTRFILE) UNIQUE ) -
              DATA ( FILE (VDATNAM) -
              VOLUMES (321942) CYLINDERS (40) -
              ...... ) -
              INDEX ( FILE (VDINDXNM) -
              VOLUMES (321942) CYLINDERS (10) -
              ...... )
/*
/&
```

The DLBL and EXTENT statements contain the same information as described above under *Defining a VSAM Data Space.* Note that the

VOLUMES parameter and the space allocation parameter (CYLINDERS, TRACKS, or RECORDS) must be included in the DEFINE command, and must agree with the information in the EXTENT statements. The space allocated to a Unique file must be an integral number of cylinders and each extent must begin on a cylinder boundary. However, the TRACKS or RECORDS parameter can be used in the DEFINE command. If the KEYRANGES option is specified, the number of volumes must be equal to the number of key ranges. Excessive volumes will be ignored and missing volumes will cause an error. A Unique file can have a maximum of 16 extents per volume.

A Unique file cannot be extended, and space left over after the records are loaded cannot be released. The file is limited to the extents of the data spaces which contain it. The extents of a Unique file, like those of other access methods, are described in the VTOC. They are also described in the VSAM catalog. Since the user chooses the size of the area allocated to the index, he must ensure that is is large enough to contain the index.

Processing a VSAM File

When an existing file is processed, DLBL and EXTENT statements are required. They are used by VSAM OPEN routines. The catalog contains most of the information required by VSAM to process a file. VSAM does not require much information from job control statements. An example of job control statements used for processing an existing VSAM file is shown below:

```
// JOB      PROCESS A VSAM FILE
// ASSGN    SYS001,X'130'
// DLBL     VFILENM,'MSTRFILE',,VSAM
// EXTENT   SYS001,321942
// EXEC     USERPGM,SIZE=nnK
   CSECT
   .
   .
   ACB      DDNAME=VFILENM,...
   .
   .
   END
/*
/&
```

File-ID, type code (VSAM), and filename are required in the DLBL statement. The filename, (VFILENM) is a cross reference to the ACB and connects job control to the processing program. The file-ID (MSTRFILE) matches the 44-byte name of the file, stored in the catalog, and connects job control and the processing program to the file.

Only the symbolic unit number and the volume serial number are required in the EXTENT statement. The space parameters are not used when referring to an existing file becuase that information is in the VSAM catalog. One EXTENT statement is required for each volume the file occupies. An EXTENT statement which specifies a volume not listed in the file's catalog entry is ignored. If the correct volume is not mounted on a logical unit, a message is issued to the operator to mount the correct volume.

If a multi-volume file is opened for direct or keyed sequential processing, all volumes must be mounted at that time. If it is opened for addressed sequential processing only, the volumes need only be mounted as they are used.

When processing a Unique file, the user can supply the same DLBL and EXTENT cards that were used to define the file. VSAM obtains the volume serial number from the first EXTENT card for each volume and ignores the space parameters on all the EXTENT cards.

Defining the VSAM Catalog

Before VSAM can be used on a DOS/VS system, the VSAM catalog must be created. The catalog is defined, like a VSAM file, by the DEFINE command of Access Method Services. The catalog occupies a data space, defined at the same time, which is described by format-1 and format-3 labels in the VTOC. Two examples of job control statements used to define a catalog are shown below:

```
// JOB      DEFINE A VSAM CATALOG
// DLBL     IJSYSCT,'VSAMCAT',,VSAM
// EXTENT   SYSCAT,321940,1,,100,250
// EXEC     IDCAMS,SIZE=26K
   DEFINE   MASTERCATALOG( NAME(VSAMCAT) -
              VOLUME (321940) TRACKS (250) -
              FILE (IJSYSCT) )
/*
/&
```

```
// JOB      DEFINE CATALOG AND ALLOCATE VOLUME
// DLBL     IJSYSCT,,,VSAM
// EXTENT   SYSCAT,321940,1,,020,3980
// EXEC     IDCAMS,SIZE=26K
   DEFINE   MASTERCATALOG( NAME(AMASTCAT) -
              VOLUME (321940) CYLINDERS (199) -
              FILE (IJSYSCT) ) -
              DATA ( CYLINDERS (10) )
/*
/&
```

In the first example, the catalog and its data space are defined. In the second example, an entire 2314 volume is allocated as VSAM data space and the catalog is defined and allocated 10 cylinders of that data space.

In the DLBL statement, the user must specify the filename and the code which identifies VSAM. The filename must be specified as IJSYSCT. The 44-byte name (File-ID) of the catalog is chosen by the user and must be specified in the DEFINE command. It is not required in the DLBL statement when the catalog is being defined. In the EXTENT statement, the logical unit must be SYSCAT. The user must decide which volume and which extents will contain the catalog. The volume must have been assigned to a unit by the CAT command during IPL. Note that the VOLUMES parameter and the space allocation parameter (CYLINDERS, TRACKS, or RECORDS) must be included in the DEFINE command, and must agree with the information in the EXTENT statements. If the CYLINDERS parameter is used, each extent must begin on a cylinder boundary. The data space containing the catalog can consist of up to 16 extents.

VSAM opens the catalog, after it has been defined, once in each job step in which a file or data space is defined or processed. The catalog must be on the volume mounted on the unit specified in the CAT command.

Assembler Language Considerations

VSAM macro instructions are coded in an assembler language processing program to gain access to the data. There are macros for:

- Connecting and disconnecting a processing program and a file. These prepare a bridge for VSAM between the program and the data.
- Specifying parameters that relate the program and the data. These identify the file and describe the kind of processing to be done.
- Manipulating the information relating the program and the data. These are used to specify changes in processing.
- Requesting access to a file. These initiate the transfer of data between direct access storage and virtual storage.

Connecting and Disconnecting a Processing Program and a File

OPEN connects a processing program to a file, so that VSAM can satisfy the program's requests for data. CLOSE completes processing and frees resources that were obtained by the OPEN routine. TCLOSE secures records that were added to the file.

OPEN: Connecting a Processing Program to a File

The OPEN routine, by calling a VSAM authorization routine, verifies that a processing program has the authority to process a file.

OPEN constructs VSAM control blocks and loads the routines required for processing. (VSAM routines are not link-edited into the processing program like those of other access methods.) By examining the DLBL statement indicated by the ACB macro filename parameter and the volume information in the catalog, OPEN verifies that the necessary volumes have been mounted and checks whether each volume matches its catalog information. If a key-sequenced file is being opened, OPEN checks whether the data of the file has been updated separately from its index. OPEN also checks that the user is authorized to access the file; the password he supplies must match the appropriate password in the catalog.

CLOSE: Disconnecting a Processing Program from a File

The CLOSE routine completes any I/O operations that are unfinished when a processing program issues a CLOSE macro for a file.

CLOSE updates the catalog for any changes in the attributes of a file. The addition of records to a file may cause its end-of-file indicator to change, in which case CLOSE updates the end-of-file indicator in the catalog. These end-of-file indicators help ensure that the entire file is accessible. If an error prevents VSAM from updating the indicators, the file is flagged as not properly closed, and subsequent executions of the OPEN macro are given an error code indicating the failure. For more information on correcting this condition, see the discussion of the Access Method Services VERIFY command in the section *Data Security and Integrity.*

CLOSE restores control blocks to the status that they had before the file was opened, and frees the virtual storage space that OPEN used to construct VSAM control blocks.

TCLOSE: Securing records added to a file

A TCLOSE macro is issued to complete unfinished operations and update the catalog. Processing may continue without reopening the file. The TCLOSE macro is used to protect data while the file is being loaded. This is described in the section *Protecting Data with VSAM.*

Specifying Parameters that Relate the Program and the Data

To open a file for processing, the user must identify the file and the types of processing to be done. The ACB macro specifies the file to be processed, the EXLST macro specifies a list of user-supplied exit routines, and the RPL macro specifies information for a request to access a particular record in the file. These are the declarative macros of VSAM. The GENCB macro can be used in place of the ACB, EXLST, or RPL macros to generate processing specifications during the execution of a processing program rather than while assembling or compiling the program.

ACB: Specifying the Access-Method Control Block

The ACB macro specifies an Access-Method Control Block (ACB). Each VSAM file has an ACB; it identifies the file and contains information about it. The filename of the DLBL job control statement that describes the file is included, so that the OPEN routine can connect the program to the data.

The other information specified enables OPEN to prepare for the kind of processing to be done:

- The address of a list of exit-routine names that the user supplies (EXLST parameter). The EXLST macro, described next, is used to construct the list.
- The amount of space for I/O buffers (BUFSP parameter) and the number of I/O buffers (BUSND and BUFNI parameters) that VSAM will use to process data and index records. The minimum number of buffers allowed depends on how much buffer space is allocated, the number of concurrent requests to be allowed, and whether processing will be direct or sequential.
- The password, if required, for the level of authorization to access the file (read, read and update, etc.) (PASSWD parameter).
- The processing options to be used (MACRF parameter): keyed or addressed access, or both; sequential, direct, or skip sequential processing, or a combination; retrieval, storage, or update (including deletion), or a combination.
- For processing concurrent requests, the number of requests that are defined for processing the data set. (See the discussion of the RPL macro following EXLST.)

EXLST: Specifying the Exit List

The EXLST macro specifies the names of optional exit routines that the user can supply for analyzing physical and logical errors and for end-of-file processing. Any number of ACBs in a program can indicate the same Exit List, and an exit routine can be used for more than one file.

Analyzing Physical Errors (SYNAD).
When VSAM encounters an error in an I/O operation that the DOS/VS error recovery routines cannot correct, it exits to the physical-error analysis (SYNAD) routine. VSAM sets a code in the RPL to indicate whether the I/O error occurred during reading or writing the data or the index.

Analyzing Logical Errors (LERAD).
Errors not directly associated with an I/O operation, such as an invalid request, cause VSAM to exit to the logical-error analysis (LERAD) routine. VSAM sets a code in the RPL that indicates the type of logical error.

End-of-File Processing (EODAD).
When a record beyond the last record in the file is requested during sequential access, the end-of-file (EODAD) routine is given control. The last record is the highest-addressed record for addressed or control-interval access or the highest-keyed record for keyed access. If an EODAD exit routine is not available, control is given to the LERAD exit routine.

Overlapping I/O Operations (EXCPAD).
When VSAM starts an I/O operation caused by a request macro, the execute-channel-program (EXCPAD) exit routine is given control. The EXCPAD routine must return control to VSAM, which continues the user's mainline routine at the instruction following the request macro. The EXCPAD exit is intended for use by programmers of utilities and systems.

RPL: Specifying the Request Parameter List

The RPL macro produces a Request Parameter List (RPL). The RPL specifies the information needed by a request macro to access a record in the file. The request macros are GET, PUT, POINT, and ERASE (described under *Requesting Access to a File*). The RPL identifies the file to which the request is directed by naming the ACB of the file. The MODCB macro (described below) is used to modify some of the parameters for changing the type of processing, such as from direct to sequential or from update to nonupdate.

For concurrent requests, that require VSAM to keep track of more than one position in a file, any number of RPL macros may be used asynchronously by a processing program or its subtasks to process a file. The requests can be sequential or direct or both, and they can be for records in the same part or different parts of the file.

Only the RPL parameters appropriate to a given request need the specified:

Processing Options for a Request (OPTCD).
A request is for keyed, addressed, or control-interval access. The processing can be sequential, skip sequential (keyed access only), or direct. The request is for updating or not updating a record. A nonupdate direct request to retrieve a record can optionally cause positioning at the following record for subsequent sequential access.

For a keyed request, the user specifies either a generic key or a full key to which the key field of the record is to be matched. A generic key can match several records while a full key matches only one record. He can also specify that, if the key does not match the key of any record in the file, the record with the next greater key will be processed.

For retrieval, a request is either for a data record to be placed in a work area in the processing program (move mode) or for the address of the record within VSAM's I/O buffer to be passed to the processing program (locate mode).

Address of the Work Area for or Pointer to a Data Record (AREA).
For retrieval, update, insertion, or addition, a work area for the records must be provided in the user's partition (move mode). For retrieval, the user can have VSAM give him the address of the record within VSAM's I/O buffer (locate mode) in this field.

Size of the Work Area for a Data Record (AREALEN).
This parameter specifies the length of the work area in which the user and VSAM place records for move mode or a four-byte address of the record for locate mode. Having a work area that is too small is considered a logical error.

Length of the Data Record Being Processed (RECLEN).
For storage, the processing program indicates the length to VSAM; for retrieval, VSAM indicates it to the processing program.

Length of the Key (KEYLEN).
This parameter is required only for processing by generic key. For ordinary keyed access, the full key length is available from the catalog.

Address of the Area Containing the Search Argument (ARG).
The search argument is either a key or an RBA. If the OPTCD parameter indicates a generic key, the KEYLEN parameter tells how many high-order (leftmost) bytes of the search argument will be used.

Address of the Next RPL in a Chain (NXTRPL).
The user can process several records with a single GET, PUT, or ERASE by chaining RPLs together. For example, each RPL in a chain could contain a unique search argument and point to a unique work area. A single GET macro would retrieve a record for each RPL in the chain. Chaining RPLs is not the same as issuing concurrent requests that require VSAM to keep track of multiple positions in a file. A chain of RPLs is processed as a single request.

GENCB: Generating Control Blocks and Lists

The GENCB macro can be used to generate an ACB, EXLST, or RPL during the execution of the processing program, rather than to assemble it with the corresponding macro. GENCB is coded the same as the other macros, but it generates one or more copies of a control block or list.

Manipulating the Information Relating the Program and the Data

The macros MODCB, SHOWCB, and TESTCB are for modifying, displaying, and testing the contents of an ACB, EXLST, or RPL.

MODCB: Modifying the Contents of Control Blocks and Lists

The MODCB macro is used to specify new values for fields in an ACB, EXLST, or RPL. For example, if an RPL is used to retrieve directly the first record having a certain generic key and then to retrieve sequentially the rest of the records having that generic key, MODCB would be used to alter the RPL to change from direct to sequential access.

SHOWCB: Displaying Fields of Control Blocks and Lists

SHOWCB enables the user to examine the contents of an ACB, EXLST, or RPL. VSAM displays the requested fields in an area provided by the user. Fields additional to those defined in the macros can be displayed. For example, when a file is open, the user can display various counts, such as number of control interval splits, number of deleted records, and number of index levels. The RBA of the last record accessed and the error codes set in the ACB or RPL after macro execution can also be displayed.

TESTCB: Testing the Contents of Control Blocks and Lists

The TESTCB macro enables the user to test the contents of a field or combination of fields in an ACB, EXLST, or RPL and to alter the sequence of the processing steps. It is similar to a branch instruction. He can test the error codes set in the ACB or the RPL, for instance, or the attributes of a file, such as record length.

Requesting Access to a File

All of the preceding macros prepare to process a file. The request macros, GET, PUT, POINT, and ERASE, initiate an access to data. Each of these macros is associated with a Request Parameter List that fully defines the request. Another request macro, ENDREQ, is provided to (1) terminate processing of a request when completion is not required, or (2) free VSAM from keeping track of a position in the file. The only parameter that is specified with a request macro is the identity of the Request Parameter List.

The options for using GET, PUT, POINT, and ERASE are outlined in the discussion of the RPL macro, and the use of each macro is discussed in the section *VSAM Processing Procedures*.

Summary

A summary of the assembler language processing options is given in Figure 4.29.

VSAM FUNCTION	ENTRY–SEQUENCED ORGANIZATION (RBA)	KEY–SEQUENCED ORGANIZATION (INDEX)	INDEX UPDATED	RECORD MOVED
CREATE or REORGANIZE	Use AMS; "REPRO" SAM, or VSAM file in problem program.	Use AMS ; " DEFINE " catalog entry, then " REPRO " SAM, ISAM, or VSAM file	YES	
ADD	New records are placed at the end of the file,	ADD or INSERT by KEY only. " REPRO " into partly filled file (merge)	YES if split occurs	Right in contr. interv. or to next cntr. interval (split)
DELETE	NO. User program must recognize 'deleted' records.	Use "GET UPDATE" plus "ERASE"	NO	left
UPDATE	Use "GET UPDATE" plus "PUT UPDATE"	Use "GET UPDATE" plus "PUT UPDATE" Record length may change.	YES if split occurs	left or right
RETRIEVE SEQUENTIAL BY LOCATION (RBA)	For first record: "POINT" with RBA and "GET"; subsequent records: subsequent "GET"s	For first record: "POINT" with RBA and "GET"; subsequent records: subsequent "GET"s		
RETRIEVE SEQUENTIAL BY KEY* (INDEX)	NO	For first record: "POINT" with KEY and "GET"; subsequent records: subsequent "GET"s.		
RETRIEVE RANDOM BY LOCATION (RBA)	For each record : determine RBA, and "GET" for Retrieve, update.	For each record: determine RBA, and "GET" for Retrieve, update or delete.		
RETRIEVE RANDOM BY KEY* (INDEX)	NO (User may develop a randomizing routine to determine an RBA)	For each record: GET with KEY		

Figure 4.29. VSAM processing summary for assembler language

* KEY can be equal to, or less than the one produced as output by VSAM; generic (truncated) keys can be processed.

Using ISAM Programs with VSAM

This section is intended for users of ISAM who are converting to VSAM. It contains detailed information for programmers to decide whether existing ISAM programs can use the ISAM Interface Program (IIP) to process files that have been converted from ISAM format to VSAM format. The IIP minimizes conversion costs and scheduling problems by permitting ISAM programs to process VSAM files. ISAM programs can process ISAM files and VSAM files concurrently through the IIP.

Comparison of VSAM and ISAM

In most cases, if the performance options described in the section *Optimizing the Performance and Storage of VSAM* are used, performance is better with VSAM while achieving essentially the same results that can be achieved with ISAM; VSAM can also achieve results that cannot be achieved with ISAM. The use of existing ISAM processing programs to process key-sequenced files depends upon the extents to which VSAM and ISAM are similar in what they do, as well as upon the limitations of the IIP. This subsection describes the similarities and differences between VSAM and ISAM in the areas that the user is familiar with from using ISAM and indicates the functions of VSAM that have no counterpart in ISAM.

Comparison of VSAM and ISAM in Common Areas

A number of things that ISAM does are done differently or not at all by VSAM, even though the same practical results are achieved.

Index structure.

Both a VSAM key-sequenced file and an indexed-sequential file have an index that consists of levels, with a higher level controlling a lower level. In ISAM, some or all of the index records of the cylinder index can be kept in virtual storage. VSAM keeps individual index records in virtual storage, the number depending on the amount of buffer space provided.

Relation of index to data.

The relation of a VSAM index to the direct access storage space whose records it controls is quite different from the corresponding relation for ISAM, with regard to overflow areas for record insertion.

ISAM keeps a two-part index entry for each primary track that a file is stored on. The first part of the entry indicates the highest-keyed record on the primary track. The second part indicates the highest-keyed record from that primary track that is in the overflow area and gives the physical location in the overflow area of the lowest-keyed record from that primary track. All the records in the overflow area from a primary track are chained together, from the lowest-keyed to the highest-keyed, by pointers that ISAM follows to locate an overflow record. Overflow records are unblocked, even if primary records are blocked.

VSAM does not distinguish between primary and overflow areas. A control interval, whether used or free, has an entry in the sequence set, and after records are stored in a free control interval, it is processed exactly the same as other used control intervals. Data records are blocked in all control intervals and addressed, without chaining, by way of an index entry that contains the key (in compressed form) of the highest-keyed record in a control interval.

Defining and loading a file.
All VSAM files are defined in the catalog. Records are loaded into a file with Access Method Services or with the processing program, in one execution or in stages. When loading new records into an empty key-sequenced file, the index is built automatically. Access Method Services does not merge input files. However, for a key-sequenced file, input records are merged in key sequence with existing records of the output file.

VSAM Functions That Go Beyond ISAM
VSAM has capabilities that ISAM does not have:

Addressed sequential access. The user can retrieve and store the records of a key-sequenced file by RBA, as well as by key. With ISAM, he can position by physical address, but he must retrieve in a separate request.

Direct retrieval by generic key. VSAM can retrieve a record directly, not only with a full-key search argument, but also with a generic search argument. ISAM can only position at a record by generic argument: the record must be retrieved separately.

Deletion of records. With ISAM, records cannot be deleted until the file is reorganized; the user must keep track of the records he wants to delete. VSAM automatically reclaims the space in a key-sequenced file and combines it with any existing free space in the affected control interval. VSAM's use of distributed free space for insertions and deletions requires less file reorganization than ISAM does.

Concurrent request processing. A processing program can issue concurrent requests from a single ACB. The requests can be sequential or direct, or both for the same part or different parts of the file. VSAM maintains a position in the file for each concurrent request.

Secondary allocation of storage space. When a VSAM file is defined without the Unique option, direct access storage space for automatic secondary allocation can be specified. The amount of space can be specified in number of data records or in number of tracks or cylinders.

Automatic file reorganization. VSAM partially reorganizes a key-sequenced file by splitting a control area when it has no more free control intervals and one is needed to insert a record. VSAM allocates a new control area and gives it the contents of approximately half of the control intervals of the old control area: about half of the control intervals of each control area are then free.

No abnormal terminations by OPEN. The VSAM OPEN routine does not abnormally end, but returns an explanatory message in all cases where it cannot carry out a request to open a file.

How to Use the ISAM Interface

To use the IIP, the user must ensure that his existing ISAM programs meet the restrictions in the section *Restrictions in ISAM Interface Use.* He must convert each ISAM file to a key-sequenced VSAM file by defining the VSAM file and loading the ISAM file into it. He also must convert ISAM job control statements to VSAM job control statements. The ISAM programs do not have to be re-compiled or re-link-edited to use the IIP.

Converting an ISAM File to a VSAM File
To convert an ISAM file to a VSAM file, the user must first use the Access Method Services DEFINE command to define a key-sequenced VSAM file. The VSAM file may be defined on a volume(s) that already contain enough free VSAM data space for it or the data spaces may be defined along with the file (Unique file). The section *Job Control and VSAM* describes the job control statements needed for the DEFINE command. In *DOS/VS Access Method Services*, GC33-5382, the DEFINE command is described.

The information from several DTFIS parameters must be specified in the DEFINE command:

DTFIS parameter	VSAM DEFINE parameter
HOLD=YES	Specify SHAREOPTIONS (4)
KEYLEN=n and KEYLOC=n	Specify KEYS (length position). *length* should always be set to KEYLEN. *position* should be set to (1) KEYLOC if RECFORM=FIXBLK in the DTFIS or (2) 0 if RECFM=FIXUNB.
RECSIZE=n	Specify RECORDSIZE (average maximum). The average and maximum values must be equal. If RECFORM=FIXBLK IN THE DTFIS, RECORDSIZE should be set to RECSIZE. If RECFORM=FIXUNB, RECORDSIZE should be set to RECSIZE + KEYLEN.
VERIFY=YES	Specify WRITECHECK.

The BUFFERSPACE parameter in the DEFINE command specifies how much space VSAM will have for I/O buffers. If the user does not specify the BUFFERSPACE parameter, the default is at least two data buffers and one index buffer. If the default is taken, one index buffer and two or more data buffers will be allocated if IOROUT=LOAD, ADDRTR, or RETRVE during IIP processing and two data buffers and one or more index buffers will be allocated if IOROUT=ADD. However, for better performance, the user can specify space for more than two data buffers and index buffers.

After he defines the VSAM file, the user must load the VSAM file by copying the ISAM file into it. He may use his ISAM load program, by way of the IIP, or he may use the Access Method Services REPRO command. In *DOS/VS Access Method Services*, GC33-5382, the REPRO command is described. If the user has records marked for deletion in the ISAM file and does not want them copied into the VSAM file, he should use his ISAM load program. The REPRO command will copy all records from the ISAM file, including those marked for deletion.

Figure 4.30 summarizes converting indexed sequential files to key-sequenced files and processing them with either programs that have been converted from ISAM to VSAM or with programs that still use ISAM.

Converting ISAM Job Control Statements

The job control statements for ISAM must be replaced by VSAM job control statements. An example of VSAM job control statement used with an ISAM program is shown below:

```
// JOB      PROCESS A VSAM FILE
// ASSGN    SYS001,X'190'
// DLBL     IFN,'MSTRFILE',,VSAM
// EXTENT   SYS001,321942
// EXEC     ISAMPGM,SIZE=nnK
   CSECT
   .
   .
IFN DTFIS   ..........
   .
   .
   END
/*
/&
```

The type code *VSAM* causes the ISAM Interface Program to be called. The same VSAM job control statements are required regardless of the type of ISAM program. One DLBL statement is required for the file and an EXTENT statement is required for each volume of the file. See the section *Job Control and VSAM* for more information on how to write job control statements for VSAM.

What the ISAM Interface Program Does

When a processing program that uses ISAM opens a VSAM file, the DOS/VS OPEN routine detects the need for the ISAM Interface Program (by a type code of *VSAM* in the DLBL statement). It calls the IIP OPEN routine to construct control blocks (ACB and RPL) required by VSAM and an AMDTF used by the IIP processing routines, load the ISAM command processor, and flag the DTFIS for the IIP to intercept ISAM requests.

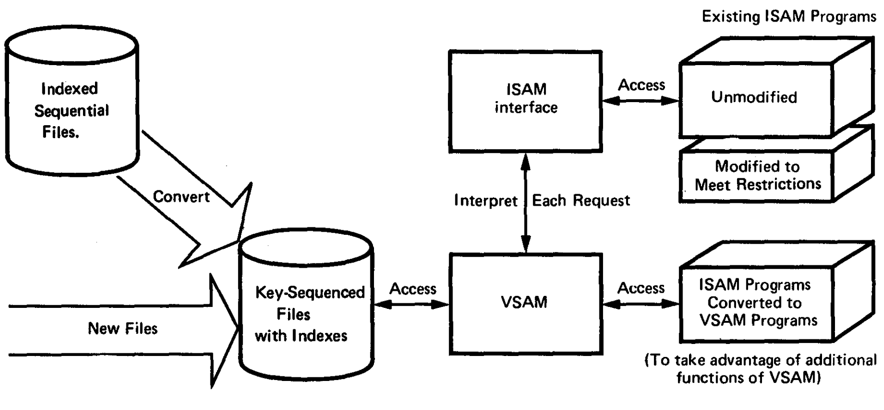

Existing ISAM Programs
Indexed Sequential Files.
Convert
ISAM interface
Access
Unmodified
Modified to Meet Restrictions
Interpret Each Request
New Files
Key-Sequenced Files with Indexes
Access
VSAM
Access
ISAM Programs Converted to VSAM Programs
(To take advantage of additional functions of VSAM)

Figure 4.30. Using the ISAM interface

Most existing programs that use ISAM can process VSAM files through the interface with little or no change.

The IIP intercepts each subsequent ISAM request, analyzes it to determine the equivalent keyed VSAM request, which it defines in the Request Parameter List constructed by OPEN, and then initiates the request.

The IIP interprets the return codes from VSAM. If the VSAM condition corresponds to an ISAM condition, the respective bit in the filenameC byte in the DTFIS is turned on. For unrecoverable errors that cannot be posted in the filenameC byte, a message is printed, the VSAM file is closed (by the VSAM CLOSE routine), and the job is terminated. If a physical (I/O) error occurs and ERREXT=YES was specified in the DTFIS, the interface transfers additional error information to the processing program. The contents of the filenameC byte and the ERREXT parameter list with IIP are described in *DOS/VS Supervisor and I/O Macros*, GC33-5373.

Restrictions in ISAM Interface Program Use

The ISAM Interface Program enables programs that use ISAM to issue only those requests that VSAM or the interface can simulate. These are the restrictions for using the interface routine:

- Record ID processing of ISAM cannot be used because VSAM does not use the record ID function.
- VSAM does not return (1) device-dependent information or (2) the virtual storage or DASD address of the record containing the error in the ERREXT parameter list.
- The ISAM program cannot open a DTF while another DTF is already open for the same file unless Sharing Option 4 is specified for the VSAM file.

Optimizing the Performance and Storage of VSAM

This section explains VSAM options that affect performance and virtual storage and direct access storage requirements: the size of the control interval, the percentage of distributed free space, the division of key-sequenced data into key ranges, and the handling of indexes. These options are specified in the DEFINE command of Access Method Services when a file is created. Other options that affect performance and data integrity are described in the following section *Data Security and Integrity.* This section also discusses statistics about the file that VSAM makes available to the user.

Control Interval Size

The user can let VSAM select the size of a control interval or he can request a particular size through the CONTROLINTERVALSIZE parameter of the DEFINE command. The size requested, however, must fall within acceptable limits determined by VSAM. These limits depend on (1) the least amount of space VSAM can use for I/O buffers, (2) the device type, and (3) the size of the data records. The user can issue the LISTCAT statement of Access Method Services or the SHOWCB macro instruction to find the control interval size used by VSAM.

A file's control interval size affects performance and storage requirements as follows:

- As the size of his logical records increases, the user may need a larger control interval because logical records cannot span control intervals.
- As control interval size increases, more buffer space is required for each control interval.
- As control interval size increases, fewer I/O operations (control interval accesses) are required to retrieve a given number of records and fewer index records must be read. This is usually more significant for sequential and skip sequential processing.
- Free space will probably be used more efficiently (fewer control interval splits and less wasted free space) as the size of the control interval increases relative to the size of the logical records. This is especially true with variable length records. Free space in a control interval will not be used if there is not enough for a complete logical record.

The control interval size must be a multiple of 512 bytes because physical block size, chosen by VSAM, is 512 bytes, 1024 bytes, 2048 bytes, or 4096 bytes (3330 and 3340 only) and 8192 bytes (3340 only). If the user specifies a control interval size that is not a multiple of 512 bytes, VSAM will increase it to the next higher multiple of 512 bytes. The physical block size will be set as large as possible. For example, if control interval size is 1024 bytes, the physical block size will be 1024 bytes; if control interval size is 1536 bytes, the physical block size will be 512 bytes. If the control interval size is larger than 8192 bytes, it must be a multiple of 2048 bytes.

A control area can never be larger than one cylinder of the device. If the user allocates space in the DEFINE command in terms of CYLINDERS, or if the file is Unique, VSAM will set the control area size to one cylinder. VSAM will also set the control area size to one cylinder if the user allocated space in terms of RECORDS or TRACKS and VSAM can make the control area a cylinder without violating any other allocation rules. If the control area is less than a cylinder, it will be an integral number of tracks in size and it can extend across cylinders. However, a control area can never extend across an extent on the device.

The user can specify the size of the index control interval (index record) but, since it must equal block size, the only sizes available are 512 bytes, 1024 bytes, 2048 bytes, and 4096 bytes (not for the 2314).

I/O-buffer size is important because VSAM transmits the contents of a control interval, and the amount of virtual storage space for I/O buffers limits the size of a control interval. The amount of space for I/O buffers is the most flexible variable for influencing control interval size. The size and other attributes of the data records generally depend on the needs of the application. If only one request at a time can be active, VSAM requires a minimum of three buffers for key-sequenced files, two for data control intervals and one for an index control interval, and two buffers for an entry-sequenced file. If concurrent requests can be active, one additional data buffer and one additional index buffer (for key-sequenced files) are required for each additional request. If you do not specify a minimum buffer space in the BUFFERSPACE parameter of the DEFINE command, VSAM defaults to enough buffer space for the minimum number of buffers required.

Distributed Free Space

In the section *VSAM File Structures,* we discussed the way VSAM uses distributed free space when a record is inserted into a key-sequenced file. We said that records can be inserted in a file that does not have any distributed free space, by means of a control area split. Free space in the immediate area into which a record is inserted speeds up the insertion and avoids control area splitting, which may move a group of records to a different cylinder, away from the preceding and following records in key sequence. The percentage of free space is specified in the FREESPACE parameter of the DEFINE command. Both the percentage of free space in a control interval and the percentage of free space in a control area (number of free control intervals) can be specified.

How much space must be provided? That dpends on how many (and where) records will be inserted, lengthened or deleted. If the user does not provide enough free space, additional processing time will be needed to split control intervals or control areas and to process the records sequentially when they are physically out of sequence. If he provides too much free space, more direct access storage than necessary will be used to contain the file and extra control interval accesses will be needed in sequential processing to read the same number of records. Of course, if the file is for reference only, it might not need any free space. When estimating file growth, remember that (1) when records in a key-sequenced file are deleted or shortened, the space freed is available as free space and (2) each control interval or control area resulting from a split contains about 50% free space. The user will have to determine the best tradeoff between performance and direct access storage requirements.

Index Options

Four options influence performance and storage requirements through the use of the index with a key-sequenced file. Each option can improve performance, but some of them require additional virtual or direct access storage space. The options are:

- Number of index records in virtual storage
- Index and file on separate volumes
- Sequence-set records adjacent to the file
- Replication of index records.

Index-Set Records in Virtual Storage

For keyed access, VSAM needs to examine the index of a file. Before the processing program begins to process the file, it must specify the amount of space VSAM can use to buffer index records. Enough space for one index record is the minimum; but, when the space is large enough for only one or two index records, an index record may be continually deleted to make room for another and then retrieved again later when it is required. Ample space to buffer index records can improve performance by preventing this situation provided that the buffer allocation does not cause excessive paging by DOS/VS. Remember that VSAM searches the sequence set for sequential access and every index level for direct access.

The user can ensure that index records will be in virtual storage by specifying enough space for I/O buffers for index records through the BUFNI and BUFSP parameters of the ACB when he begins to process a file. VSAM keeps as many index-set records in virtual storage as the space will hold. Whenever an index record must be retrieved to locate a record, VSAM makes room for it by deleting from the space the index record that VSAM judges to be the least useful under the circumstances then prevailing. It is generally the index record that belongs to the lowest index level then represented in the space which has been unused the longest.

Index and Data on Separate Volumes

When a key-sequenced file is defined, the entire index or the index set alone can be placed on a separate volume from the data, either on the same or on a different type of storage device. Data and index of a cluster (file) are defined separately and the volume that is to contain each is specified in the VOLUMES parameter of the DEFINE command. Only the index set is placed on the separate volume if the sequence set is imbedded with the data as described below.

Using different volumes can eliminate the disk arm contention between accessing index records and accessing data records when using keyed access.

Sequence-Set Records Adjacent to the File

Having the sequence set adjacent to the file is one way to reduce disk arm movement for a key-sequenced file. When the user defines the file, he can use the IMBED parameter of the DEFINE command to specify that the sequence set index record for each control area is to be on the first track of the control area. This avoids two separate seeks when access to a data record requires VSAM to examine the sequence-set index record of the control area in which the data record is stored. One arm movement enables VSAM to retrieve or store both the index record and the contents of the control interval in which the data record is stored. However, if some control areas extend across cylinders (see the preceding section on *Control Interval Size*), this option may not eliminate separate seeks for the sequence set record and the data. When this option is taken, sequence set records are replicated, as described next.

Replication of Index Records

By specifying the REPLICATE parameter of the DEFINE command, the user can have each index record written on a track of a direct access volume as many times as it will fit. Replication reduces the time lost waiting for the index record to come around to be read (rotational delay). Average rotational delay is half the time it takes for the volume to complete one revolution. Replication of a record reduces this time. For instance, if ten copies of an index record fit on a track, average rotational delay is only one-twentieth of the time it takes for the volume to complete one revolution.

This option costs direct access storage space; it requires a full track of storage for each index record replicated. The user has to weigh the relative values of direct access storage space and processing speed.

Index records can be replicated in these combinations of sequence set and index set:

- Sequence set separated from index set and only sequence set records replicated (IMBED).
- Sequence set separated from index set but all index records replicated (IMBED, REPLICATE).
- Sequence set and index set together and all index records replicated (REPLICATE).

Separating the sequence set from the index set is for placing the sequence set adjacent to the data, which is the previous option we discussed. Figure 4.31 illustrates replication of a sequence set record that has been placed adjacent to its control area.

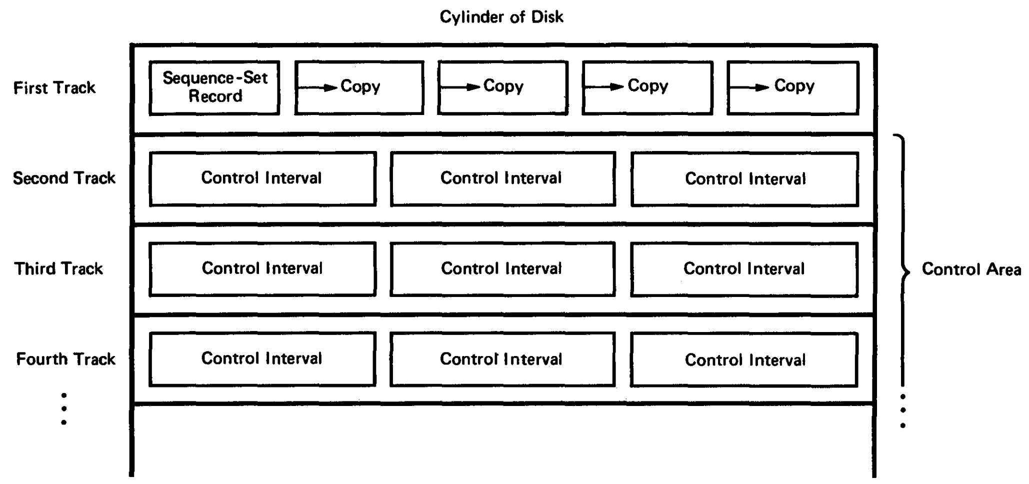


Figure 4.31. Sequence set record placed adjacent to control area to reduce disk arm movement and replicated to reduce rotational delay.

Key Ranges

The records of a key-sequenced file can be grouped on volumes according to key ranges. A payroll file, for example, could have employee records beginning with A, B, C, and D on one volume; E, F, G, H, and I on a second volume; etc. Each portion of a multivolume file can be on a specified volume. Each keyrange of a file, as well as the end of the file, is preformatted as described under *Data Security and Integrity.* The KEYRANGES option in the DEFINE command of Access Method Services establishes key ranges for the file.

File Statistics

VSAM keeps certain statistical information about a file in its catalog entry. These statistics, on file size and activity, can help the user decide when to reorganize the file or what processing modes to use.

The entire catalog entry, the statistics and the parameters selected when the file was defined, can be listed by using the LISTCAT command of Access Method Services. The SHOWCB and TESCB macros can be used by a processing program to display or test one or more file statistics. These statistics include:

- Control-interval size
- Percentage of free control intervals per control area
- Number of bytes of available space (includes distributed free space and allocated space beyond the last record)
- Length and displacement of the key
- Maximum record length
- Number of buffers
- Number of records
- Password
- A timestamp that indicates if either the data or the index has been processed separately
- Number of levels in the index
- Number of extents
- Number of records retrieved, added, deleted, or updated
- Number of control interval splits in the data and in the sequence set of the index
- Number of EXCP commands issued.

Data Security and Integrity

How safe is data with VSAM? What provisions does VSAM make to ensure that data is not lost or destroyed by errors in the system, or accessed by unauthorized persons? How easy is it to determine what the cause of a problem is and to do something about it? This section is intended for installation managers and system programmers interested in the answers to these questions.

The protection of data includes data integrity, or the safety of data from accidental destruction, and data security, or the safety of data from theft or intentional destruction. We will discuss the attributes and options of VSAM that ensure data integrity, the protection of shared data, the use of passwords to prevent unauthorized access to data, and the methods of determining the causes of problems.

Data Integrity

The attributes and options of VSAM that affect data integrity are:

- Method of inserting records into a key-sequenced file
- Control-interval principle
- Method of indicating the end of a file
- Verifying write operations.

Method of Inserting Records into a Key-Sequenced File

We discussed the method of inserting new records into a key-sequenced file in the section *VSAM File Structures*. Free space distributed throughout used control intervals allows VSAM to insert a record into a control interval by shifting records in it without an I/O operation. VSAM splits control intervals and control areas, when necessary, in a way that does not expose any data to loss, even if an I/O error occurs before the split is completed.

If an I/O error does occur during a split, however, some records may be duplicated. The second copy of a duplicated record may be retrieved during sequential processing. However, the first copy is always the most recent one. Duplicate copies of records may not be retrievable if the control interval or control area is split again.

Control-Interval Principle

With a key-sequenced file, the control interval is the unit pointed to by entries in a sequence-set index record. Only a record addition or a record insertion that splits a control interval or a control area causes a modification of the index. For instance, even though a record insertion might change the RBA of the record with the highest key in the control interval, the index entry is not altered, since the pointer in it is to the control interval, not to the record.

Method of Indicating the End of a File

VSAM combines two procedures for achieving data integrity:

- Preformatting the last control area of a file or a key range
- Updating the catalog to indicate (1) the RBA of the end of the file, and (2) the highest keyed record in the file.

Preformatting a File

Preformatting the end of a file as each control area comes into use ensures greater data integrity than formatting it only at the end of processing. If a key-sequenced file is divided into key ranges, the end of each key range is preformatted. VSAM formats a control area before using its control intervals by putting control information in them and putting an end-of-file indicator after the last control interval. The end-of-file indicator helps prevent the loss of data that has been added to the end of a file.

VSAM optionally preformats control areas when initially loading records into a file and always preformats them when subsequently adding records to the file. Whether he uses the REPRO command of Access Method Services or his own processing program, The user has two options when initially loading records into a file: SPEED or RECOVERY.

- The SPEED option improves load speed: VSAM does not format the last control area of a file until the file is closed.
- The RECOVERY option improves the ability to recover from a failure to complete loading. Each time a control area is filled with records, VSAM formats the next control area before storing records in it. In this way each set of new records is protected against loss by the end-of-file indicator as it is added to the file.

Updating the Catalog

The addresses kept by the catalog for the end of the file enable VSAM to keep track of the physical end and, for a key-sequenced file, the logical end of the file. VSAM updates these addresses either (1) when it assigns and preformats a new control area or index control interval (Sharing Option 4 only) or (2) when the processing program issues either a TCLOSE macro instruction or, at the end of file processing, a CLOSE macro instruction. This depends on the preformat option the user chooses when he initially loads his records into the file. By using the VERIFY command of Access Method Services, he can recover data in cases where VSAM was unable to close a file properly and update the end-of-file indicator in the catalog. See the discussion of the VERIFY command in the Access Method Services section.

Verifying Write Operations
To improve the integrity of data written to direct access storage, the user can request VSAM to verify each write operation for accuracy. Verification takes additional time, but it decreases the chance of I/O error during subsequent retrieval.

Protection of Shared Data

A VSAM file can be shared by processing programs in two or more partitions. If a file is shared (Sharing Options 3 and 4), more than one user can OPEN it at the same time to update or add records. If the file is not shared, only one user at a time can OPEN it to update or add records. Any number of users can retrieve records from the file regardless of whether it is shared or not. The degree of sharing to be allowed for the file is specified, when the file is defined, in the SHROPT parameter of the DEFINE command of Access Method Services. The SHROPT parameter can be changed by the ALTER command of Access Method Services. One of the following file sharing options can be specified:

- Sharing Option 1: The file may be opened by any number of users for input processing (retrieve records) or it can be opened by one user for output processing (update or add records). This option maintains full (read and write) integrity.
- Sharing Option 2: The file may be opened by more than one user for input processing *and,* at the same time, it may be opened by one user for output processing. This option maintains write integrity but, since the file might be modified while records are being retrieved from it, each user must maintain his own read integrity.

- Sharing Option 3: The file can be opened by any number of users for both input and output processing. VSAM does nothing to maintain either the write integrity of the file or the read integrity of the users.
- Sharing Option 4: A key-sequenced file can be opened by any number of users for both input and output processing. VSAM maintains write integrity by using the trackhold facility of DOS/VS. Read integrity will be maintained by VSAM only when records are being retrieved for update. If records are not being retrieved for update, some records in control intervals being updated concurrently by more than one user may be skipped by VSAM because each user might retrieve a different copy of the control interval. Each task can issue requests from only one ACB per file at any given time. Also, an ACB can be opened by only one task at a time. It must be closed before another task can use it.

Data Security

Access Method Services provides options to protect a file or the VSAM catalog against unauthorized use. These options, specified when a file (or the VSAM catalog) is defined, include passwords and a user-written security verification routine. They can be altered by using Access Method Services.

Passwords

A password, if required, can be supplied by the processing program in a field pointed to by the ACB. If the processing program does not supply the password, it must be supplied by the operator. If the catalog requires a password, it must be issued each time a job requiring Access Method Services is started. The data and index of a cluster can have different passwords. Either the password of the data or index or the password of the cluster can be used when the data or index is opened separately. Four different types of passwords, for different degrees of data security, can be specified in the DEFINE command:

- Full access (MASTERPW parameter). This is the master password, which allows access to a file and any index and catalog entry associated with it for all operations (retrieving, updating, inserting, deleting). Using this password to gain access to a catalog entry gives the user the ability to delete an entire file and to alter password information or any other information in the catalog about a file, index, or catalog.
- Update access (UPDATEPW parameter). This password authorizes retrieval update, insertion, or deletion of records in a file. It gives limited access to catalog entries to define objects but not to alter their definitions or to delete entries.
- Read access (READPW parameter). This is the read-only password, which allows the user to examine data records and catalog entries, but not to add, alter, or delete them.
- Control-interval access (CONTROLPW parameter). This password authorizes processing the file by control interval access.

Two other options can be specified in the DEFINE command for use when the operator supplies a password. The ATTEMPTS option specifies how many times, 0 through 7, the operator can attempt to supply the correct password. If 0 is specified, passwords cannot be supplied by the operator.

If the ATTEMPTS option is not used, the default (2) allows the operator to attempt to supply the password twice. The CODE option specifies a one-to-eight character name, other than the name (file-ID) of the file, to which the operator responds with a password. This *prompting code* helps keep data secure by not allowing the operator to know both the name of the file and its password. If the CODE option is not specified, the name of the job and the name (file-ID) of the file is supplied to the operator.

If the processing program supplies the wrong password or the operator cannot supply the correct password in the allowed number of attempts, the job is terminated. An error code is set in the ACB indicating that the file cannot be opened because the correct password was not supplied.

Security Verification Routine.
If the owner specifies password protection when the file or VSAM catalog is defined, he can also supply a routine to check the authority of a processing program to access the file or the catalog. The AUTHORIZATION option specifies the name of the user's verification routine. VSAM transfers control to the verification routine when the program trying to open the file gives the correct password. If the master password (MASTERPW) is given, the authorization routine is bypassed. The authorization option can also include up to 256 bytes of information which will be passed to the authorization routine when it is called. When the authorization routine gets control from VSAM, the registers are set as follows:

Register	Contents	
0	Unpredictable	
1	Address of a parameter list:	
	44 bytes	Name of the file to be opened
	8 bytes	Prompting code (from the CODE option) or zeros
	8 bytes	File owner identification (from the OWNER parameter of DEFINE)
	2 bytes	Length of string passed to routine that was specified in the AUTHORIZATION parameter
	n bytes	The authorization string (up to 256 bytes)
2-13	Unpredictable	
14	Return address to VSAM	
15	Entry point to verification routine.	

When the authorization returns to VSAM, register 15 should be set to zero if the processing program is authorized to access the file or catalog. If register 15 is not zero, VSAM will not allow the processing program to open the file.

Determining the Causes of Problems

VSAM offers several diagnostic aids for the user to determine the cause of errors.

Exits to the Error-Analysis Routines

VSAM provides optional exits to a user-supplied routine to handle I/O errors or a user-supplied routine to handle logical errors. The EXLST macro instruction is used to specify the names of the user's exit routines. The exit routines can examine return codes in the ACB or RPL to determine the cause of the error. Figure 4.32 shows the conditions under which the user exit routines are given control.

VSAM Messages

Like other access methods, VSAM issues messages to the operator if an incorrect volume is mounted or a subsequent volume of a multivolume file must be mounted. VSAM also issues messages to the operator if an error occurs during catalog processing, a file was not closed during previous processing, or when the operator must supply a password so that a file can be opened. The ISAM interface issues messages to the operator when errors occur while using an ISAM program to access a VSAM file. These messages are described in *DOS/VS Messages.* Access Method Services issues messages to the programmer. They are documented in *DOS/VS Access Method Services,* GC33-5382.

VSAM Return Codes

Most errors detected by a VSAM processing program are indicated by *return codes* which are set in Register 15 following execution of a macro instruction. If an error occurs, additional information will be indicated by an *error code* which is set in either the Access-Method Control Block (ACB), the Request Parameter List (RPL), or Register 0 depending on the macro. If the return code indicates an error, the appropriate user-supplied exit routine will be taken if it is active. The exit routine can examine the error code to determine what error occurred and how to handle it. Return codes and exit routines are described in detail in the VSAM chapter of *DOS/VS Supervisor and I/O Macros,* GC33-5373. The error checking that a user program must do after execution of an imperative VSAM macro instruction is summarized in Figure 4.32.

Imperative Macro	Return Code found in	If Return Code not 0, Error Code found in	Method of Inspecting Error Code	Exits Taken if User Exit Routines are Supplied
OPEN CLOSE TCLOSE	Register 15	ACB	Code ERROR parameter in TESTCB or SHOWCB	SYNAD exit for I/O errors in CLOSE; code FDBK in TESTCB or SHOWCB
GET PUT POINT ERASE	Register 15	RPL (If Return Code is not X'00' or X'04')	Code FDBK parameter in TESTCB or SHOWCB	LERAD exit for logical errors (Return Code is X'08')* SYNAD exit for I/O errors (Return Code is X'0C')
GENCB SHOWCB TESTCB MODCB	Register 15	Register 0	Inspect Register 0	None

*End-of-file is indicated by an error code of X'04'. The EODAD exit is taken if an EODAD routine is supplied. Otherwise, the LERAD exit is taken.

Figure 4.32. Summary of Error Checking for VSAM Imperative Macros

Section 5: APPENDIXES

Appendix 1: Devices Supported by DOS/VS

Punched Card Devices

The following punched card equipment is supported by DOS/VS:

- IBM 1442 Card Read Punch Model N1
- IBM 1442 Card Punch Model N2
- IBM 2501 Card Reader Models B1 and B2
- IBM 2520 Card Read Punch Model B1
- IBM 2520 Card Punch Models B2 and B3
- IBM 2540 Card Read Punch Model 1
- IBM 2560 Multifunction Card Machine Model A1
- IBM 2596 Card Read Punch
- IBM 3504 Card Reader Models A1 and A2
- IBM 3505 Card Reader Models B1 and B2
- IBM 3525 Card Punch Models P1, P2, and P3
- IBM 5425 Multifunction Card Unit Models A1 and A2

IBM 1442 Card Read Punch Model N1

The IBM 1442 Card Read Punch Model N1 is a combined I/O unit for punched cards, which reads at a rated speed of 400 cards per minute, and punches at a rated speed of 160 columns per second. It has two stackers; cards go to stacker 1, unless directed to stacker 2 by the program. The 1442 Model N1 contains its own control unit.

IBM 1442 Card Punch Model N2

The IBM 1442 Card Punch Model N2 punches cards at a rated speed of 160 columns per second. It has one stacker, and it contains its own control unit.

IBM 2501 Card Reader Models B1 and B2

The IBM 2501 Card reader Model B1 reads punched card input at a rated speed of 600 cards per minute. The 2501 Model B2 reads punched card input at a rated speed of 1000 cards per minute. Each model has one stacker and contains its own control unit.

IBM 2520 Card Read Punch Model B1

The IBM 2525 Card Read Punch Model B1 is a combined punched card I/O unit, which reads and punches at a rated speed of 500 cards per minute. The device has one input hopper and two output stackers; output goes to stacker 1 unless directed to stacker 2 by the program. The 2520 contains its own control unit.

IBM 2520 Card Punch Models B2 and B3

The IBM 2520 Card Punch Model B2 punches cards at a rated speed of 500 cards per minute. The 2520 Model B3 punches at a rated speed of 300 cards per minute. Both models have two stackers; output goes to stacker 1 unless directed to stacker 2 by the program. The 2520 contains its own control unit.

IBM 2540 Card Read Punch

The IBM 2540 Card Read Punch is a combined punched card I/O unit which is connected to a computer system through the IBM 2821 Control Unit. Fully buffered card reading and punching is provided. The device has five stackers located between the read and punch feeds; the center stacker can be used for either feed.

The read feed operates at a rated speed of 1,000 cards per minute, or 800 cards per minute if the 51-colums interchangable read feed (special feature) is installed. The punch feed operates at a rated speed of 300 cards per minute. Cards in the read feed go to stacker R1 unless directed to stacker R2 or RP3 by the program. Cards in the punch feed go to stacker P1 unless directed to stacker P2 or RP3 by the program.

IBM 2560 Multifunction Card Machine

The IBM 2560 Multifunction Card Machine provides the combined functions of card reader/punch, collator, and, on a Model A1 with the Card Print special feature, card interpreter/document printer in one device. The device permits collating, gangpunching, reproducing, summary punching, punching of calculated results, printing and classifying of cards in one single pass of the cards. The read feed has a primary and a secondary hopper feeding cards individually at a rated speed of 500 cards per minute. The read feed contains pre-read, read, and pre-punch stations. The punch feed, following the read feed, contains a punch, pre-print, and print station. Cards follow a common path and are fed into one of the five radial stackers. The device includes its own control unit.

IBM 2596 Card Read Punch

The IBM 2596 Card Read Punch reads and punches 96-column cards. Reading is at a rated speed of 500 cards per minute; punching is at a rated speed of 120 cards per minute. If the 1510 card print (special feature) is installed the device provides printing of three lines (32 characters each) for data being punched. The read and punch feeds are separate. There are four stackers: normal read stacking (1), selective read stacking (2), normal punch stacking (3), and selective punch stacking (4). The 2596 contains its own control unit.

IBM 3504 Card Reader Models A1 and A2

The IBM 3504 Card Reader reads (fully buffered) at a rated speed of 800 cards per minute (Model A1), or 1,200 cards per minute (Model A2). Cards are read by means of light sensing. The device has two non-programmable stackers which operate in an alternating mode.

The 3504 attaches natively to a System/370 Model 125 CPU through an integrated 3504 Card Reader Attachment. The 3504 is a natively attachable version of the 3505 Card Reader (described below).

IBM 3505 Card Reader Models B1 and B2

The IBM 3505 Card Reader reads (fully buffered) at a rated speed of 800 cards per minute (Model B1), or 1,200 cards per minute (Model B2). Cards are read by means of light sensing. The device has two non-programmable stackers which operate in an alternating mode. An extra programmable stacker can be installed as a special feature. Other optional special features are:

- Optical Mark Read, for reading up to 40 columns of marked data.

- Read column eliminate, which provides the suppression of reading selected card columns, under program control.
- Adapters which enable an IBM 3525 Card Punch to be connected to a system through the 3505. If the 3525 is equipped with special features (see below), adapters for those special features are included in the 3505.

The 3505 contains its own control unit.

IBM 3525 Card Punch Models P1, P2, and P3

The IBM 3525 Card Punch is a full-function card punch which, when equipped with the appropriate special features, can *read* and *print* as well as *punch* 80-column cards in a single pass through the machine. It operates at a rated speed of 100 (Model P1), 200 (Model P2), or 300 (Model P3) cards per minute. The basic unit, as a card punch only, contains two stackers; cards go into stacker 1 unless directed to stacker 2 by the program.

Optional special features are:

- Card Read. This feature provides an optical hole sensing station ahead of the punch station.
- Card Print. This feature provides a print station following the punch station. There are two versions of the card print special feature: one version provides a possibility to print up to and on any of 25 lines on a card, under program control; the other version is limited to two lines. Each line contains 64 printing positions.

Adapters for the special features are to be installed in the IBM 3505 Card Reader (see above), which acts as the control unit for the 3525.

IBM 5425 Multifunction Card Unit Models A1 and A2

The IBM 5425 Multifunction Card Unit can be used as a card reader, card punch, and card printer/interpreter. Its functions are basically the same as described for the 2560 Model A1.

The 5425 processes 96-column cards. In addition, it is supported by DOS/VS as an 80-column card device for all system functions that require card I/O.

Model A1 of the 5425 reads 250 cards and punches 60 cards per minute; Model A2 reads 500 cards and punches 120 cards per minute.

The 5425 includes its own control unit; it can be natively attached to a System/370 Model 115 or Model 125.

Printers

The following line printers are supported by DOS/VS:

- IBM 1403 Printer Models 2, 3, 7, and N1
- IBM 1443 Printer Model N1
- IBM 3203 Printer Models 1 and 2
- IBM 3211 Printer Model 1
- IBM 3213 Console Printer
- IBM 5203 Printer Model 3
- IBM 5213 Console Printer Model 1

IBM 1403 Printer Models 2, 3, 7, and N1

The IBM 1403 Printer Models 2 and 7 operate at a maximum speed of 600 lines per minute. Models 3 and N1 operate at a maximum speed of 1,100 lines per minute. A print line is 120 characters on the Model 7, or 132 characters on the models 2, 3, and N1. Each print position can print any of 48 characters. Forms spacing and skipping are governed by a 12-channel tape in the carriage.

All models of the 1403 are connected to a system through the IBM 2821 Control Unit.

IBM 1443 Printer Model N1

The IBM 1443 Printer Model N1 operates at a rated speed of 240 lines per minute, with a 52-character set which includes 16 special characters. Other character sets are available: if used, the operating speed may increase or decrease, depending on the character set used. A print line may consist of 120 characters. The device contains its own control unit.

IBM 3203 Printer Models 1 and 2

The IBM 3203 Printer Model 1 operates at a maximum speed of 600 lines per minute; the maximum speed of the Model 2 is 200 lines per minute. For both models, a 48-character set and the UCS (Universal Character Set) feature are standard. The UCS feature allows the use of character sets other than the standard one. Each print line has a standard capacity of 132 print positions. Form spacing and skipping is controlled by a forms control buffer.

Both models of the 3203 printer can be natively attached to a System/370 Model 115 or 125.

IBM 3211 Printer Model 1

The IBM 3211 Printer Model 1 operates at a rated speed of 2,000 lines per minute. A 48-character set is standard. Each print line has a capacity of 132 character positions, and another 18 positions are available as a special feature. Continuous marginally punched form spacing and skipping is controlled by a forms control buffer.

The 3211 is connected to a system through the IBM 3811 Control Unit. An IBM 3216 Interchangable Train Cartridge must be used on the 3211. This allows maximum flexibility for varying character sets, each set operating at maximum speed.

IBM 3213 Console Printer

The IBM 3213 Console Printer is a serial printing device, printing at a maximum rated speed of 85 characters per second. Each print line has a maximum capacity of 126 characters. The 88-character set of PTTC/EBCD is standard. The device contains its own control unit.

IBM 5203 Printer Model 3

The IBM 5203 Printer Model 3 operates at a maximum speed of 300 lines per minute. It has a standard 48-character set and an optional UCS (Universal Character Set) feature. The UCS feature allows the use of character sets other than the standard one. Each print line has a standard capacity of 96 print positions; this standard carriage can be replaced by a wider one with 120 or 132 print positions. Form spacing and skipping is controlled by a forms control buffer. The 5203 printer with 96 print positions cannot be used as SYSLST device.

The 5203 printer can be natively attached to a System/370 Model 115.

IBM 5213 Printer Model 1

The IBM 5213 Printer Model 1 is a serial printing device, printing at a rated speed of 85 characters per second. A print line can contain 132 characters. The device contains its own control unit.

Optical Readers

The following optical readers are supported by DOS/VS:

- IBM 3881 Optical Mark Reader Model 1
- IBM 3886 Optical Character Reader Model 1.

IBM 3881 Optical Mark Reader Model 1

The IBM 3881 Optical Mark Reader Model 1 reads handwritten or machine printed marks on paper documents. The documents can range in size from 3 inches by 3 inches to 9 inches by 12 inches. The 3881 can read up to 40 marking positions across an 8 1/2 inch document and up to 6 rows per inch vertically. An 8 1/2 inch by 11 inch document can contain up to 2480 marking positions. Approximately 4,000 8 1/2 inch by 11 inch documents can be read per hour. The maximum size of the output record is 900 bytes.

IBM 3886 Optical Character Reader Model 1

The IBM 3886 Optical Character Reader is a general-purpose optical reader. It can recognize data created by:

- Numeric handprinting
- High speed computer printing
- Typewriters
- Preprinting on forms

Data can be provided to your program exactly as it was read from the document or it can be edited and formatted. Document processing and the data provided to the problem program is controlled by a format record that is loaded into the 3886 when the file is opened. Line and page numbering can also be performed if the feature is included on the device. When a document has been read, it can be routed to the normal stacker or to the reject stacker if an error was found on the document.

Magnetic Tape Devices

The following magnetic tape devices are supported by DOS/VS:

- IBM 2401 Magnetic Tape Unit Models 1-6, and 8
- IBM 2415 Magnetic Tape Unit and Control Models 1-3, and 4-6
- IBM 2420 Magnetic Tape Unit Models 5 and 7
- IBM 2495 Tape Cartridge Reader Model 1
- IBM 3410 Magnetic Tape Unit Models 1, 2, and 3
- IBM 3411 Magnetic Tape Unit Models 1, 2, and 3
- IBM 3420 Magnetic Tape Unit Models 3, 5, and 7

The major characteristics of these devices are shown in Figure 5.1.

	Maximum Data Rates		Control Unit
	Kilobytes per second	Bytes per inch	
IBM 2401, M. 1	30	800	2803 or 2804
2	60	800	
3	90	800	
4	60	1,600	
5	120	1,600	
6	180	1,600	
8	60	800	
IBM 2415, M. 1–3	15	800	*
4–6	30	1,600	
IBM 2420, M. 5	160	1,600	2803
7	320	1,600	
IBM 2495, M. 1	.9	20	*
IBM 3410, M. 1	20	1,600	**
/3411 2	40	1,600	
3	80	1,600	
IBM 3420, M. 3	120	1,600	3803
5	200	1,600	
7	320	1,600	

* Control Unit is included in the device.
** A first tape drive in a sequence of 1410's must be a 3411 which is a 3410 with a control unit.

Figure 5.1. Characteristics of magnetic tape devices supported by DOS/VS

Diskette Device

The IBM 3540 Diskette Input/Output Unit is supported as a sequential access device. The major characteristics of this device are shown in Figure 5.2.

TRACKS PER VOLUME	77
o Track 0	System Use
o Tracks 1-73	Data Records
o Track 74	Reserved
o Track 75-76	Alternates for Defective Tracks
RECORDS PER TRACK	26
BYTES PER RECORD	128
BYTES PER TRACK	3,328
BYTES PER VOLUME	242,944

Figure 5.2 Diskette layout and storage capacity

Direct Access Devices

The following direct access devices are supported for data file residence by DOS/VS:

- IBM 2311 Disk Storage Drive
- IBM 2314 Direct Access Storage Facility
- IBM 2319 Disk Storage
- IBM 3330 Family of Disk Storage Devices
- IBM 3340 Disk Storage
- IBM 2321 Data Cell Drive

All DASD devices have the following characteristics in common:

- Each physical block stored on a DASD has a discrete location and a unique address.
- Data is stored on DASD in such a way that any block can be located without extensive searching.
- Data can be accessed directly rather than serially.
- If desired, data can also be processed sequentially.
- When DASD data is updated, the updated version of a physical block can physically replace the old one.

The IBM DASD devices feature removable, interchangable disk packs. A disk pack can be easily removed and replaced by another one, in less than a minute. These units have flexibility comparable to a tape system, plus the advantage of direct access processing.

On the IBM 2321 Data Cell Drive, data is stored on magnetically coated strips. Two hundred strips are contained in a cell assembly, which is removable and interchangeable.

The main characteristics of the direct access devices listed above are described in Appendix 2.

IBM 2311 Disk Storage Drive

Access Mechanism
Ten read/write heads are mounted on a vertical assembly. The heads are aligned vertically and are all moved together to any of 203 positions. Therefore, each time the read/write heads are moved to some position, one entire cylinder of ten data tracks is accessible for reading and writing. Only electronic switching between the read/write heads is necessary to select a particular track within a cylinder.

Storage and Record Capacity
The 7.25-million byte capacity of each volume is based on 200 tracks per disk surface. The remaining three tracks per surface are used as alternate tracks for defective tracks.
Up to eight 2311 drives can be connected to one storage control unit.

IBM 2314 Direct Access Storage Facility

The IBM 2314 consists of eight online disk storage modules, one spare (offline) module, and a control unit. The spare module is available for immediate use if servicing or routine maintenance is necessary for any of the other modules. Of the total of nine modules, any eight can be online at a time.

Access Mechanism
Twenty read/write heads are mounted on a vertical assembly, one for each module. Each module has its own access mechanism which operates independently. The heads of a particular module are aligned vertically and are all moved to any of 203 positions. Therefore, each time the read/write heads are moved to some position, one entire cylinder of twenty data tracks become available for reading or writing.

Storage and Record Capacity
Information is read from, or written to disk surfaces of the IBM 2316 Disk Pack. The 29.17-million byte capacity of one module is based on 200 tracks per surface. The remaining three tracks are used as alternate tracks to replace defective tracks.

IBM 2319 Disk Storage

The IBM 2319 Disk Storage is a high-speed direct access storage facility for System/370 Models 135 and 145 users. Its function and performance are identical to the IBM 2314, but its configuration consists of only three modules per 2319 Model A1. This configuration can be expanded to six or nine modules (only eight online at a time), by adding one or two 2319 Model A2s. A 2319 Model A2 is attachable only to a 2319 Model A1, a 2319 Model A1 attaches to the System/370 Model 145 through the Integrated File Adapter.

The IBM 2319 Disk Storage Models B1 and B2 are high-speed direct access storage facilities for System/360 and System/370 users. The configurations available are identical to those for the 2319 Models A1 and A2, except that the Models B1 and B2 are attached to a system through the IBM 2314 Model B1 Storage Control Unit.

Functionally, the 2319 (all models) is identical to the 2314.

IBM 3330 Family of Disk Storage Devices

The IBM 3330 Family of Disk Storage Devices offers the user a high-speed direct access storage facility with a capacity of two to sixteen disk drives, each drive having a data capacity of 100 million bytes. It consists of the following components:

- IBM 3333 Disk Storage and Control.
  A 3333 Disk Storage Module consists of two independent disk drives. It may act as the control unit for one to three additional 3330 modules (see below) or it may be used individually. One 3333 module may, with or without 3330 modules, directly be connected to a system; two 3333 modules may, with or without 3330 modules, be connected to a system via the 3830 Control Unit Model 2.

- IBM 3330 Disk Storage.
  A 3330 Disk Storage modules consists of two independent disk drives. One to four 3330 modules may be connected to a system via the 3830 Control Unit Model 1; one to three 3330 modules may be attached to a 3333 (see above).

Thus, a system may have one or two 3333 modules attached, each module followed by one to three 3330 modules. Or it may have one to four 3330 modules attached.

Access Mechanism

Each disk drive contains an access mechanism of nineteen read/write heads, mounted on a vertical assembly. The heads are aligned vertically and are all moved to any of 411 positions.

Storage and Record Capacity

Data is stored on IBM 3336 Disk Packs. These packs are mounted in powered drawers, and are easily removable and interchangeable. The 100 million byte capacity of one volume is based on 404 cylinders per disk surface. The remaining seven tracks per surface are used as alternate tracks to replace defective tracks.

IBM 3340 Disk Storage

The IBM 3340 Disk Storage is a direct access storage facility which combines large storage capacity with high performance and maximum data integrity. This high data integrity is achieved by the concept of data module and drive (see *Access Mechanism* below).

The 3340 Disk Storage can be configured from combinations of three modules:

- Model A2 with two drives
- Model B1 with one drive
- Model B2 with two drives

Each system must have at least one model A2. The model A2 attaches directly to the System/370 Models 115 and 125, and to Models 135, 145, 155-II, and 158 via the appropriate IFA (Integrated File Adapter), ISC (Integrated Storage Control), or 3830-2 control unit. Models B1 and B2 can be attached to a model A2 or to another B model.

Up to four drives can be attached to a System/370 Model 115, and up to eight drives to a Model 125. Up to eight drives per string can be attached to the other System/370 models.

Access Mechanism
The 3340 is designed according to the concept of data module and drive, that is, the data module contains - sealed in a cartridge - the recording disks, and the access arms and read/write heads which remain with the data module when it is removed from the drive.

Storage and Record Capacity
Data is stored on IBM 3348 Data Modules. Data Modules are available in two models:

- Model 35, with a capacity of approximately 35 million bytes.
- Model 70, with a capacity of approximately 70 million bytes.

Thus, the maximum capacity on an IBM 3340 Disk Storage Model A2 is 140 million bytes.

IBM 2321 Data Cell Drive

The IBM 2321 Data Cell Drive is a device for storing data on magnetically coated strips. Two hundred strips are contained in a single removable and interchangeable cell assembly. Ten cell assemblies, each of them containing twenty subcells, can be mounted on a data cell drive at a time.
A rotary positioning system positions a selected subcell of ten strips beneath an access station. At this station, one selected strip is withdrawn from the subcell and rotated past a read/write head element for data transfer. The strip is then returned to its original location in the subcell.

Access Mechanism
The access mechanism of the 2311 consists of a read/write head block which contains 20 magnetic elements. It can be positioned to one of five positions, creating five cylinders of 20 data tracks each, and providing for 100 recording tracks per strip.

Other Devices Supported by DOS/VS

In addition to the devices described above, DOS/VS supports the following devices:

- Terminal devices:
  - IBM 1030 Data Collection System
  - IBM 1050 Data Communication System
  - IBM 1060 Data Communication System
  - IBM 2721 Portable Audio Terminal
  - IBM 2740 Communication Terminal Models 1 and 2
  - IBM 2760 Optical Image Unit
  - IBM 2770 Data Communication System
  - IBM 2780 Data Transmission Terminal Models 1-4
  - IBM 2790 Data Communication System
  - IBM 2972 Banking Terminal
  - IBM 3735 Programmable Buffered Terminal
  - IBM 3740 Data Entry System
  - IBM 3780 Data Communication Terminal
- Display devices:

  - IBM 2260 Display Station Models 1 and 2
  - IBM 2265 Display Station
  - IBM 3270 Information Display System

- Manual Controls:
  - IBM 3210 Console Printer-Keyboard
  - IBM 3215 Console Printer-Keyboard

- Miscellaneous equipment:
  - IBM 1017 Paper Tape Reader Models 1 and 2
  - IBM 1018 Paper Tape Punch Model 1
  - IBM 1255 Magnetic Character Reader Models 1-3
  - IBM 1259 Magnetic Character Reader Model 2
  - IBM 1270 Optical Character Reader Models 1-4
  - IBM 1275 Optical Character Reader Models 2 and 4
  - IBM 1287 Optical Reader Models 1-5
  - IBM 1288 Optical Page Reader Model 1
  - IBM 1419 Magnetic Character Reader Model 1
  - IBM 2671 Paper Tape Reader Model 1
  - IBM 2816 Tape Switching Unit Model 1
  - IBM 7770 Audio Response Unit Model 3

Appendix 2: Attributes of Direct Access Storage Devices

Figures 5.3, 5.4, 5.5, and 5.6 give useful information about the physical attributes of the direct access devices supported by DOS/VS.

STORAGE DEVICE▶	2311	2314* 2319*	3330* 3333*	3340**
Volumes per device	1	8	8	2
Cylinders per volume	200	200	404	696
Cylinders per device	200	1,600	3,232	1,392
Tracks per cylinder	10	20	19	12
Tracks per volume	2,000	4,000	7,676	8,352
Tracks per device	2,000	32,000	61,408	16,704
Bytes per track	3,625	7,294	13,030	8,368
Bytes per cylinder	36,250	145,880	247,570	100,416
Bytes per volume	7,250,000	29,176,000	100,018,280	69,889,536
Bytes per device	7,250,000	233,408,000	800,146,240	139,779,072

Figure 5.3. Disk storage capacity table

* *The table shows the maximum data capacity of an installation with eight disk drives.*

** *The table shows the data capacity of a model A2 with two Model 70 data modules.*

Bytes per	track	2,000
	cylinder	40,000
	strip	200,000
	subcell	2,000,000
	cell	40,000,000
	full array	400,000,000
Tracks per	cylinder	20
	strip	100
	subcell	1,000
	cell	20,000
	full array	200,000
Cylinders per	strip	5
	subcell	50
	cell	1,000
	full array	10,000
Strips per	subcell	10
	cell	200
	full array	2,000
Subcells per	cell	20
	full array	200
Cells per	full array	10

Figure 5.4. Data cell capacity table

Maximum Bytes per Record Formatted without keys					Records per Track	Maximum Bytes per Record Formatted with keys				
2311	2314 2319	2321	3330	3340		2311	2314 2319	2321	3330	3340
3625	7294	2000	13030	8368	1	3605	7249	1984	12974	8293
1740	3520	935	6447	4100	2	1720	3476	920	6391	4025
1131	2298	592	4253	2678	3	1111	2254	576	4197	2608
830	1693	422	3156	1966	4	811	1649	406	3100	1801
651	1332	320	2498	1540	5	632	1288	305	2442	1465
532	1092	253	2059	1255	6	512	1049	238	2003	1180
447	921	205	1745	1052	7	428	877	190	1689	977
384	793	169	1510	899	8	364	750	154	1454	824
334	694	142	1327	781	9	315	650	126	1271	706
295	615	119	1181	686	10	275	571	103	1125	611
263	550	101	1061	608	11	244	506	85	1005	533
236	496	86	962	544	12	217	452	70	906	469
213	450	73	877	489	13	194	407	58	821	414
193	411	62	805	442	14	174	368	47	749	367
177	377	53	742	402	15	158	333	38	686	327
162	347	44	687	366	16	143	304	29	631	291
149	321	37	639	335	17	130	277	21	583	260
138	298	30	596	307	18	119	254	15	540	232
127	276	24	557	282	19	108	233	9	501	207
118	258	20	523	259	20	99	215		467	184
109	241	15	491	239	21	90	198		435	164
102	226	10	463	220	22	82	183		407	145
95	211	6	437	204	23	76	168		381	129
88	199		413	188	24	69	156		357	113
82	187		391	174	25	63	144		335	99
77	176		371	161	26	58	133		315	86
72	166		352	149	27	53	123		296	74
67	157		335	137	28	48	114		279	62
63	148		318	127	29	44	105		262	52
59	139		303	117	30	40	96		247	42

Figure 5.5. Record capacities on DASD

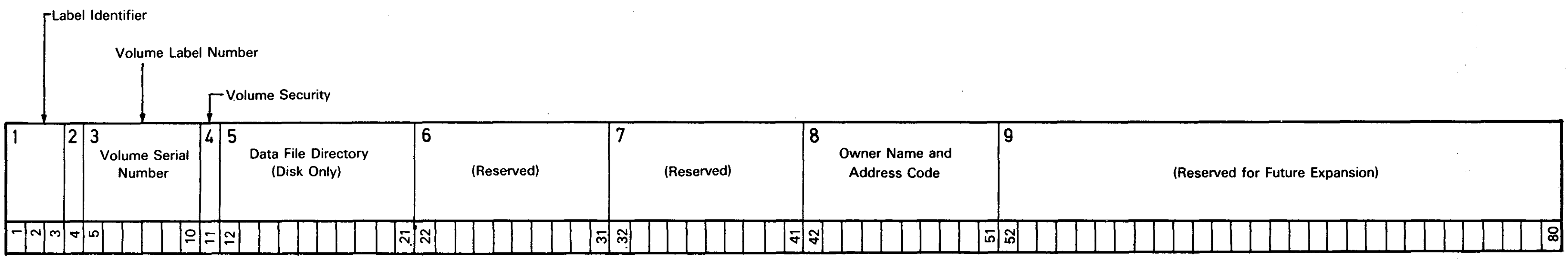
STORAGE DEVICE	TRACK CAPACITY IN BYTES, WHEN R0 IS USED AS SPECIFIED BY IBM PROGRAMMING SYSTEMS	BYTES REQUIRED FOR DATA RECORDS			
		DATA RECORDS (except for last record)		LAST RECORD	
		Without key	With key	Without key	With key
2311	3625	$61 + \frac{537.DL}{512}$	$81 + \frac{537.(KL+DL)}{512}$	DL	20+KL+DL
2314, 2319	7294	$101 + \frac{2137 . DL}{2048}$	$146 + \frac{2137 (KL+DL)}{2048}$	DL	45+KL+DL
2321	2000	$84 + \frac{537 . DL}{512}$	$100 + \frac{537 . (KL+DL)}{512}$	DL	16+KL+DL
3330, 3333	13030	135+DL	191+KL+DL	DL	56+KL+DL
3340	8368	167+DL	242+KL+DL	167+DL	242+KL+DL

KL = key length
DL = data length

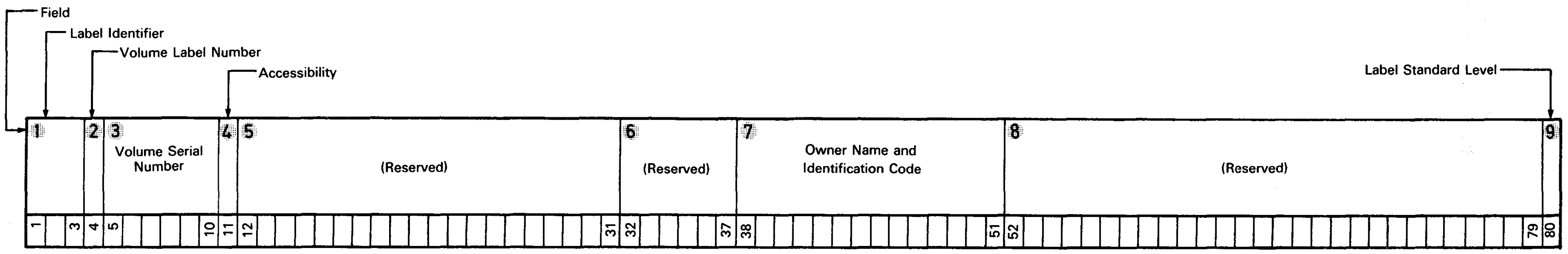
Figure 5.6. Track capacities on DASD

Appendix 3: Standard Label Formats for Magnetic Tape, Diskette, and DASD

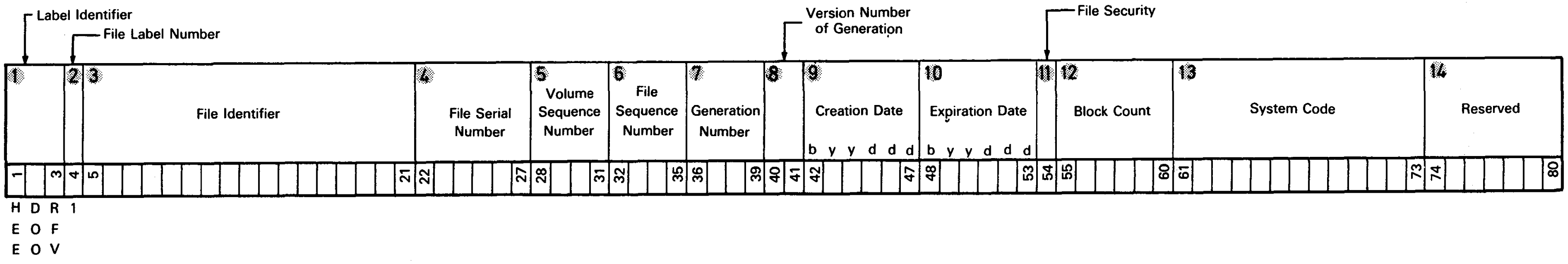
IBM Standard Volume Label, Tape or DASD



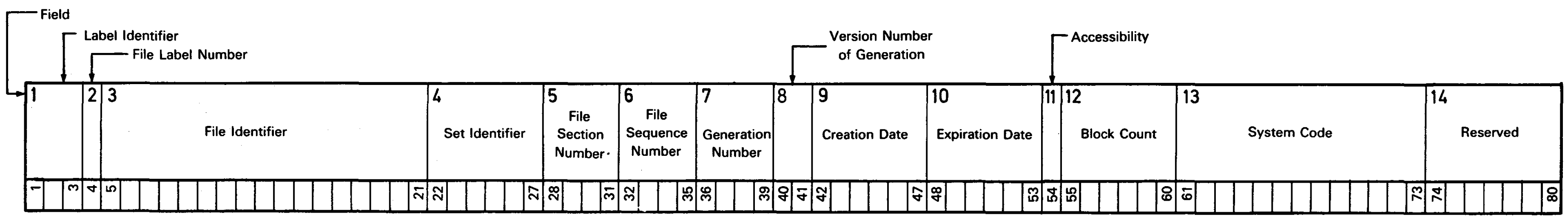
ANSI Standard Volume Label, ASCII Tapes



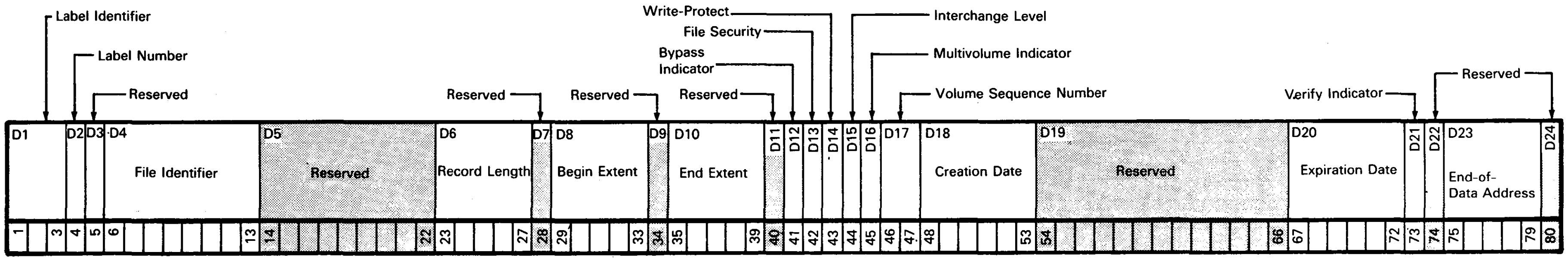
IBM Standard Tape File Label



ANSI Standard Tape Label, ASCII Tapes

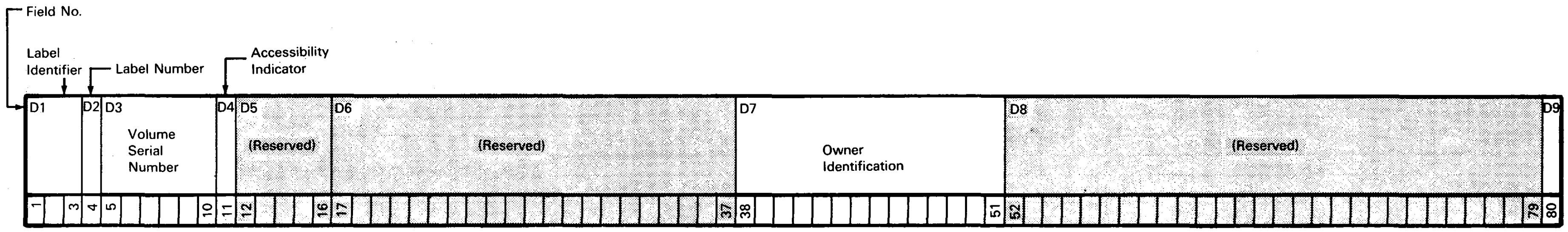


Standard Diskette Volume Label (VOL1)

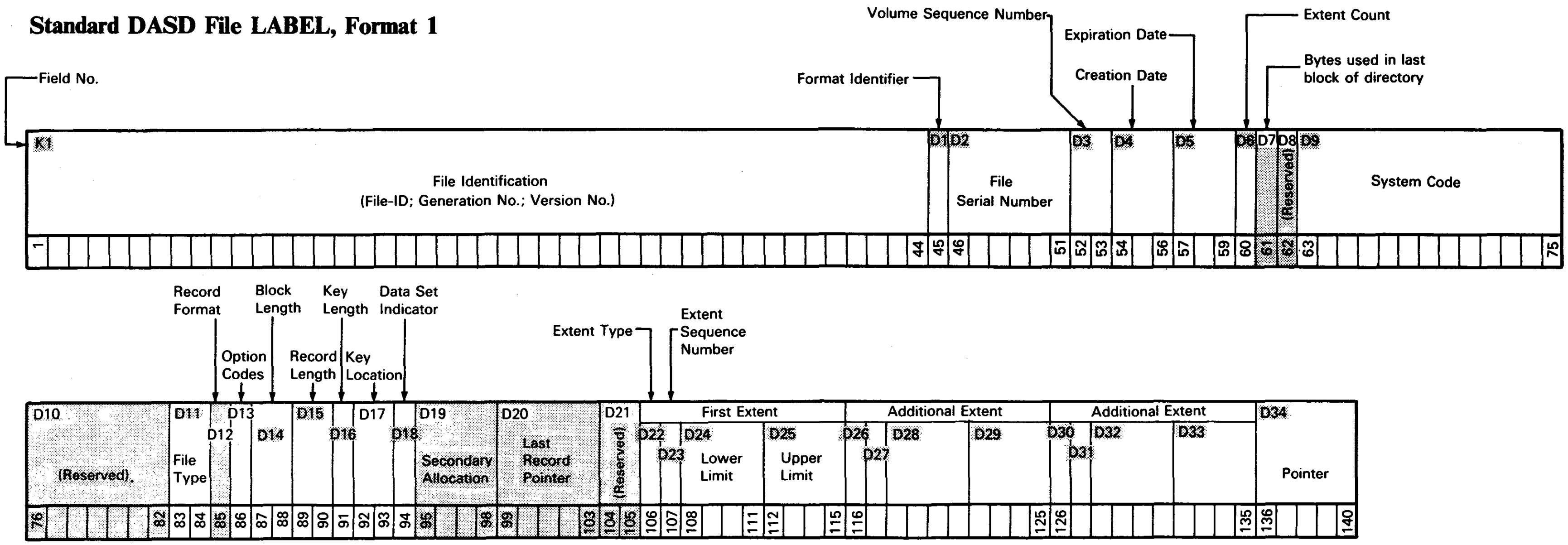


Shaded areas apply to OS only, or are reserved for future use.

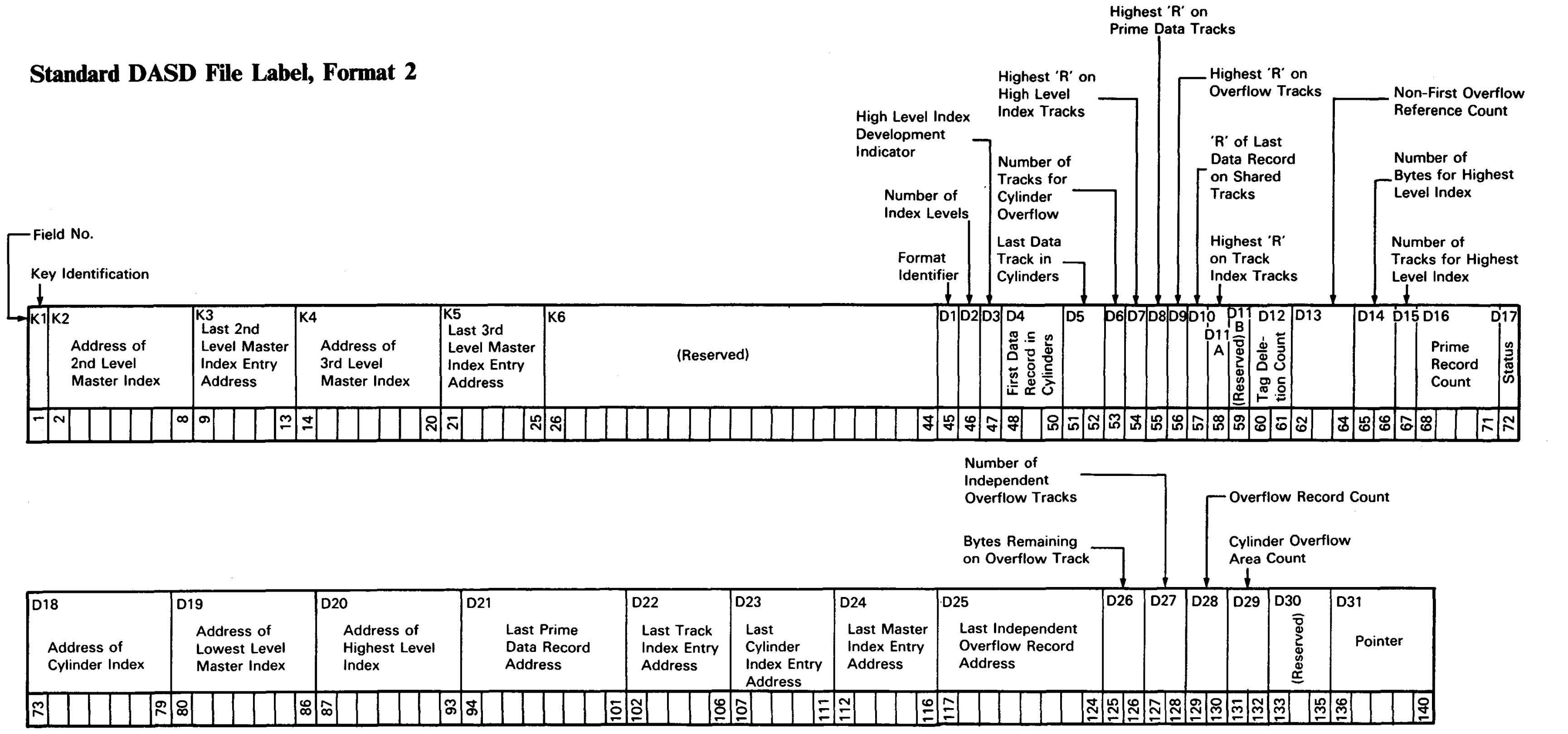
Standard Diskette HDR1 Label



Standard DASD File LABEL, Format 1

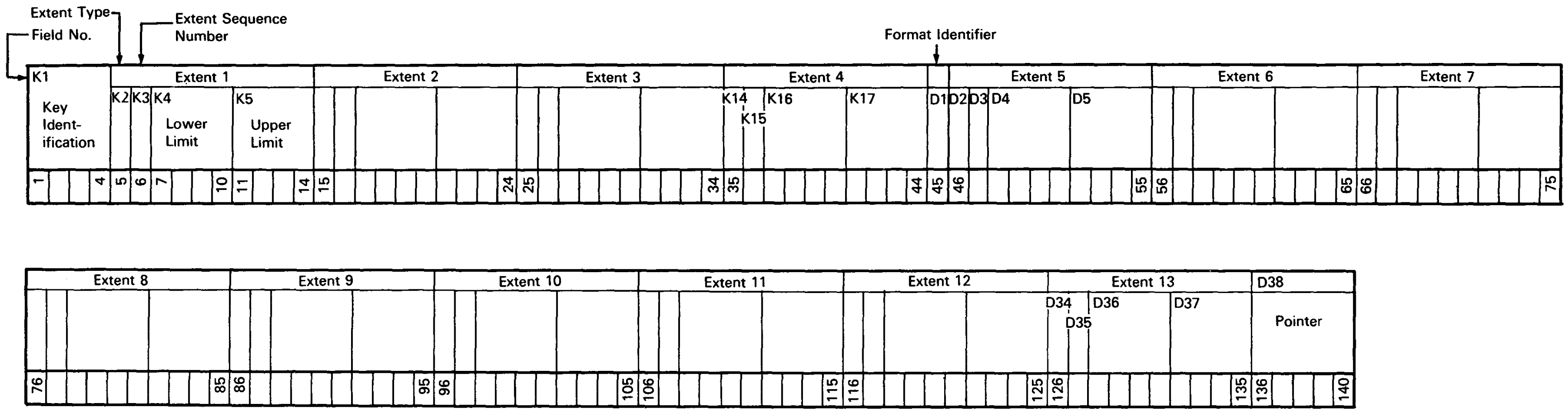


Standard DASD File Label, Format 2

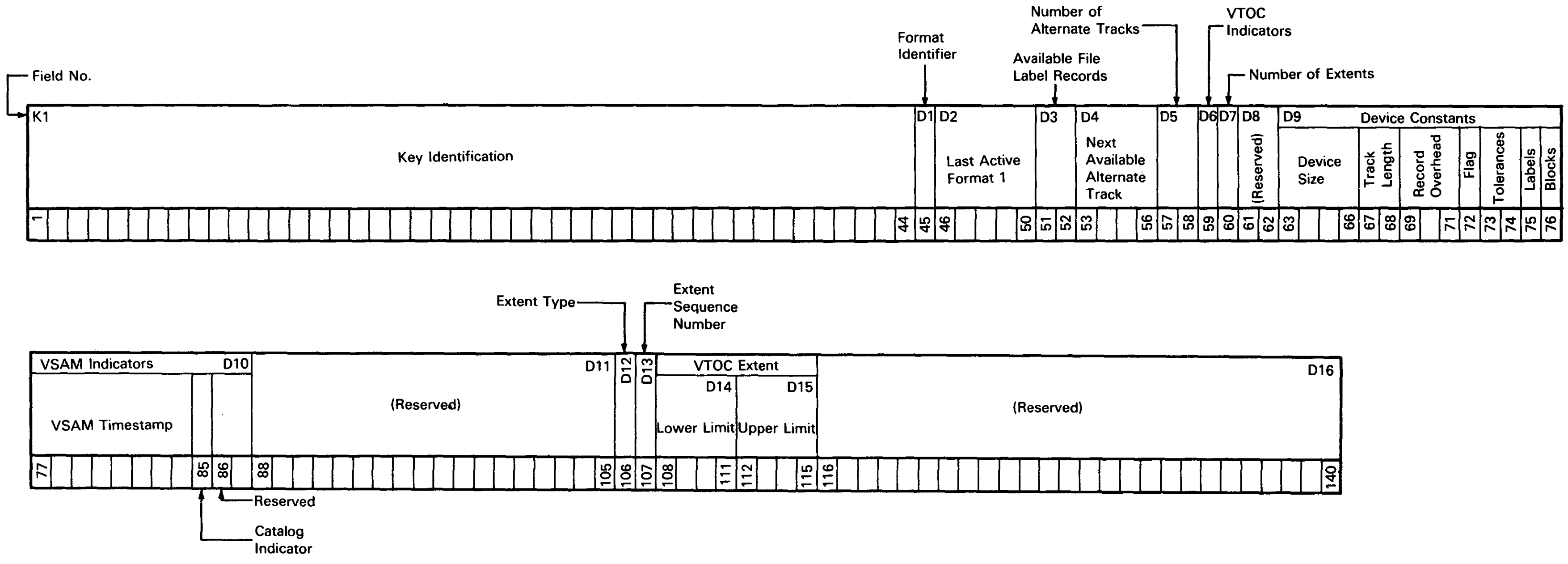


Field No.
Key Identification
Format Identifier
Number of Index Levels
High Level Index Development Indicator
Last Data Track in Cylinders
Number of Tracks for Cylinder Overflow
Highest 'R' on High Level Index Tracks
Highest 'R' on Prime Data Tracks
Highest 'R' on Overflow Tracks
'R' of Last Data Record on Shared Tracks
Highest 'R' on Track Index Tracks
Non-First Overflow Reference Count
Number of Bytes for Highest Level Index
Number of Tracks for Highest Level Index
K1
K2 Address of 2nd Level Master Index
K3 Last 2nd Level Master Index Entry Address
K4 Address of 3rd Level Master Index
K5 Last 3rd Level Master Index Entry Address
K6 (Reserved)
D1
D2
D3
D4 First Data Record in Cylinders
D5
D6
D7
D8
D9
D10
D11 A
D11 B (Reserved)
D12 Tag Deletion Count
D13
D14
D15
D16 Prime Record Count
D17 Status
1 2 8 9 13 14 20 21 25 26 44 45 46 47 48 50 51 52 53 54 55 56 57 58 59 60 61 62 64 65 66 67 68 71 72
Number of Independent Overflow Tracks
Bytes Remaining on Overflow Track
Overflow Record Count
Cylinder Overflow Area Count
D18 Address of Cylinder Index
D19 Address of Lowest Level Master Index
D20 Address of Highest Level Index
D21 Last Prime Data Record Address
D22 Last Track Index Entry Address
D23 Last Cylinder Index Entry Address
D24 Last Master Index Entry Address
D25 Last Independent Overflow Record Address
D26
D27
D28
D29
D30 (Reserved)
D31 Pointer
73 79 80 86 87 93 94 101 102 106 107 111 112 116 117 124 125 126 127 128 129 130 131 132 133 135 136 140

Standard DASD File Label, Format 3



Standard DASD File Label, Format 4



Appendix 4: Programming Considerations for Some Special Types of I/O Equipment

This appendix provides additional information for the control, through the Sequential Access Method, of the following device types:

- Punched card equipment
- Paper tape equipment
- Magnetic and Optical Character Readers
- Optical Mark Readers

Punched Card Considerations

The range of punched card equipment provided by IBM allows a user to select devices that best support his applications (see Appendix 1). Some of these devices are able to perform only one function, for example, reading or punching. Other types are able to perform different functions in separate card paths, and there are devices that can perform different functions in one card path. This part of Appendix 4 provides hints to bear in mind when using a device that can perform multiple functions.

Associated Files

Associated files are possible on IBM 2560, 3525, and 5425 card devices. One speaks of associated files when more than one function (read, punch, print) must be performed on one card file. For example, Figure 5.7 shows what can be done on a single pass of cards through a card path.

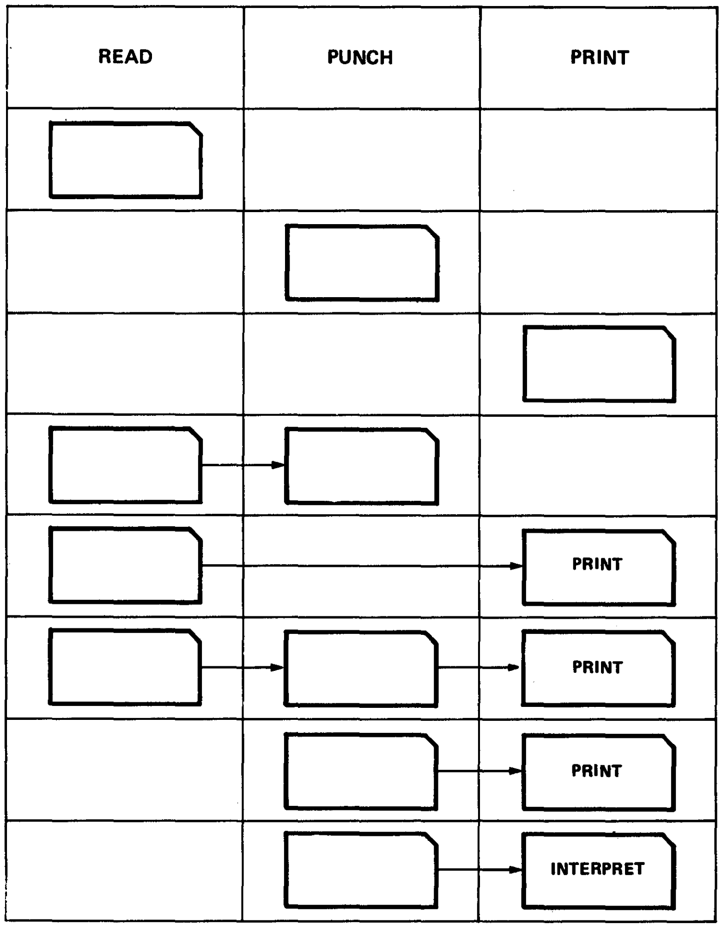


Figure 5.7. What can be done on a single pass

In DOS/VS assembler language or RPG a file definition must be given for each function which is to be performed on the cards (an exception is punch-interpret which requires only one file definition). Each of the associated files is then specified as associated by means of the operands FUNC and ASOCFLE in the DTFCD and *DTFPR* declarative macro instructions. (DTFPR is to be used for the print function of a card device.)

The FUNC operand is used to specify the function combinations; the ASOCFLE operand is used to specify the name and address of the associated file.

```
DTFCD          Read—Punch
DTFCD }      { Read—Punch—Print
DTFPR }      { Read—Print
               Punch—Print
```

```
FUNC= { I | R | P | RP | RW | RPW | PW }
```

This operand specifies the type of file to be processed. Where applicable, the valid entries are:

I (Interpret), R (Read), P (punch), RP (Read-Punch), RW (Read-Print), RPW (Read-Punch-Print), and PW (Punch-Print).

When FUNC=I is specified, the cards will be both punched and interpreted; no associated file is necessary to achieve this.

RP, RW, RPW, and PW are used, together with the ASOCFLE operand, to specify associated files. When one of these parameters is specified for one associated file it must also be specified for the other associated files.

```
ASOCFLE=filename
```

This operand specifies the filename of associated read, punch, and print files, and enables macro sequence checking by the logic module of each associated file. It is used together with the FUNC operand to define associated files. One filename is required per DTF for associated files.

FUNC=	In ASOCFLE operand of . . .		
	read DTFCD, specify filename of . . .	punch DTFCD, specify filename of . . .	print DTFPR, specify filename of
RP	punch DTFCD	read DTFCD	
RW	print DTFPR		read DTFCD
PW		print DTFPR	punch DTFCD
RPW	punch DTFCD	print DTFPR	read DTFCD

For example, if FUNC=PW is specified, specify the filename of the print DTFPR in the ASOCFLE operand of the punch DTFCD, and specify the filename of the punch DTFCD in the ASOCFLE operand of the print DTFPR.

I/O Areas / Work Areas

Associated files can have only one I/O area. Each associated file may have a different I/O area with or without a work area; the I/O area or work area may also be the same for associated files. For example, by specifying the same I/O area (or work area) for an RW file, cards will be read and interpreted. Or, if the same I/O area (work area) is used for the associated print and punch files of an RPW card file, the information which is punched will also be printed (interpreted).

Processing Considerations for Associated Files on a 2560 or 5425
DOS/VS does not provide special macro instructions to control the overlapping of reading with processing. For a 2560 or a 5425, however, the DOS/VS Assembler language programmer who uses LIOCS can achieve a type of overlapped processing in the way as is described in the following text. The DOS/VS Assembler language programmer who uses PIOCS may design his own overlapped processing.

Read-Punch Associated Files.
For read-punch (RP) associated files, the GET for the read file as well as the PUT for the punch file must always both be issued for each card. If no punching is desired, the output area or work area must be filled with blanks; LIOCS tests for blanks in the output area or work area and if it finds them no punching is performed. If in the DTFCD macro instruction the operand CTLCHR=YES or ASA is specified for the punch file, the appropriate 370 or ASA control character must always be present in the first byte; only the data portion following the control character must be filled with blanks, if so desired. If the CNTRL macro instruction is used, it must be issued before the PUT. As a result of the PUT, LIOCS will initiate the reading of the next card, and read it into a special buffer which is part of the DTF table for the read file. The user needs not and cannot set up this buffer or control its use. The next GET will obtain the data from this buffer into the input area. Thus, by issuing the PUT as soon as possible after a GET, as much as possible of the next card will be read while the program is doing other processing.

Read-Print Associated Files.
For read-print (RW) associated files, the GET for the read file must always be issued for each card. The PUT for the print file needs to be issued only if actual printing is desired. However, the PUT will initiate the reading of the next card as is explained for RP files above; it is therefore recommendable to always issue a PUT even if no printing is desired: in such a case the output area or work area should be filled with blanks. If no PUT is issued, no overlapped processing will be achieved.

Read-Punch-Print Associated Files.
For read-punch-print (RPW) associated files, a GET for the read file must always be issued for each card. Also, the PUT for the punch file must always be issued for each card. The PUT for the print file needs to be issued only if actual printing is desired. However, it is the PUT for the print file that initiates the reading of the next card, as described for RP files, above; it is therefore recommendable to issue a PUT for the print file at all times. If the PUT for the print file is omitted, no overlapped processing can be achieved. If an output area or work area is filled with blanks, no punching and/or printing will occur, as is described above for RP and RW files. If in the DTFCD macro instruction the operand CTLCHR=YES or ASA is specified for the punch file, the appropriate control character must always be present in the output area or work area

(first byte); only the data portion of the output area or work area may be filled with blanks (see above). If the CNTRL macro instruction is used it must be issued before the PUT for the punch file.

Paper Tape Considerations

Paper tape I/O routines can, in DOS/VS, be programmed in assembler language only. Input and output is performed by normal GET and PUT macro instructions.

Paper tape readers accept the following two record formats:

- Fixed-length, unblocked format (Format F)
- Undefined format (Format U).

The characteristics of a paper tape file are specified in the DTFPT macro instruction. The record format specified does not specifically apply to the format of the data as it appears physically on paper tape, but to the logical format of the records as they appear in the I/O area. The physical data may have characters embedded that must be deleted from the records, such as delete characters and shift codes.

The DTFPT macro instruction specifies the characteristics of the file. Some of these specifications are described in this section:

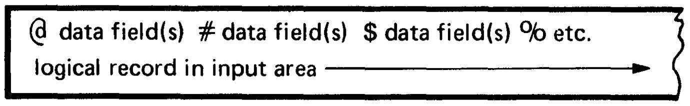
BLKSIZE	Specifies the size of a record. For fixed-length records this is the size of a logical record; for undefined records this is the maximum size of a physical record.
OVBLKSZ	Specifies the amount of character to be read from paper tape. It is used for fixed-length records. OVBLKSZ may be equal to or higher than BLKSIZE.
SCAN LSCAN FSCAN TRANS LTRANS FTRANS	These operands of the DTFPT macro instruction refer to translation tables. Physical data may contain any paper tape code, whereas logical data is expected to be in EBCDIC. Also the data may be compressed by deleting shift codes and delete characters from the physical data. This can be done by an automatic translation process, under control of IOCS.
EORCHAR	Undefined records must end with an end-of-record character. For input, this character is specified in a translation table; for output, the end-of-record character is added by IOCS as specified in this operand.
DELCHAR	A user may specify that certain characters must be deleted from the physical data. For input, these characters are specified in a translation table; for output, the delete character can be used as specified in this operand, but only for the IBM 1018 with the Error Correction Feature.

Paper Tape Input

Data that is read from paper tape may physically be in any paper tape code a user requires. Logical data in virtual storage is expected to be in internal IBM System/370 code (EBCDIC). If some code must be translated this can be done automatically, as well as translation of shifted code (figure shift and letter shift). Code translation is discussed separately in this section.

After a GET has been issued a logical record is obtained from physical data. During this process, delete characters and any shift characters are removed from the data. Data that follows such characters is shifted to the left so that the user need not be concerned about such code.

More serious is the problem of synchronizing the data fields with the program. The paper tape is read character by character, and all characters are placed in subsequent character locations in the input area. If, in some data field, one character too many or too few is specified, all following fields will be out of phase. Therefore one usually adds extra characters with a special bit configuration to the data. These characters are expected to occur in each record in the same character location. By checking this location a user can identify incorrectly formatted records:


@ data field(s) # data field(s) $ data field(s) % etc.
logical record in input area

Another method of checking whether data that is processed is valid, is to expand data fields with an additional check character which is the result of some calculation.

For example, for numeric data fields one may add all characters on even locations, do the same for all characters on uneven locations, multiply the two sums, and then use the last character of the product as a check character. A data field with the content 85318 would then be represented on paper tape as 85318*4*: 8+3+8=19, 5+1=6, 19x6=11*4*.

Undefined Record Format on Input

Each record must be followed by an end-of-record (EOR) character. After a GET, a count-controlled read is performed, count-controlled by the operand BLKSIZE. Reading stops when an EOR character is sensed. The input area must be at least one position longer than the longest record anticipated, including any delete characters and shift codes that may be embedded in the data. If an input area is filled up completely, the record is assumed to be overlength, and the wrong-length record routine of IOCS will become active.

After data has been read up to EOR, the delete characters and shift characters are removed by the translation process. A translated and compressed logical record is presented to the user.

Consecutive EOR characters are skipped. The system will never return to the user with a data length zero. The length of the logical records is communicated to the user in a register.

Fixed-Length, Unblocked Format on Input

Records must not be followed by an EOR character, since all characters enter virtual storage as normal characters. An EOR character does not stop the reading of fixed-length records.

The term 'fixed-length' applies to the format of a logical record in the input area, after it has been translated (if necessary) and compressed. It does not apply to the format of the data as it appears physically on the paper tape. A paper tape file consists of one continuous string of data characters, and it is IOCS that establishes boundaries between the records by means of the operand BLKSIZE as specified by the user. The physical data may have embedded delete characters and shift codes. It may be necessary therefore to read more characters than the size of a logical record. The number of characters that must be read is specified by the OVBLKSZ operand in the DTFPT macro instruction.

After a GET, IOCS starts a count-controlled read until the input area contains the number of characters specified in OVBLKSZ. Then the translation process starts, eliminating shift codes and delete characters from the data. If the resulting record is shorter than BLKSIZE, additional reads are performed until IOCS has obtained a logical record with a size equal to BLKSIZE. As a result, some characters may belong to the next logical record. These characters are moved to the beginning of the input area when the next GET is issued. It is therefore important that the programmer does not clear the input area beyond the size of the logical record as defined in BLKSIZE. If he does, he will destroy part of the following logical record.

Undefined Format vs Fixed-Length Format

When using undefined format, a GET reads data until an EOR character is read. The programmer must make sure that his input area can fully accommodate that data, taking additional characters in the physical data into account.

When using fixed-length format, IOCS performs as many read operations as necessary to obtain one logical record. If the I/O area is large enough to accommodate the logical data, but not large enough to accommodate the physical data, this is solved by additional reads when the preceding read and its translation and compression process is completed.

The main difference between the processing of the two formats is that IOCS can recognize record boundaries of undefined records, but cannot recognize those boundaries in fixed-length records. If too few or too many characters are specified for some particular record, the problem is more serious for a fixed-length record than for an undefined record. For undefined records, only the wrong record will be out of phase, whereas for fixed-length records all records following the wrong one will be out of phase. As long as the user ensures that his input area is large enough to contain all physical data of one record, therefore, it is usually better to use and undefined record format, even if the logical records have a fixed-length format.

Paper Tape Output

Data may be written in any code a user requires. Translation of shifted or non-shifted code can be done automatically. Code translation is discussed separately in this section.

After a PUT has been issued, the logical record is expanded by IOCS during the translation process, if required. Shift characters are added to the data when necessary.

Undefined Record Format on Output

Since IOCS adds an EOR character to each record, it need not be written by the user. The output area and the BLKSIZE operand must reflect at least the longest record anticipated.

Fixed-Length Record Format on Output
The number of characters contained in each logical record is specified in the BLKSIZE operand. Logical records are translated by IOCS if necessary. As a result of a PUT, a count-controlled write causes the specified number of characters to be written. For shifted code files, the records are expanded by IOCS, which adds the figure shift and letter shift characters to the data. IOCS performs additional writes to construct the required physical block.

Code Translation

Translation of Non-Shifted Code on Input
The TRANS operand in the DTFPT macro instruction is used for translation of non-shifted code directly into internal IBM System/370 code. If the input is in EBCDIC, no translation is required, and the TRANS operand may be omitted.

The SCAN operand may be used alone or in conjunction with TRANS to delete characters from records that do not contain shifted code. There must not be any 04 or 08 entries in the scan table referred to by SCAN.

Translation of Shifted Code on Input
If the input contains shifted code, the FTRANS, LTRANS, and SCAN operands must be specified in the DTFPT macro instruction.

Translation of shifted code is accomplished by IOCS as follows:

1. The data is first scanned for shift characters. The segments between shift characters are translated, using the appropriate shift table.
2. The translated segments are moved to the left to remove the shift characters.
3. Steps 1 and 2 are repeated for each segment until the complete record has been translated and compressed.

These steps result in a translated and compressed record, left-justified in the input area. The record length is communicated to the user in a register, which is designated in the RECSIZE operand.

The EOR character at the end of undefined records must be shift-independent. That is, it must be effective whether the coding is in letter shift or figure shift. If there is valid code in either shift that corresponds to the coding of the EOR character established for a particular job, then this shift code must not be included in the input.

IOCS assumes that the first record read from paper tape starts with figure shift coding. Therefore, if the first record starts with letter shift code, the user must make sure that the first character in the first physical block is a letter shift character. The shift status is carried from one record to the next and remains unchanged until another shift character is encountered.

Translation of Non-Shifted Code on Output
The TRANS operand in the DTFPT macro instruction is used for translation of non-shifted code, from internal IBM System/370 code into any other code required by the user. If the output is to be punched in EBCDIC, no translation is required, and the TRANS operand may be omitted.

Translation of Shifted Code on Output
If the output contains shifted code, the TRANS, FSCAN, and LSCAN operands must be included in the DTFPT macro instruction. The operand OVBLKSZ may be used or omitted. If omitted, the records are written segment by segment, and IOCS adds a shift character in front of each segment. If OVBLKSZ is used, however, the segments are moved to the right, while shift characters are inserted in the data before each segment. If OVBLKSZ is specified too low, additional writes are performed to produce the physical data.

Additional information about input/output control is provided in *DOS/VS Supervisor & I/O Macros*, GC33-5373.

MICR/OCR Considerations

Note: This section excludes the IBM 3886 Optical Character Reader. For more information on the 3886, see 3886 Optical Character Reader Considerations.

Magnetic Ink Character Recognition (MICR) devices, and Optical Character Reader/Sorter (OCR) devices can be operated in any partition. The user is supplied with an extension to the supervisor, which monitors, by means of external interrupts, the reading of documents into a user supplied I/O area (document buffer area).

The user must access all MICR/OCR documents through logical IOCS macro instructions. Upon request, LIOCS gives a next sequential document and automatically engages and disengages the devices to provide and continuous stream of input. Detected error conditions and information about errors are passed to the user in each document buffer.

MICR and OCR devices are unique in that documents must be read at a rate dictated by the device rather than by the program. To allow time for necessary processing (including the determination of pocket selection), the device generates an external interruption at the completion of each read operation for each document. The supervisor gives absolute priority to external interrupt processing.

In a multiprogramming system with MICR/OCR document processing, any partition can utilize MICR/OCR devices. In problem programs, these devices can be controlled by means of Assembler language only, at the LIOCS GET level if one device is attached, or at the LIOCS READ/CHECK/WAITF level if multiple MICR/OCR devices are attached. In the latter case, the user is allowed to continue processing as long as one file has documents ready for processing.

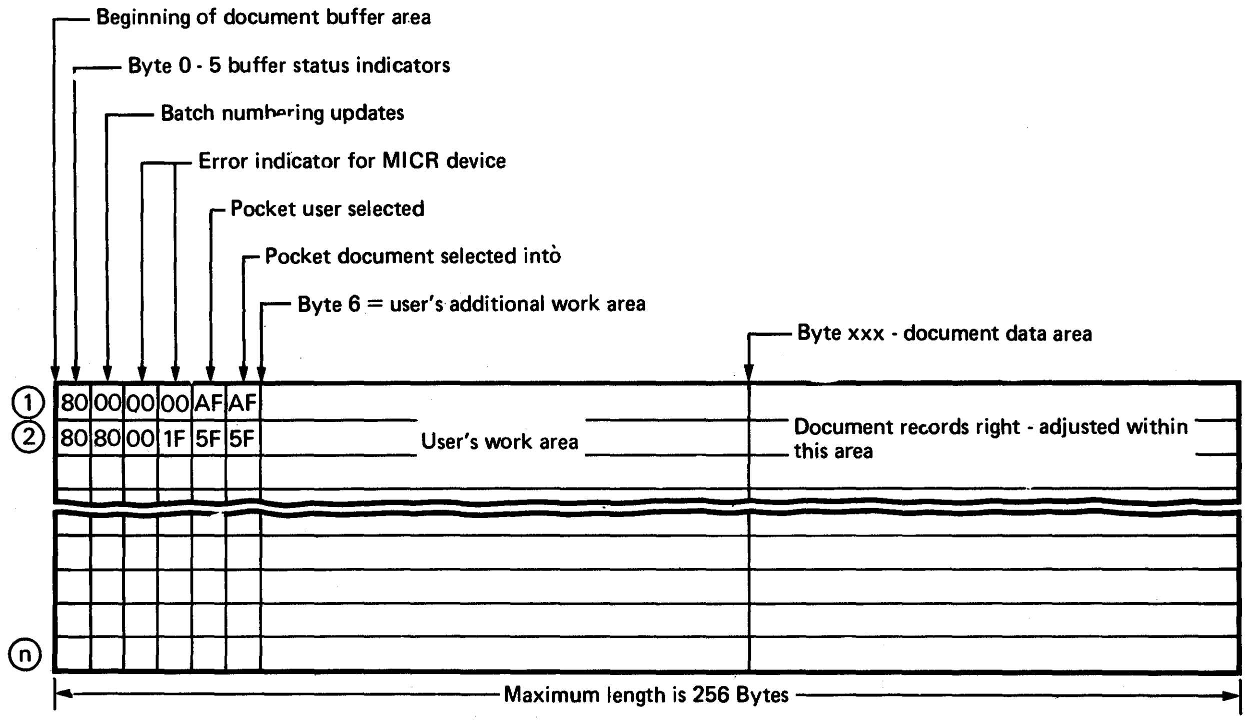
This introduction to MICR/OCR processing already indicates that the two types of devices are very much the same, as far as the processing characteristics are concerned.

Before a user can begin any type of processing on MICR/OCR devices, he must be aware of the special buffer format. Each document buffer must not exceed 256 bytes, including the six-byte buffer status indicates, any additional user work area, and the maximum document data area.

A user may specify any number of document buffers between 12 and 254; the maximum number depends on the amount of virtual storage available.

Figure 5.8 shows a storage map of a set of document buffers. Technical details about buffer control can be found in the publication *DOS/VS Supervisor & I/O Macros*, GC33-5373.

The first time a GET (or READ) is executed, the supervisor engages the device for continuous reading. Each time thereafter, the GET (or READ) merely points (through IOREG) to the next sequential buffer within each document buffer area. When a buffer for a file becomes available, the user's main line processing continues with the instruction after the GET (or READ-CHECK combination).


Beginning of document buffer area
Byte 0 - 5 buffer status indicators
Batch numbering updates
Error indicator for MICR device
Pocket user selected
Pocket document selected into
Byte 6 = user's additional work area
Byte xxx - document data area
①
②
ⓝ
80 00 00 00 AF AF
80 80 00 1F 5F 5F
User's work area
Document records right - adjusted within this area
Maximum length is 256 Bytes

① Indicates the normal condition (no errors) when the document is being processed and the stacker selection is complete to pocket A (1412).

② Indicates the normal condition (no errors - all fields read) when the document is being processed and the stacker selection is complete to pocket 5 and batch numbering update was performed (1419 model 1 or 31).

ⓝ Number of buffers (between 12 and 254) is limited by the amount of virtual storage available.

Figure 5.8. MICR document buffer area

Each time an end-of-document condition occurs on an MICR device, the user's main line processing routine, or any other routine having control at that time, is interrupted by the supervisor's external interrupt routine. The external interrupt routine branches immediately to the user's stacker selection routine. After the user selects a pocket, he exits from his stacker selection routine so that the supervisor can issue the stacker selection command. At this time, the MICR device(s) should be reading document data into its (their) respective document buffer area(s). The supervisor, in priority order, passes control to the user's main line processing routine, or the routine that was interrupted, at the point of its interruption.

Thus, MICR document processing continues concurrently within the:

1. User's main line processing routine,
2. Supervisor's external interrupt routine, and
3. User's stacker selection routine.

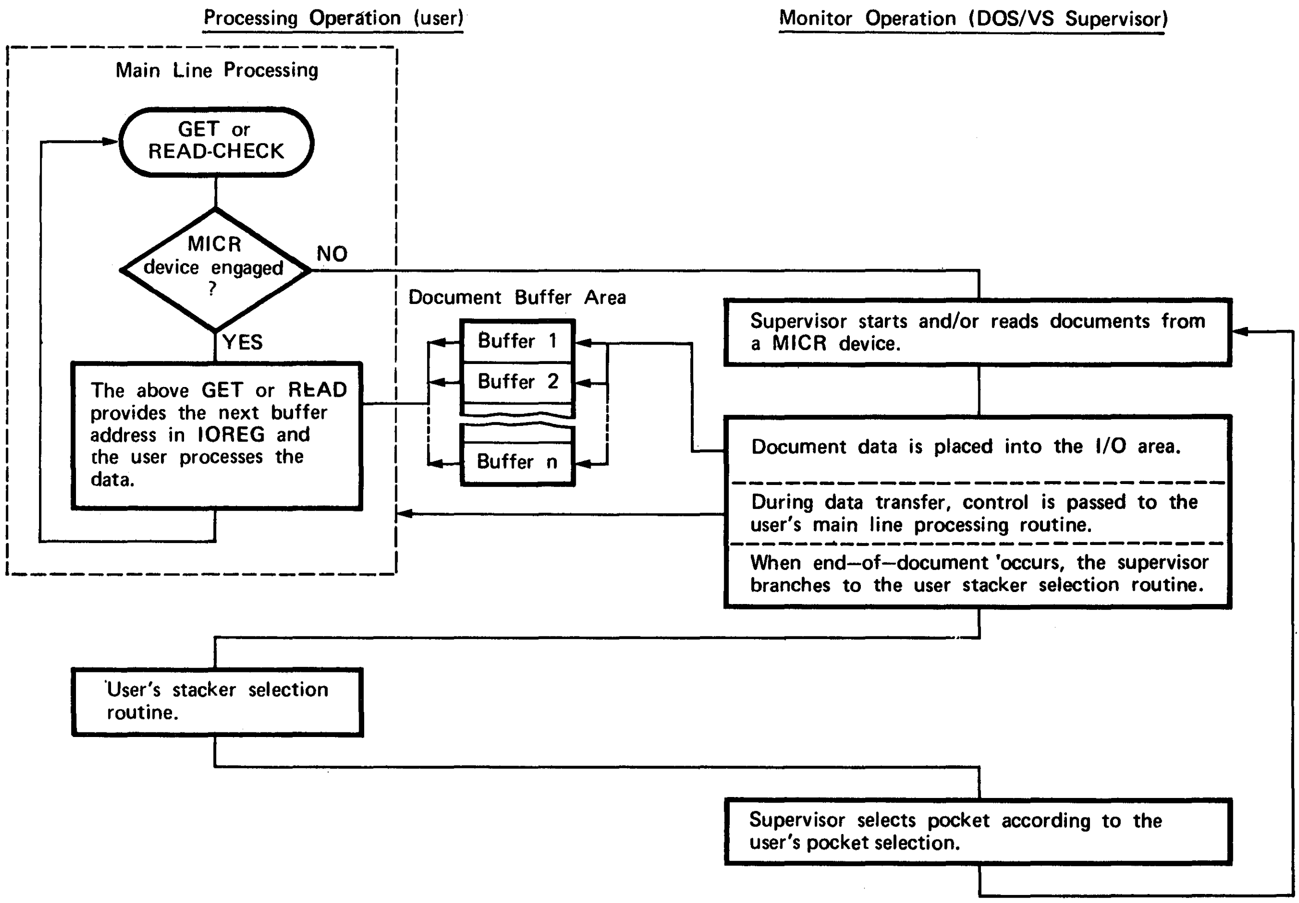

Processing Operation (user)
Monitor Operation (DOS/VS Supervisor)
Main Line Processing
GET or READ-CHECK
MICR device engaged ?
NO
YES
The above GET or READ provides the next buffer address in IOREG and the user processes the data.
Document Buffer Area
Buffer 1
Buffer 2
Buffer n
Supervisor starts and/or reads documents from a MICR device.
Document data is placed into the I/O area.
During data transfer, control is passed to the user's main line processing routine.
When end–of–document occurs, the supervisor branches to the user stacker selection routine.
User's stacker selection routine.
Supervisor selects pocket according to the user's pocket selection.

Figure 5.9. MICR/OCR document processing

The order for exiting from these routines is the reverse of the indicated order. processing and monitor operations continue concurrently until the reader is disengaged, either normally or due to an error.

End-of-file processing must be detected and handled by the user's main line processing routine.

The GET macro performs the functions of a READ except that it waits while the document buffer fills. Instead, the READ posts an indicator in the buffer (byte 0, bit 5) for the user to examine with the CHECK macro. If this indicator bit is ON, the buffer is not ready for processing and a branch is made to the second operand address of the CHECK macro. The user's routine at this operand address can then READ and CHECK another file for document availability. If this buffer is ready for processing, control passes to the next instruction. If a special non-data status exists, the user should analyze the conditions in his ERROPT routine and issue a READ to obtain a document unless an unrecoverable I/O error has occurred. If a second operand is not provided within the CHECK macro. control passes to the ERROPT routine address.

At least one WAITF macro must be issued between two successive executions of any one READ to the same file. The multiple WAITF is essential to the operation of the multiprogramming feature of the system. Its function is to test device operation availability or buffer processing availability. If work can be done on any specified file, control remains in the partition. If not, control passes to a lower priority partition until this partition is ready for processing.

3886 Optical Character Reader Considerations

The IBM 3886 Optical Character Reader, Model 1, can be operated in either a foreground or background partition. You must access all operations through LIOCS macro instructions or through the Physical Input/Output Control System (PIOCS). In problem programs, the device is controlled by means of the Assembler language only, at the READ/WAITF LIOCS level.

Two steps are required to use the 3886 as an input device. In one assembly, you must define the documents to be read. Then, in the problem program, you issue the instructions to process the documents.

Defining Documents

Two macro instructions are provided for defining documents. One, the DFR macro instruction, defines attributes common to a group of line types. The other, the DLINT macro instruction, defines specific attributes of an individual line type. As many as 26 DLINT macro instructions can be associated with one DFR macro as long as the number of line types plus the number of fields is less than or equal to 53.

The DFR and associated DLINT macro instructions are used in one assembly to build a format record. Only one DFR with its associated DLINT macro instructions may be specified in each assembly, and the DFR must precede all DLINT macros in the assembly. The format record must be link-edited into the core image library so that it can be loaded into the 3886 when the file is to be processed. A format record is loaded into the 3886 control unit when the file is opened. This format record contains information about the documents being read, each individual line on the document and each field in the line. This information is used to read the line and edit the data before it is passed to the problem program. For more information about the DFR and DLINT macros, see *DOS/VS Supervisor and I/O Macros,* GC33-5373.

Processing Documents

Existing and new DOS LIOCS macro instructions are used to process documents on the 3886. Some parameters on existing macro instructions have been changed or added. The following macros support the 3886:

Macro	**Functions**
DTFDR	Defines the 3886 file.
OPEN(R)	Opens the 3886 file and loads the first format record.
SETDEV	Loads a different format record and exits to the COREXIT or EOF routine if applicable.
DRMOD	Provides the I/O functions.
READ	Reads a record (one line) from the 3886.
WAITF	Waits for the read operation to be completed. Exits to COREXIT or EOF routine if applicable.
CNTRL	Ejects and stacker selects a document, writes the line mark or page mark or performs timing mark check, and exits to the COREXIT or EOF routine, if applicable.
CLOSE(R)	Closes the 3886 file.

Reading Data Records
Each time a READ macro instruction is issued, one line of data is supplied to your program. Each line read is considered to be a data record. A header record is also provided to your program with each data record. The header record is 20 bytes in EBCDIC. Figure 5.10 shows the contents of the header record.

The data record passed to your program is a fixed-length record containing up to 130 bytes of data. You specify the length of the record in the DTFDR macro instruction. The data record is in one of two formats as follows:

1. If standard mode is specified (IMAGE=NO in the DLINT macro instruction), the data record contains the EBCDIC character codes for the line of data after the editing functions have been performed. The editing functions are specified in the DLINT macro instruction.
2. If image mode is specified (IMAGE=YES in the DLINT macro instruction), the data record contains two types of information: field length and data from the document. The first 28 bytes contain 14 two-byte entries that indicate the length of each field in the record. If the number of fields is less than 14, the entries for the rightmost unused fields contain EBCDIC zeros (X'F0F0'). The data read from the document follows the field length entries, beginning in the 29th byte.

If the number of characters in the data record is less than the space allowed for the input record, the unused rightmost portion of the record is padded with blanks (X'40').

Document Control and Marking
DOS/VS 3886 support also provides methods of:

- Changing format records
- Ejecting and stacker selecting documents
- Performing timing mark checks
- Line and page marking documents that have been read.

All format records are created in separate assemblies; they must be cataloged in the core image library before they can be used for processing documents. You can change format records during program execution by using the SETDEV macro instruction.

Line and page marking on the 3886 requires a special feature. For more information on this feature, see *IBM 3886 Optical Character Reader Component Description and Operating Procedures*, GA21-9147.

Bytes	Field Name	Field Contents	
0-1	Document line number	Indicates the two-character decimal line number, as specified in the READmacro instruction.	
2-3	Line format record number	Indicates the two-character decimal line format record number used to read this line of data. It is the same number specified in the LFR parameter of the associated DLINT macro instruction.	
4	Read sequence number	Indicates the number of times this line has been read. If the line is read more than nine times, this byte contains X'F9.' If a READ macro instruction is issued but end of page is found before a line, this byte contains X'F0.'	
5	Line information indicator	*Hex Contents*	*Meaning*
		F0	The line is not blank and contains no errors.
		F1	The line is blank.
		F2	A reject character and/or wrong length field occurred in a critical field. If image mode is used, F2 indicates only reject characters.
		F3	The group erase symbol was encountered on the line. The data record contains blanks, X'40.'
		F4	A reject character and/or wrong length field occurred in a non-critical field. If image mode is used, F4 indicates only reject characters.
		F5	End of page (EOP) occurred on the requested read operation. All other bytes in the header data contain X'F0.' All bytes in the recognition data contain blanks, X'40.'
		F6	A reject character and/or wrong length field occurred in both a critical field and a non-critical field. If image mode is used, F6 indicates only reject characters.

Figure 5.10. Header Record Contents (Part 1 of 2)

Byte	Field Name	Field Contents
6 - 19	Field information indicator	A one - byte indicator exists for each of the 14 fields allowed on a line. Byte 6 is an indicator for field 1, byte 7 for field 2, and so on. Each byte contains one of the following: *Hex Contents* — *Meaning* F0 — No errors exist in the field or the field is not present. F2 — One or more reject characters were found in this field. F4 — This field is the wrong length. (This setting cannot occur with image mode.) F6 — Both reject character and wrong length field conditions occured for this field. (This setting cannot occur with image mode.) F8 — This field was blank before any editing functions were performed. If ALBNOF was specified, F4 will be set instead of F8.

Figure 5.10. Header Record Contents (Part 2 of 2)

OMR Considerations

Format Descriptor Card for O and R Mode

If MODE=O or MODE=R is specified, a format descriptor card defining the card columns to be read, or eliminated, must be provided. This descriptor card must be the first in the data set. When it is found, an 80-byte record is built which relates to the specified format on a column-per-column basis. If the format descriptor record is not found, a message is issued to the operator and the job is terminated.

The format descriptor card is written as follows:

FORMAT (N1,N2)[,(N3,N4)...]

FORMAT must be punched in columns 2-7, followed by a blank in column 8. Operands begin in column 9 and may continue through column 71; they must be separated by a comma. Continuation cards can be specified by punching an X in column 72; coding on the next card must then begin in column 16. Both N1 and N2 must be greater than or equal to one, and less than or equal to 80. N2 must be greater than or equal to N1. If the format descriptor card is written FORMAT (N1,N2),(N3,N4),..., N2 must be less than N3. For OMR, N3 minus N2 must be greater than or equal to two.

For MODE=O, N1 indicates the first column, and N2 indicates the last column, to be read in OMR mode. Only every other column between N1 and N2 can be read in OMR mode; therefore, N1 and N2 must both have even values or both have odd values.

For MODE=R, N1 indicates the first column *not* to be read, and N2 indicates the last column *not* to be read.

For example, if the operand MODE=O is specified, and it is desired to read columns 1, 3, 5, 7, 9, 70, 72, 74, 76, 78, and 80 in OMR mode, the following format descriptor card would be used:

```
FORMAT (1,9),(70,80)
```

Or, if the operand MODE=R were specified and it were desired to read all card columns except 20 through 30 and 52 through 76, the following format descriptor card would be used:

```
FORMAT (20,30),(52,76)
```

Coding of OMR Input

The following rules apply to the coding of an input card to be read in OMR mode:

- Mark characters (characters to be read optically) must be separated by at least one column which contains neither marks nor punches. "M" in the example indicates mark characters and "ƀ" indicates the blanks:

  MƀMƀMƀƀM

- Mark characters must be separated from any columns containing punched holes (in the example indicated by "H") by at least one column which contains neither marks nor punches:

  MƀHƀHHH

- Mark characters in odd columns must be separated from mark characters in even columns by at least two columns which contain nether marks nor punches (in the example the numbers above the characters indicate card columns):

```
12345678
MƀMƀMƀM
```

OMR Data Records

Although OMR data is physically located in alternating columns the data in the I/O area is compressed into contiguous bytes. The relationship of data on card columns to the location of the data in storage is as follows:

1. If column n does not contain OMR data, the data content of column n+1 represents the contiguous byte in virtual storage which follows the column n data byte.
2. If column n does contain OMR data, the data content of column n+2 represents the contiguous byte in virtual storage which follows the column n data byte. The data contents of column n+1 is not placed in virtual storage.

3. The data content of column 1 always represents the first data byte in virtual storage.

Figure 5.11 shows how these rules apply to the data card and its format descriptor card, and the record which results from reading the data card.

When a weak mark or poor erasure is detected in a column by the IBM 3505 Card Reader, the column's data is replaced with a hexadecimal 3F (X'3F') when reading in EBCDIC mode, or two hexadecimal 3Fs (X'3F3F') when reading in column binary mode. Checking for this condition is the user's responsibility.

If X'3F' is placed in the data, a X'3F' is also placed in byte 80 of the I/O area when reading in EBCDIC mode, or in byte 160 when reading in column binary mode, to indicate OMR reading error. The user can then determine whether or not an OMR reading error occurred on the card by checking this byte. If, however, the I/O area length is less than 80 for EBCDIC mode or less than 160 for column binary mode, the X'3F' is not placed in virtual storage. In this case, to determine if a reading error occurred the user must check each OMR byte for a X'3F'.

Card column	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Card Data	P_1	P_2	ƀ	M_4	ƀ	M_6	ƀ	ƀ	M_9	ƀ	M_{11}	ƀ	P_{13}	P_{14}	P_{15}	P_{16}	P_{17}	P_{18}	P_{19}	P_{20}
Format Data	ƀ	ƀ	ƀ	F_4	–	F_6	–	ƀ	F_9	–	F_{11}	–	ƀ	ƀ	ƀ	ƀ	ƀ	ƀ	ƀ	ƀ
		Switch from punch to mark					Switch from even to odd marks					Switch from mark to punch								
Format Descriptor Card		F	O	R	M	A	T		(	4		6	)		(	9	,	1	1	)
Channel Data	P_1	P_2	ƀ	M_4	M_6	ƀ	M_9	M_{11}	P_{13}	P_{14}	P_{15}	P_{16}	P_{17}	P_{18}	P_{19}	P_{20}				

ƀ = Must have neither hole nor mark data
ƀ = Hexadecimal 40
— = May be character or blank
P_x = Punched data in column x
M_x = Mark data in column x
F_x = Format data for column x

Figure 5.11. OMR Data and Format Coding Example.

Appendix 5: The American National Standard Code for Information Interchange (ASCII)

In addition to the EBCDIC mode, DOS/VS accepts magnetic tape files written in ASCII (the American National Standard Code for Information Interchange), a 128-character, 7-bit code. The high-order bit in this 8-bit environment is zero. ASCII is based on the specifications of the American National Standards Institute, Inc. (ANSI).

Under DOS/VS, ASCII data files are processed in EBCDIC. At system generation time if ASCII=YES is specified in the SUPVR macro, two translate tables are included in the supervisor. When these tables are used, logical IOCS translates from ASCII to EBCDIC as soon as the data has been read into the I/O area. For ASCII output the data is translated from EBCDIC to ASCII just before writing the record. The address of the ASCII to EBCDIC translate table is in bytes 44-47 of the extension of the communication region for each partition. The address of the EBCDIC to ASCII table is 256 bytes higher than that of the first table. The address of the communication region extension is found in bytes 136-139 of the communication region.

Figure 5.12 shows the relative bit positions of the ASCII character set. An ASCII character is described by its column/row position in the table. The four high-order bits are listed in columns across the top of the figure; the four low-order bits are in rows along the left side. Because the letter P in ASCII is located under column 5 and row 0, it is described in ASCII notation as 5/0. ASCII 5/0 and EBCDIC X'50' represent the same binary configuration (B'0101 0000'); however, this configuration is graphically represented by P in ASCII and by & in EBCDIC. ASCII notation is always expressed in decimal; so, for example, the ASCII Z is expressed 5/10 (not 5/A).

Figure 5.13 shows ASCII to EBCDIC correspondence. For those EBCDIC characters that have no direct equivalent in ASCII, the substitute character (SUB) is provided during translation.

Note: If an EBCDIC file has been translated into ASCII, and then the user translates back into EBCDIC, these substitute characters may not receive the expected value.

b7 → b6 → b5 →					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
Bits[3] b4	b3	b2	b1	Column → / Row ↓	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DE1	![1]	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	¦
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^[2]	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

[1] The graphic | (Logical OR) may also be used instead of ! (Exclamation Point).

[2] The graphic ¬ (Logical NOT) may also be used instead of (Circumflex).

[3] The 7 bit ASCII code expands to 8 bits when in storage by adding a high order 0 bit.

Example: Pound sign (#) is represented by

	b7	b6	b5	b4	b3	b2	b1
0	0	1	0	0	0	1	1

Control Character Representations

NUL	Null	DLE	Data Link Escape (CC)
SOH	Start of Heading (CC)	DC1	Device Control 1
STX	Start of Text (CC)	DC2	Device Control 2
ETX	End of Text (CC)	DC3	Device Control 3
EOT	End of Transmission (CC)	DC4	Device Control 4
ENQ	Enquiry (CC)	NAK	Negative Acknowledge (CC)
ACK	Acknowledge (CC)	SYN	Synchronous Idle (CC)
BEL	Bell	ETB	End of Transmission Block (CC)
BS	Backspace (FE)	CAN	Cancel
HT	Horizontal Tabulation (FE)	EM	End of Medium
LF	Line Feed (FE)	SUB	Substitute
VT	Vertical Tabulation (FE)	ESC	Escape
FF	Form Feed (FE)	FS	File Separator (IS)
CR	Carriage Return (FE)	GS	Group Separator (IS)
SO	Shift out	RS	Record Separator (IS)
SI	Shift in	US	Unit Separator (IS)
		DEL	Delete

(CC)	Communication Control
(FE)	Format Effector
(IS)	Information Separator

Special Graphic Characters

SP	Space	<	Less Than
!	Exclamation Point	=	Equals
|	Logical OR	>	Greater Than
"	Quotation Marks	?	Question Mark
#	Number Sign	@	Commercial At
$	Dollar Sign	[	Opening Bracket
%	Percent	\	Reverse Slant
&	Ampersand	]	Closing Bracket
'	Apostrophe	^	Circumflex
(	Opening Parenthesis	¬	Logical NOT
)	Closing Parenthesis	_	Underline
*	Asterisk	`	Grave Accent
+	Plus	{	Opening Brace
,	Comma	¦	Vertical Line (This graphic is stylized to distinguish it from Logical OR)
-	Hyphen (Minus)	}	Closing Brace
.	Period (Decimal Point)	~	Tilde
/	Slant		
:	Colon		
;	Semicolon		

Figure 5.12. ASCII character set

ASCII					EBCDIC				
Character	Col	Row	Bit Position		Col (in Hex)	Row (in Hex)	Bit Position		Comments
NUL	0	0	0000	0000	0	0	0000	0000	
SOH	0	1	0000	0001	0	1	0000	0001	
STX	0	2	0000	0010	0	2	0000	0010	
ETX	0	3	0000	0011	0	3	0000	0011	
EOT	0	4	0000	0100	3	7	0011	0111	
ENQ	0	5	0000	0101	2	D	0010	1101	
ACK	0	6	0000	0110	2	E	0010	1110	
BEL	0	7	0000	0111	2	F	0010	1111	
BS	0	8	0000	1000	1	6	0001	0110	
HT	0	9	0000	1001	0	5	0000	0101	
LF	0	10	0000	1010	2	5	0010	0101	
VT	0	11	0000	1011	0	B	0000	1011	
FF	0	12	0000	1100	0	C	0000	1100	
CR	0	13	0000	1101	0	D	0000	1101	
SO	0	14	0000	1110	0	E	0000	1110	
SI	0	15	0000	1111	0	F	0000	1111	
DLE	1	0	0001	0000	1	0	0001	0000	
DC1	1	1	0001	0001	1	1	0001	0001	
DC2	1	2	0001	0010	1	2	0001	0010	
DC3	1	3	0001	0011	1	3	0001	0011	
DC4	1	4	0001	0100	3	C	0011	1100	
NAK	1	5	0001	0101	3	D	0011	1101	
SYN	1	6	0001	0110	3	2	0011	0010	
ETB	1	7	0001	0111	2	6	0010	0110	
CAN	1	8	0001	1000	1	8	0001	1000	
EM	1	9	0001	1001	1	9	0001	1001	
SUB	1	10	0001	1010	3	F	0011	1111	
ESC	1	11	0001	1011	2	7	0010	0111	
FS	1	12	0001	1100	1	C	0001	1100	
GS	1	13	0001	1101	1	D	0001	1101	
RS	1	14	0001	1110	1	E	0001	1110	
US	1	15	0001	1111	1	F	0001	1111	
SP	2	0	0010	0000	4	0	0100	0000	
|①	2	1	0010	0001	4	F	0100	1111	Logical OR
"	2	2	0010	0010	7	F	0111	1111	
#	2	3	0010	0011	7	B	0111	1011	
$	2	4	0010	0100	5	B	0101	1011	
%	2	5	0010	0101	6	C	0110	1100	
&	2	6	0010	0110	5	0	0101	0000	
'	2	7	0010	0111	7	D	0111	1101	
(	2	8	0010	1000	4	D	0100	1101	
)	2	9	0010	1001	5	D	0101	1101	
*	2	10	0010	1010	5	C	0101	1100	
+	2	11	0010	1011	4	E	0100	1110	
,	2	12	0010	1100	6	B	0110	1011	
-	2	13	0010	1101	6	0	0110	0000	Hyphen, Minus
.	2	14	0010	1110	4	B	0100	1011	
/	2	15	0010	1111	6	1	0110	0001	
0	3	0	0011	0000	F	0	1111	0000	
1	3	1	0011	0001	F	1	1111	0001	
2	3	2	0011	0010	F	2	1111	0010	
3	3	3	0011	0011	F	3	1111	0011	
4	3	4	0011	0100	F	4	1111	0100	
5	3	5	0011	0101	F	5	1111	0101	
6	3	6	0011	0110	F	6	1111	0110	
7	3	7	0011	0111	F	7	1111	0111	
8	3	8	0011	1000	F	8	1111	1000	
9	3	9	0011	1001	F	9	1111	1001	
:	3	10	0011	1010	7	A	0111	1010	
;	3	11	0011	1011	5	E	0101	1110	
<	3	12	0011	1100	4	C	0100	1100	
=	3	13	0011	1101	7	E	0111	1110	
>	3	14	0011	1110	6	E	0110	1110	
?	3	15	0011	1111	6	F	0110	1111	

Figure 5.13. ASCII to EBCDIC correspondence (Part 1 of 2)

ASCII				EBCDIC			
Character	Col	Row	Bit Pattern	Col (in Hex)	Row (in Hex)	Bit Pattern	Comments
@	4	0	0100 0000	7	C	0111 1100	
A	4	1	0100 0001	C	1	1100 0001	
B	4	2	0100 0010	C	2	1100 0010	
C	4	3	0100 0011	C	3	1100 0011	
D	4	4	0100 0100	C	4	1100 0100	
E	4	5	0100 0101	C	5	1100 0101	
F	4	6	0100 0110	C	6	1100 0110	
G	4	7	0100 0111	C	7	1100 0111	
H	4	8	0100 1000	C	8	1100 1000	
I	4	9	0100 1001	C	9	1100 1001	
J	4	10	0100 1010	D	1	1101 0001	
K	4	11	0100 1011	D	2	1101 0010	
L	4	12	0100 1100	D	3	1101 0011	
M	4	13	0100 1101	D	4	1101 0100	
N	4	14	0100 1110	D	5	1101 0101	
O	4	15	0100 1111	D	6	1101 0110	
P	5	0	0101 0000	D	7	1101 0111	
Q	5	1	0101 0001	D	8	1101 1000	
R	5	2	0101 0010	D	9	1101 1001	
S	5	3	0101 0011	E	2	1110 0010	
T	5	4	0101 0100	E	3	1110 0011	
U	5	5	0101 0101	E	4	1110 0100	
V	5	6	0101 0110	E	5	1110 0101	
W	5	7	0101 0111	E	6	1110 0110	
X	5	8	0101 1000	E	7	1110 0111	
Y	5	9	0101 1001	E	8	1110 1000	
Z	5	10	0101 1010	E	9	1110 1001	
[	5	11	0101 1011	4	A	0100 1010	
\	5	12	0101 1100	E	0	1110 0000	Reverse Slant
]	5	13	0101 1101	5	A	0101 1010	
¬②	5	14	0101 1110	5	F	0101 1111	Logical NOT
_	5	15	0101 1111	6	D	0110 1101	Underscore
`	6	0	0110 0000	7	9	0111 1001	Grave Accent
a	6	1	0110 0001	8	1	1000 0001	
b	6	2	0110 0010	8	2	1000 0010	
c	6	3	0110 0011	8	3	1000 0011	
d	6	4	0110 0100	8	4	1000 0100	
e	6	5	0110 0101	8	5	1000 0101	
f	6	6	0110 0110	8	6	1000 0110	
g	6	7	0110 0111	8	7	1000 0111	
h	6	8	0110 1000	8	8	1000 1000	
i	6	9	0110 1001	8	9	1000 1001	
j	6	10	0110 1010	9	1	1001 0001	
k	6	11	0110 1011	9	2	1001 0010	
l	6	12	0110 1100	9	3	1001 0011	
m	6	13	0110 1101	9	4	1001 0100	
n	6	14	0110 1110	9	5	1001 0101	
o	6	15	0110 1111	9	6	1001 0110	
p	7	0	0111 0000	9	7	1001 0111	
q	7	1	0111 0001	9	8	1001 1000	
r	7	2	0111 0010	9	9	1001 1001	
s	7	3	0111 0011	A	2	1010 0010	
t	7	4	0111 0100	A	3	1010 0011	
u	7	5	0111 0101	A	4	1010 0100	
v	7	6	0111 0110	A	5	1010 0101	
w	7	7	0111 0111	A	6	1010 0110	
x	7	8	0111 1000	A	7	1010 0111	
y	7	9	0111 1001	A	8	1010 1000	
z	7	10	0111 1010	A	9	1010 1001	
{	7	11	0111 1011	C	0	1100 0000	
¦	7	12	0111 1100	6	A	0110 1010	Vertical Line
}	7	13	0111 1101	D	0	1101 0000	
~	7	14	0111 1110	A	1	1010 0001	Tilde
DEL	7	15	0111 1111	0	7	0000 0111	

① The graphic ! (Exclamation Point) can be used instead of | (Logical OR).

② The graphic ^ (Circumflex) can be used instead of ¬ (Logical NOT).

Figure 5.13. ASCII to EBCDIC correspondence (Part 2 of 2)

ASCII Tape Files

Records of an ASCII tape file may be preceded by a *block prefix* with a length of 0-99 bytes.

On ASCII variable-length (format D) input, if the block prefix is four bytes long and contains the length of the physical record (unpacked decimal format), the block prefix may be used to check the physical record length. On ASCII variable-length output, DOS/VS can create a block prefix that is four bytes long containing the length of the physical record. The length of the physical record includes the length of the block prefix. For fixed and undefined records, DOS/VS ignores the block prefix on input and does not restore this field on output.

The length of an ASCII physical record also includes any padding characters present. For certain operating systems when the block size of a file is required to be a multiple of a constant value n, unused end positions of physical records may be filled with padding characters, thus ensuring that all blocks conform with the required length. DOS/VS accepts these padding characters (corresponding to EBCDIC X'5F') on ASCII input, but it does not perform any padding operation on output.

All ASCII records are processed by DOS/VS in EBCDIC, and the system takes care of the conversion and translation.

ASCII Variable-Length (Format D)

Format D provides for variable-length records in variable-length blocks, each of which may include a block length. If the block prefix is four bytes long and the physical record length is contained in the block prefix field, the system performs length checking on input (if specified by the user).

Figure 5.14 shows format D records. The first four bytes of the logical record (dddd) contain the length of that logical record. The system makes use of this record length information in deblocking and blocking.

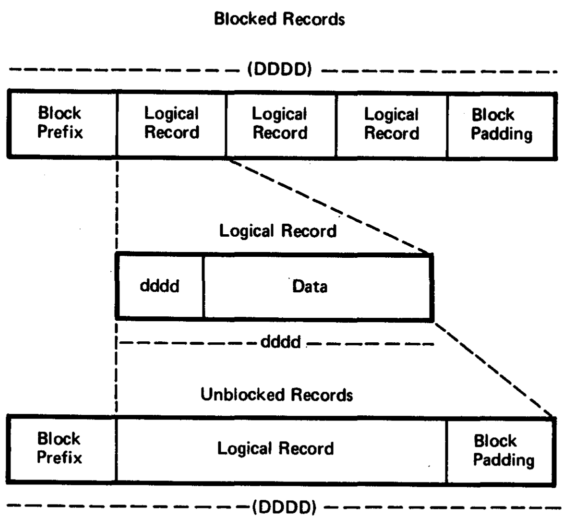


Figure 5.14. Format D records (ASCII)

In format D, the block length (DDDD) may be contained in the block prefix (in unpacked decimal format) if this field is four bytes long. In that case the physical block length is automatically provided when the file is written. On output the system only supports a block prefix with a length of zero or four bytes. Although the block prefix does not appear in the logical record furnished to the user, input and output areas must be large enough to accomodate it and any padding characters within the physical record.

If a padding character is detected by IOCS in the first position of the four-byte record length descriptor field (required as the first four bytes of each logical record), all remaining bytes in the block are bypassed. The next logical record is then retrieved from the next block.

Glossary

This glossary defines most of the terms used in this book. For a more complete list of data processing terms, the reader is referred to the IBM Data Processing Glossary, GC20-1699.

IBM is grateful to the American National Standards Institute (ANSI) for permission to reprint its definitions from the American National Standard Vocabulary for Information Processing (Copyright © 1970 by American National Standards Institute, Incorporated), which was prepared by Subcommittee X3K5 on Terminology and Glossary of American National Standards Committee X3. ANSI definitions are preceded by an asterisk.

Note: The definitions of some of the terms below are different from the definitions in the IBM Data Processing Glossary. This is because the terminology used in that book is less familiar to users of earlier versions of DOS. Rather than to change the familiar terminology and perhaps confuse the reader, we have, for the purpose of this book, chosen a somewhat older terminology in some instances.

Access Method:
Any of the data management techniques (sequential, indexed-sequential, or direct) available for transferring data between virtual storage and an input/output device.

Access Method Services:
A multifunction service program that defines VSAM files and allocates space for them, converts indexed-sequential files to key-sequenced files with indexes, modifies file attributes in the catalog, reorganizes files, facilitates data portability between operating systems, creates backup copies of files and indexes, helps make inaccessible files accessible, and lists the records of the files and catalogs.

ASCII (American National Standard Code for Information Interchange):
A 128-character, 7-bit code. The high-order bit in the System/370 8-bit environment is zero.

Block:
1. To group logical records physically for the purpose of saving storage space in external storage, or increasing the efficiency of access or processing.
2. See: *Physical block.*

Block Prefix:
1. An optional, 0-99 byte field preceding an ASCII record on magnetic tape. It contains data specified by the user or, for variable-length (ASCII Format D) records, the physical block length.
2. (general) Additional data stored in one or more fields that precede application data. A block prefix is used in those cases where an access method requires additional information about the data following.

Buffer:
1. A storage device in which data is assembled temporarily during data transfer between virtual storage and an input/output device. An example is the 2821 control unit, a control and buffer storage unit for card readers, card punches, and printers.
2. A portion of virtual storage into which data is read, or from which data is written. Synonymous with *I/O area.*

CCB: See: *Command Control Block.*

CCW: See: *Channel Command Word.*

Chained records:
A method of grouping logical records on a diskette in order to enhance access or processing efficiency.

Channel Command Word (CCW):
A doubleword at the location in real storage specified by the channel address word. One or more CCWs make up the *Channel program.*

Channel Program:
One or more Channel Command Words (CCWs) that control a specific sequence of channel operations. Execution of the specific sequence is initiated by a single SIO machine instruction.

Checkpoint Record:
A record containing the status of the job and of the system at the time the checkpoint routine writes the record. This record provides the necessary information for restarting a job without returning to the beginning of the job.

Checkpoint/Restart:
A means of restarting execution of a program at some point other than the beginning. When a CHKPT macro is issued in a problem program, checkpoint records are created. These records contain the status of the job and the system. When it is desired to restart a program at a point other than the beginning of the job, the restart procedure uses the checkpoint records to re-initialize the system.

Checkpoint Routine:
A routine that records information for a checkpoint.

Command Control Block (CCB):
A 16-byte field required for each channel program executed by physical IOCS (PIOCS). This field is used for the communication between PIOCS and the problem program.

Control Program:
A group of programs that provides functions such as the handling of I/O operations, error detection and error recovery, program loading, and communication between the program and the operator. IPL, supervisor, and job control make up the control program in DOS/VS.

Control Section:
The smallest separately relocatable unit of a program; that portion of text specified as an entity, all elements of which are to be loaded into contiguous virtual storage locations.

Data Compression:
The process of changing the representation of data into a compressed representation, by replacing a string of repetitive characters by a number which indicates the amount of characters eliminated. Data compression may be used for saving space in virtual storage or auxiliary storage.

Data Conversion:
The process of changing data from one form of representation to another.

Data Encoding:

The process of changing the representation of data into a coded representation, through a translation routine. Data encoding may be used for saving space in main storage or auxiliary storage, or for security.

Data Set Security:
A feature that provides protection for disk files. A secured file cannot be accidentally accessed by a problem program.

Device Independence:
The capability of a program to process the same type of data on different device types (punched card devices, printers, magnetic tape, or disk).

DTF (Define The File) Macro:
A macro which describes the characteristics of an input/output file, indicates the type of processing for the file, and specifies the virtual storage areas and routines to process the file. These characteristics are described using the appropriate parameters in the keyword operands of the DTF macro.

Extent:
A continuous space on a direct access storage device, occupied by or reserved for a particular file.

**File:*
A collection of related records treated as a unit. For example, one line of an invoice may form an item, a complete invoice may form a record, the complete set of such records may form a file, the collection of inventory control files may form a library, and the libraries used by an organization are known as its data bank.) See: *Logical file, Physical file.*

Fixed-Length Record:
A record having the same length as all other records with which it is logically or physically related. (Contrasted with *variable-length record.*)

Header Label:
A file label that precedes the data records on a unit of recording media.

I/O (Input/Output) Area:
A portion of virtual storage into which data is read or from which data is written. See: *buffer.*

IOCS (Input/Output Control System):
A group of macros and the routines which process them, provided by IBM for handling the transfer of data between virtual storage and external storage devices.

Load Point:
The beginning of the recording area on a reel of magnetic tape.

Logic Module:
The logical IOCS routine that provides an interface between a processing program and physical IOCS.

Logical File:
A collection of one or more logical records, treated as a unit by user-written data processing routines in application programs.

Logical Record:

A record identified from the standpoint of its content, function, and use rather than its physical attributes; that is, one which is meaningful with respect to a program. (Contrasted with *Physical block.*)

Multifile Volume:
A unit of recording media, such as a magnetic tape reel or disk pack, that contains more than one file.

Multivolume File:
A file which, due to its size, requires more than one unit of recording media (such as a magnetic tape reel or disk pack) to contain the entire file.

Nonstandard Labels:
Labels that do not conform to the IBM-standard label conventions. They can be of any length, need not have a specified identification, and do not have a fixed format.

Operating System:
A collection of programs that enables a data processing system to supervise its own operations, automatically calling in programs, routines, language processors, and data as needed for continuous throughput of a series of jobs.

Physical Block:
A collection of one or more stored records read from or written to external storage as a unit. The collection of stored records as a whole may be expanded by additional control information required by the storage device or the operating system.

Physical File:
A collection of one or more physical blocks, stored in external storage in one of several prescribed arrangements, and described by control information to which the system has access.

Private Library:
A relocatable, core image, or source statement library that is separate and distinct from the system library.

Problem Program:
1. The user's object program. It can be produced by any of the language translators. It consists of instructions and data necessary to solve his data-processing problem or to achieve a certain result.
2. A general term for any routine that is executed in the data processing system's problem state; that is, any routine that does not contain privileged operations. (Contrasted with *Supervisor.*)

Processing Program:
A general term for any program that is both loaded and supervised by the control program. This includes IBM-supplied programs such as language processors, linkage editor, librarian, sort/merge, and system utilities, as well as user-supplied programs. (Contrasted with *control program.*)

Real Storage:
All addressable storage from which instructions can be executed or from which data can be loaded directly into registers.

Record:

A general term for any unit of data that is distinct from all others when considered in a particular context. See: *Logical record, Stored record, Physical block.*

Relocatable:
The attribute of a module or control section whose address constants can be modified to compensate for a change in origin.

Resource:
Any facility of the computing or operating system required by a job or task. This includes storage, I/O devices, the central processing unit, files, and the control and processing programs.

Restart:	See: *Checkpoint/restart.*
SDL:	See: *System Directory List.*

Self-Relocating:
A programmed routine that is loaded at any doubleword boundary and can adjust its address values so as to be executed at that location.

Self-Relocating Program:
A program that can be loaded into any area of virtual storage by having an initialization routine to modify al address constants at object time.

Shared Virtual Area:
An area located in the highest addresses of virtual storage. It can contain a system directory list of highly used phases and resident programs that can be shared between partitions.

Stored Record:
A logical record in a format in which it is manipulated by a logic module. Thus, a stored record may be a logical record which is expanded with any additional control information as required for an access method.

Supervisor:
A component of the control program. It consists of routines to control the functions of program loading, machine interruptions, external interruptions, operator communications, and physical IOCS requests and interruptions. The supervisor alone operates in the privileged (supervisor) state. It coexists in real storage with problem programs.

SVA: See: *Shared Virtual Area.*

Symbolic I/O Assignment:
A means by which problem programs can refer to an I/O device by a symbolic name. Before a program is executed, job control can be used to assign a specific I/O device to that symbolic name.

System Directory List:
A list containing directory entries of highly used phases and of all phases resident in the shared virtual area. This list is placed in the shared virtual area.

Telecommunications:
A general term expressing data transmission between remote locations.

Teleprocessing:

A term associated with IBM telecommunication systems expressing data transmission between a computer and remote devices.

Track Hold:
A function for protecting DASD tracks that are currently being processed. When track hold is specified in the DTF macro for that file, a track that is being modified by a task in one partition cannot be concurrently accessed by a task or subtask in another partition.

Undefined Record:
A record having an unspecified or unknown length.

Variable-Length Record:
A record having a length independent of the lengths of other records with which it is logically or physically associated. (Contrasted with *fixed length record.*)

Virtual Storage:
Addressable space that appears to the user as real storage, from which instructions and data are mapped into real storage locations. The size of virtual storage is limited by the addressing scheme of the computing system and by the amount of auxiliary storage available, rather than by the actual number of real storage locations.

Volume:
That portion of a single unit of storage media that is accessible to a read-write mechanism. For example, a reel of magnetic tape, or a disk pack on an IBM 2314 disk storage drive.

Index

GC33-5372-2

DOS/VS Data Management Guide Printed in U.S.A. GC33-5372-2

IBM®

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

DOS/VS
Data Management Guide

READER'S COMMENT FORM

GC33-5372-2

This sheet is for comments and suggestions about this manual. We would appreciate *your* views, favorable or unfavorable, in order to aid us in improving *this* publication. This form will be sent directly to the author's department. Please include your name and address if you wish a reply. Contact your IBM branch office for answers to technical questions about the system or when requesting additional publications. Thank you.

Name

Address

What is your occupation?

How did you use this manual?

As a reference source

As a classroom text

As a self-study text

Your comments* and suggestions:

*** We would especially appreciate your comments on any of the following topics:**

Clarity of the text	Accuracy	Index	Illustrations	Appearance	Paper
Organization of the text	Cross-references	Tables	Examples	Printing	Binding

YOUR COMMENTS, PLEASE . . .

This manual is part of a library that serves as a reference source for systems analysts, programmers and operators of IBM systems. Your answers to the questions on the back of this form, together with your comments, will help us produce better publications for your use. Each reply will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Please note: Requests for copies of publications and for assistance in utilizing your IBM system should be directed to your IBM representative or to the IBM sales office serving your locality.

CUT ALONG THIS LINE

Fold Fold

FIRST CLASS
PERMIT NO. 1359
WHITE PLAINS, N. Y.

BUSINESS REPLY MAIL

NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM Corporation
1133 Westchester Avenue
White Plains, N.Y. 10604

Attention: Department 813 BP

Fold Fold

IBM®

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

GC33-5372-2

DOS/VS Data Management Guide Printed in U.S.A. GC33-5372-2

IBM

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)