60496000

CONTROL DATA CORPORATION

CYBER RECORD MANAGER GUIDE FOR USERS OF COBOL

CONTROL DATA®
CYBER 170 SERIES
CYBER 70 MODELS 71, 72, 73, 74
6000 SERIES
COMPUTER SYSTEMS

REVISION RECORD	
REVISION	**DESCRIPTION**
A	Manual released.
(06-18-76)	
Publication No. 60496000	

REVISION LETTERS I, O, Q AND X ARE NOT USED

Address comments concerning this manual to:

CONTROL DATA CORPORATION
Software Documentation
215 MOFFETT PARK DRIVE
SUNNYVALE, CALIFORNIA 94086

or use Comment Sheet in the back of this manual

LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed

Page	Revision	Software Feature Change
Cover	-	
Title Page	-	
ii thru xi	A	
1-1	A	
2-1 thru 2-7	A	
3-1 thru 3-10	A	
4-1 thru 4-6	A	
5-1 thru 5-6	A	
6-1 thru 6-14	A	
7-1 thru 7-14	A	
8-1 thru 8-11	A	
9-1 thru 9-21	A	
A-1 thru A-4	A	
B-1, B-2	A	
C-1 thru C-6	A	
Index-1, Index-2	A	
Cmt Sheet	-	
Return Env	-	
Back Cover	-	

Page	Revision	Software Feature Change

PREFACE

This user guide describes the input/output system associated with the CONTROL DATA® COBOL 4 compiler. The name of the input/output system, which is common to many CDC® software products, is CYBER Record Manager Version 1. Both COBOL 4 and CYBER Record Manager are implemented for the Control Data CYBER 170, CYBER 70 Models 71, 72, 73, 74, and 6000 Series Computer Systems, under the control of the NOS/BE 1 and NOS 1 operating systems.

It is assumed that the reader of this user guide is a COBOL programmer with intermediate experience and some knowledge of CDC systems. How to compile and execute a COBOL 4 program is not discussed, nor is the subject of compatibility with the ANSI standard for COBOL (X3.23-1968), with some exceptions. Complete formats of COBOL statements are not as a rule given, and any formats illustrated do not distinguish optional from required elements. For all this information, the reader is referred to the COBOL 4 Reference Manual. Although several complete COBOL programs are provided to illustrate various aspects of COBOL input/output, the operating system control statements that would normally accompany these programs in an actual job are not included. Where the results of compiling or executing a program depend on control statements, an indication is so made. Control statements processed by CYBER Record Manager, such as the FILE control statement, are discussed, however.

Section 2 of this user guide provides an overview of CYBER Record Manager functions and their relation to COBOL. Specific topics discussed include record types and USE procedures. Sections 3 through 8 describe the six file organizations available to COBOL 4 users and the COBOL statements that utilize them. Section 9 discusses the file information table, its usage by COBOL and setting its fields through the FILE control statement. Appendixes include charts showing character sets supported and a description of the various CYBER Record Manager utilities available to supplement standard COBOL file processing.

Publications containing more detail about COBOL, CYBER Record Manager, and the various operating systems include:

Publication	Publication Number
COBOL Version 4 Reference Manual	60496800
Record Manager Version 1 Reference Manual	60495700
NOS/BE 1 Reference Manual	60493800
NOS 1.0 Reference Manual (Volume 1)	60435400

This product is intended for use only as described in this document. Control Data cannot be responsible for the proper functioning of undescribed features or parameters.

CONTENTS

APPENDIXES

FIGURES

TABLES

NOTATIONS USED IN THIS MANUAL

The following notational conventions are used in the descriptions of formats for COBOL statements:

[ ]	Brackets enclose optional portions of the format. The information contained within the brackets can be included or omitted at the user's option.
{ }	Braces denote a choice of items, but one of the items must be used.
. . .	Ellipses following brackets or braces indicate the enclosed items may be repeated at the user's option.
UPPERCASE	COBOL reserved words, used in the source program only as specified in the given formats, are written in uppercase.
lowercase	Lowercase terms represent words or symbols to be supplied by the user. These words and symbols are replaced in program examples by specific character strings.

INTRODUCTION TO CYBER RECORD MANAGER 1

CYBER Record Manager is the input/output processor that provides an interface between a COBOL 4 program and the operating system routines that process files on hardware devices. CYBER Record Manager permits file and record flexibility in both reading and writing so that a COBOL program can produce files that can be processed by other source language programs. Since the file formats created and recognized by CYBER Record Manager are independent of the language or processor through which CYBER Record Manager is called, files written through one source language can be read through another; thus, the COBOL user need not have a working knowledge of the FORTRAN language to use a file created by a FORTRAN program.

Six different file organizations are defined in COBOL 4 and supported by CYBER Record Manager to allow the COBOL user flexibility in file structure and usage. The organizations are:

Sequential	Direct
Relative	Indexed Sequential
Standard	Actual Key

All but sequential file organization require individual records in the file to have keys associated with them, so that a single record can be read or written by identifying its key. Three of these file organizations can optionally have multiple index structure: direct, indexed sequential, and actual key. Records in multiple index files can be accessed randomly or sequentially on any of several keys provided by the user. The most important or most frequently used key is the unique primary key; all the other keys are referred to as alternate keys. The terms multiple index files and alternate key files are interchangeable terms.

For all file organizations, COBOL language statements specified by the user are interpreted by the COBOL compiler and converted to CYBER Record Manager calls to arrange input/output processing. CYBER Record Manager, in turn, communicates with the operating system to manage the file functions involved:

File creation	Positioning
Label processing	Updating
Initialization	Termination
Data transfer	

Linkage between a COBOL program and CYBER Record Manager is established automatically when the COBOL object program is loaded. When a WRITE statement is executed, CYBER Record Manager accepts a record from the user, later returning the record in the exact format in which it was provided. Between the record's acceptance and its return, CYBER Record Manager communicates with the COBOL program and the operating system to arrange and carry out the processing required for the record. During this processing, CYBER Record Manager might add special terminators or control words to the record, or join it to other records, before writing it to a device. Although such manipulations might affect the format of the record in storage, they do not interfere with its logical integrity or its appearance to the user program.

The file structure specifications provided by the user in a COBOL program are interpreted at compile time by the COBOL compiler and communicated to CYBER Record Manager through the file information table. These specifications can be overridden in some cases, however, by the FILE control statement (section 9), which takes effect at execution time. Use of the FILE control statement is not recommended for users who are not familiar with the interactions between COBOL and CYBER Record Manager; discussion of it is therefore deferred until after these interactions have been described.

CYBER Record Manager also provides error processing for the various file organizations and applications. Input/output errors encountered in program execution are signaled and the numbers of appropriate error messages are listed in the program dayfile for user reference. Various labeling conventions are accommodated for those file organizations that permit labels. Both standard and nonstandard (user-provided) labels are processed through CYBER Record Manager; the label processing routines read labels, submit them for writing or user processing, and terminate user label processing.

All file processing requests in a COBOL 4 program are implemented through CYBER Record Manager. Based on clauses specified in the Environment and Data Divisions, and on statements in the Procedure Division, the COBOL compiler generates calls to CYBER Record Manager, which in turn requests the operating system to perform the actual input and output. Both file structure specifications and specific processing requests are translated from program statements into a form usable by CYBER Record Manager.

FILE INFORMATION TABLE

To establish communications with CYBER Record Manager, the COBOL compiler creates a file information table (FIT) for each file. The compiler places file information in the table for each file specified in the program as it encounters applicable source program statements. CYBER Record Manager uses the information in this table as a basis for requesting file processing action by the operating system. Figure 2-1 illustrates the interfaces involved in communicating input/output actions.

The file information table contains descriptions of the individual file's organization, record size and type, blocking structure, and processing options, as well as the logical file name by which the file is known to the operating system and other pertinent data such as labeling information, buffer size, and record area location.

Most of these file characteristics are specified by the COBOL programmer in Environment and Data Division clauses. For example, if the programmer specifies:

```
    SELECT PAYROLL-FILE                    (line 1)
    ASSIGN TO PAYFILE                      (line 2)
    ORGANIZATION IS SEQUENTIAL             (line 3)
    RESERVE 2 ALTERNATE AREAS              (line 4)
          .                  .
          .                  .
          .                  .
FD  PAYROLL-FILE
    LABEL RECORDS OMITTED                  (line p)
    BLOCK CONTAINS 640 CHARACTERS          (line q)
    DATA RECORD IS PAYREC.                 (line r)
```

the COBOL compiler obtains the following information from the entries and places it in the file information table for CYBER Record Manager use:

Logical file name (PAYFILE)	(from line 2)
File organization	(from line 3)
Buffer size	(from line 4)
Labeling information	(from line p)
Length of each block	(from line q)
Location of record area	(from line r)

Since the COBOL compiler creates the file information table automatically, without user intervention, it is not necessary for a programmer to know its exact format. The user should be aware, however, that file structure information provided by the COBOL compiler can be overridden with parameters on the FILE control statement. This statement changes file characteristics set in the file information table by compilation of the source language statements. Use of the FILE control statement is described in section 9; the file information table is explained in more detail in the Record Manager reference manual.

IMPLEMENTOR-NAMES FOR FILES

The logical file name used by CYBER Record Manager in calls to the operating system routines is the implementor-name the programmer specifies for a file in the ASSIGN clause. The logical file name uniquely identifies the file to the system. For example, if:

```
SELECT PAYROLL-FILE
ASSIGN TO PAYFILE
```

is specified, the COBOL compiler places the implementor-name PAYFILE in the file information table; thereafter the name PAYFILE is used by CYBER Record Manager, the operating system, and all other users external to the COBOL program in referencing that particular file. PAYROLL-FILE is used only internally in the COBOL program.

Choice of an implementor-name is left entirely to the programmer, as long as the first character is a letter, and the second through seventh characters are letters or digits (A through Z or 0 through 9).

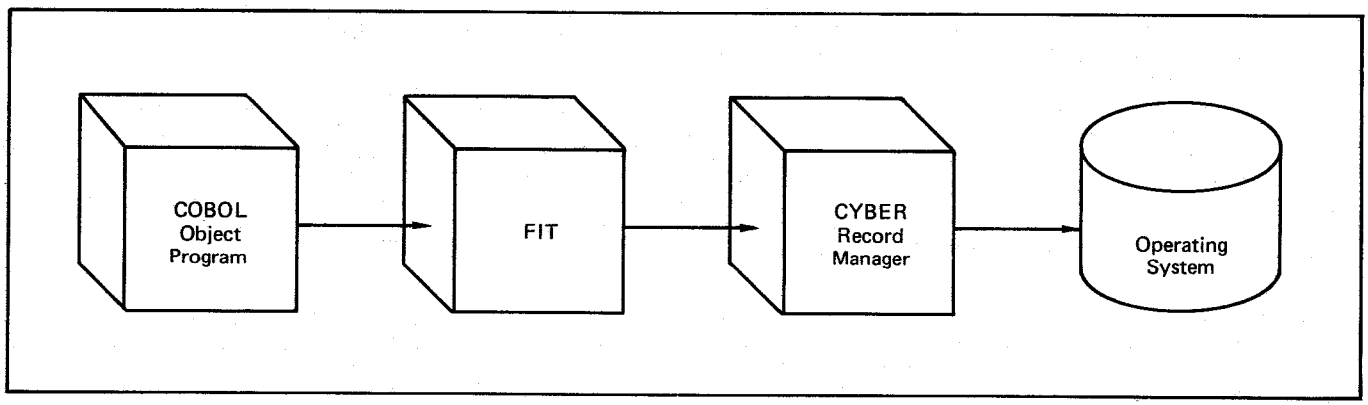

Figure 2-1. Input/Output Interfaces

The system attaches special meaning to four file names: INPUT, OUTPUT, PUNCH, and PUNCHB. In addition, the NOS 1 operating system predefines characteristics of the name P8, and NOS/BE 1 predefines the name P80C. Unless specified otherwise by the user (such as through the ROUTE control statement under NOS/BE 1), the origin or destination of these files is determined automatically when executing in batch mode:

INPUT	Implies that the file consists of card images from the job deck.
OUTPUT	Implies that the file is to be printed at the end of the job.
PUNCH	Implies that the file is to be punched in Hollerith format at the end of the job.
PUNCHB	Implies that the file is to be punched in binary format at the end of the job.
P8	Implies that the file is to be punched in absolute binary format (NOS 1 only).
P80C	Implies that the file is to be punched in absolute binary format (NOS/BE 1 only).

To access data which is part of the job deck, the SELECT statement might be as follows:

```
SELECT IN-FILE
ASSIGN TO INPUT
```

To specify a file to be printed, the statement might be:

```
SELECT REPORT-FILE
ASSIGN TO OUTPUT
```

FILE ORGANIZATIONS

There are two kinds of hardware devices on which a file can reside: tapes and mass storage. (Card punch and line printer files are mass storage files.) Of the six file organizations, only sequential files can reside on tape; all six can reside on mass storage. The six file organizations are:

Sequential
Relative
Standard
Direct
Indexed Sequential
Actual Key

The file information table fields set by the compiler for each file organization are listed in section 9.

The file organization is specified in the ORGANIZATION clause in the Environment Division. A brief description of each file organization is given below; more detailed information is presented in subsequent sections.

Sequential file organization requires that records be read and written in sequence; the records remain in the physical order in which they were written. It is most effective for files that are normally read from beginning to end; records can only be written after the last record in the file. Sequential files are implemented through CYBER Record Manager sequential file organization.

Relative file organization allows both sequential and random access of records on a mass storage device. Useful for small files with contiguous fixed length records, relative organization requires an integer key defining the record number for random access. The key is the same as the relative position of the record; for example, key 46 identifies the record in position 46. If the keys are not contiguous in the file, mass storage use is not economical since empty record slots will exist for unused keys. No CYBER Record Manager file organization corresponds exactly to relative files; they are implemented through word addressable files, with the COBOL object time routines computing the word address of the record from the integer key supplied.

Standard file organization has been retained only for compatibility with previous versions of COBOL. Standard files reside on mass storage. Record access is by means of an index, kept in central memory during program execution, which links a record's key with its physical location on the file. Records are written to the file sequentially, but cannot be read sequentially. They are read randomly according to the contents of the key item. Because of the limited processing available, and the inefficiency of the indexing scheme, standard files are not recommended for new applications. Standard files are implemented through the FORTRAN Extended mass storage input/output routines (such as READMS and WRITMS), which in turn use CYBER Record Manager word addressable file organization.

Direct file organization is useful for rapid access to files when the order of records is not important. Records are stored on mass storage randomly, according to a transformation of the primary key value known as hashing. Records can be read or written randomly by key, or they can be read sequentially. Records read sequentially, however, are not retrieved in any sorted order. Direct files are implemented through CYBER Record Manager direct access file organization; they can be multiple index files.

Indexed sequential file organization maintains records in order by primary key at all times. As records are written, they are inserted in the appropriate place in sequential order. Records can be read randomly, by specifying a key value, or sequentially, in the order of key values. Indexed sequential files are implemented through CYBER Record Manager indexed sequential file organization; they can be multiple index files.

Actual key files contain records whose key values specify the actual location of a record in the file. Thus, records are automatically in sorted order at all times. The keys are generated by CYBER Record Manager. The records can be accessed either randomly by actual key or sequentially. CYBER Record Manager does not create an index of the actual keys. Actual key files are implemented through CYBER Record Manager actual key file organization; they can be multiple index files.

MULTIPLE INDEX FILES

All files, except sequential files, must have a key associated with each record. This key, called the primary key, is used by CYBER Record Manager to locate the records in the file when the file is read or written randomly.

Additional keys, called alternate keys, can also be defined for direct, indexed sequential, and actual key files. The file can then be read using one of the alternate keys instead of the primary key. The data file is not affected by the creation of indexes for alternate keys. The indexes are kept on a separate file, called the index file, which is defined in the ASSIGN clause. A file with alternate keys defined is referred to as a multiple index file.

Alternate keys can be defined through COBOL statements when the data file is created, or the index generation utility (IXGEN) can be used to define alternate keys for an existing file. IXGEN can also be used to define new alternate keys, or to redefine or delete existing alternate keys. The IXGEN utility is discussed in appendix C.

CYBER Record Manager creates an index for each alternate key defined for a data file and updates the indexes whenever the data file is updated, or whenever the index generation utility is used to add, delete, or replace alternate keys.

When the data file is read by alternate key, the index file entries for that alternate key are searched for the desired alternate key value. The first primary key in the list of primary keys associated with that alternate key value is used to retrieve a record in the data file containing the desired alternate key value. Subsequent records also containing that alternate key value can be retrieved by executing a sequential read by alternate key.

Subsequent sections in this manual present detailed information on defining alternate keys through COBOL statements and reading by alternate key, along with general information on creating and using direct access, indexed sequential, and actual key files.

BUFFERS

Buffers are used for intermediate storage of data to be transferred between the external device and central memory. The number of records in a buffer depends on file organization and the type of blocking specified. From the buffer, each READ statement transfers one record to the input record area defined by COBOL. When a WRITE statement is executed, one record is transferred from the output record area to the buffer. For sequential files, transfer between the buffer and the external device takes place only when it is possible to completely fill or empty the buffer; for other file organizations, transfer can take place more often.

The buffer area required by a program can be assigned with the RESERVE ALTERNATE AREAS clause of the Environment Division. If the clause is omitted, or if RESERVE NO ALTERNATE AREAS is specified, the minimum buffer area is assigned. The minimum buffer area is calculated by COBOL from the File and Record Description entries. For some file organizations, more efficient processing can be achieved by assigning additional buffer areas, as described in later sections.

RECORD TYPES

The record type specification defines the format of every record in a file and enables CYBER Record Manager to determine the length of a record on a read or write. CYBER Record Manager recognizes eight record types, six of which are available to the COBOL user; D (decimal character count), F (fixed length), R (record mark), T (trailer count), W (control word), and Z (zero byte terminated). No single COBOL clause determines the record type in all cases; the COBOL compiler derives the record type indirectly from the RECORD CONTAINS clause, the OCCURS clause, or a suffix attached to the implementor-name in the ASSIGN clause. If none of these indications is present, the compiler assumes fixed length (F type) records. Table 2-1 shows the record types selected by the compiler according to RECORD CONTAINS clause and File Description entry interactions. File information table fields set for each record type are listed in section 9.

Table 2-2 shows the record types permitted in conjunction with each of the six file organizations.

TABLE 2-1. CORRESPONDENCE AMONG RECORD CONTAINS CLAUSE, RECORD DESCRIPTION ENTRY AND RECORD TYPE

RECORD CONTAINS Clause (FD Entry)	Record Description Entry		
	01 Entries of Same Length	01 Entries of Different Length	01 Entry with OCCURS . . . DEPENDING ON data-name
Clause omitted	F	F	T
RECORD CONTAINS integer CHARACTERS	F	error	error
RECORD CONTAINS integer-1 TO integer-2 CHARACTERS	F	F	T
RECORD CONTAINS integer-1 TO integer-2 CHARACTERS DEPENDING ON data-name	D	D	error
RECORD CONTAINS integer-1 TO integer-2 CHARACTERS DEPENDING ON RECORD-MARK	R	R	error

TABLE 2-2. RECORD TYPE/FILE ORGANIZATION COMBINATIONS

File Organization	Record Type					
	D	F	R	T	W	Z
Sequential	x	x	x	x	x	x
Relative		x				
Standard					x	
Direct	x	x	x	x	x	x
Indexed Sequential	x	x	x	x	x	x
Actual Key	x	x	x	x	x	x

DECIMAL CHARACTER COUNT RECORDS (D TYPE)

D type records are variable length records whose length is specified by a field within each record. D type records are defined by a RECORD CONTAINS clause of the form:

```
RECORD CONTAINS integer-1
TO integer-2 CHARACTERS
DEPENDING ON data-name
```

where integer-1 specifies the minimum record length in characters and integer-2 specifies the maximum record length in characters.

The data-name specified by the DEPENDING ON option is the name of an elementary item in the 01 level entry for the record. It must be a DISPLAY or COMPUTATIONAL-1 item. In the example in figure 2-2 RLE is the data-name in the DEPENDING ON clause; it is defined as the second elementary item of data record PAYREC.

```
FD  PAYROLL
        LABEL RECORDS OMITTED
        RECORD CONTAINS 11 TO 80
        CHARACTERS DEPENDING ON RLE
        DATA RECORD IS PAYREC.
01  PAYREC.
        02  SOC-SEC-NO PIC 9(9).
        02  RLE PIC 9(2).
        02  NAME-ADDRESS.
            03  N-CHARACTER PIC X
                OCCURS 69 TIMES.
```

Figure 2-2. D Type Records

The data-name specified by the DEPENDING ON option must be located within the fixed-length portion of the record and have a length of six or fewer characters. The COBOL program sets the data-name to the desired record length prior to record output; if it contains an integer outside the range defined in the RECORD CONTAINS clause and an input/-output statement is executed, a diagnostic is issued and the program is aborted. If DEPENDING ON RLE were omitted in the example, the compiler would default to a record type of F, and a fixed record length of 80, as calculated from the 01 level entry for PAYREC.

FIXED LENGTH RECORDS (F TYPE)

Fixed length records all contain the same number of characters. They are the record type selected by COBOL when the Data Division entries for a file contain no indication that the records in the file can vary in length; that is, when the DEPENDING ON option is omitted from the RECORD CONTAINS clause in the File Description entry (or the clause is omitted entirely), and the OCCURS . . . DEPENDING ON option is omitted from the Record Description entry.

The length of all records in a file with fixed length records is computed by COBOL at compile time as the sum of all items in the 01 level Record Description entry for the file. If different 01 level entries for the same file specify record formats with different lengths, the largest length specified is used as the length for all records in the file. In this case, the same input and output record areas are used for all record formats in the file, and the length of the largest group item is used as the length of the file's working storage area.

COBOL selects F type records for a file if the RECORD CONTAINS clause is omitted, or is of the form:

```
RECORD CONTAINS integer CHARACTERS
```

or of the form:

```
RECORD CONTAINS integer-1 TO
integer-2 CHARACTERS
```

The integer values specified are not used to compute the record length; they may be included for documentary purposes. If the form:

```
RECORD CONTAINS integer CHARACTERS
```

is used, and different lengths are specified in 01 level entries, an error results.

In the example shown in figure 2-3, the fixed length of each record in the file PAYFILE is calculated as 80 characters; the RECORD CONTAINS specification is not used.

```
FD  PAYFILE
        LABEL RECORDS OMITTED
        RECORD CONTAINS 209 CHARACTERS
        DATA RECORD PAYREC.
01  PAYREC.
        02  SOC-SEC-NO PIC 9(9).
        02  RATE  PIC 9(3)V99.
        02  HRS PIC 99V9.
        02  FILLER PIC X(63).
```

Figure 2-3. F Type Records

RECORD-MARK RECORDS (R TYPE)

Record-mark records are of varying lengths and are terminated by a special character. The right bracket character] is assumed as the record mark unless the FILE control statement defines another with the RMK parameter (see section 9).

To specify the record mark, the programmer uses the figurative constant RECORD-MARK, as follows:

```
RECORD CONTAINS 15 TO 80
CHARACTERS DEPENDING ON
RECORD-MARK
```

The record-mark character must then be placed in the last character position of the record by a MOVE statement preceding the WRITE statement for the record.

In the example shown in figure 2-4, the position of the record-mark is specified by the entry:

```
03  N-CHARACTER PIC X
      OCCURS 66 TIMES.
```

The record-mark would be placed with a statement of the form:

```
MOVE RECORD-MARK
TO N-CHARACTER (sub).
```

prior to writing the record, where sub is an identifier or constant with a value from 1 to 66.

```
FD  PAYROLL
      LABEL RECORDS OMITTED
      RECORD CONTAINS 15 TO 80
      CHARACTERS DEPENDING ON
      RECORD-MARK
      DATA RECORD IS PAYREC.
01  PAYREC.
      02  SOC-SEC-NO PIC 9(9).
      02  EMPLOYEE-NO PIC X(5).
      02  NAME-ADDRESS.
            03  N-CHARACTER PIC X
                  OCCURS 66 TIMES.
```

Figure 2-4. R Type Records

TRAILER COUNT RECORDS (T TYPE)

Trailer count records consist of a fixed-length header followed by a variable number of fixed-length trailer items. They result when the COBOL clause:

```
OCCURS integer-1 TO integer-2
TIMES DEPENDING ON data-name
```

is specified in the Record Description 01 level entry, as shown in figure 2-5.

```
01  WEEKLY-OVERTIME-HISTORY.
      02  SOC-SEC-NO PIC 9(9).
      02  EMP-NO PIC X(5).
      02  WEEK-COUNT PIC 99.
      02  WEEKLY-ENTRY OCCURS
            1 TO 52 TIMES DEPENDING ON
            WEEK-COUNT.
            03  HRS-WORKED PIC 99V9.
            03  RATE PIC 99V99.
```

Figure 2-5. T Type Records

The data items defined by the OCCURS clause (the occurrences of WEEKLY-ENTRY in the example) are called trailer items; in the actual records, a variable number of these items occur, identical in length but different in content. The data-name that is referenced in the DEPENDING ON option (WEEK-COUNT in the example) is called the trailer count field; it must be an elementary DISPLAY or COMPUTATIONAL-1 item. When a record is read or written, the value contained in the trailer count field defines the number of trailer items in the record. This number, along with the fixed length of the header (which consists of all the data items before the trailer items) determines the length of the record.

CONTROL WORD RECORDS (W TYPE)

Control word records are preceded by a system-generated header word that indicates the record length. Records can be any length; the varying sizes are differentiated in the Record Description 01 level entries included in the File Description. The control word is processed only by CYBER Record Manager; it is generated when the record is written. The control word is then stored preceding the record. When the user issues a read request, the control word is removed before the record is returned to the user.

Control word records are specified by the suffix -FW appended to the implementor-name in the ASSIGN clause, as shown in the following example:

```
SELECT PAYROLL-FILE
ASSIGN TO PAYFILE-FW
```

The clause:

```
RECORDING MODE IS BINARY
```

must be included in the File Description entry when control word records are specified; no RECORD CONTAINS clause is required.

More than one record format can be defined for files with W type records, and the formats can be fixed or variable in length. The length of each record written is the length required by the format; records are padded only to the next word boundary.

ZERO BYTE RECORDS (Z TYPE)

Zero byte records are specified as fixed length by the COBOL programmer, but are stored in compact format. CYBER Record Manager marks the end of the record with a zero byte terminator (which consists of 12 zero bits right-justified in a central memory word) and automatically drops any trailing blanks to reduce mass storage space; when the record is retrieved, the blanks are restored and the record is returned to the user program in its original state.

An exception is that blanks are not restored when records are read from files with indexed sequential, direct access, or actual key file organization. If the record read is shorter than the input record area, the contents of the remaining portion of the record area will therefore be irrelevant. Under these circumstances, the program should blank out the input record area before each read, or use some other means to ensure the integrity of the record.

All files originally read by the card reader or destined for line printer or card punch output must be defined with zero byte records. Files assigned to INPUT, OUTPUT, or PUNCH in the ASSIGN clause need no other specification to indicate zero byte records; the COBOL compiler automatically forces Z type records for these files. Zero byte records are specified for a file other than INPUT, OUTPUT, or PUNCH by adding the suffix -FZ to the implementor name specified in SELECT . . . ASSIGN:

```
SELECT ZFILE
ASSIGN TO PAYFILE-FZ
```

Appending -FZ to the implementor-name is necessary for any file whose records were originally read through the card reader, but which does not have the implementor-name INPUT (such as a file to which the INPUT file has been copied). Also, any file referenced by the UPON option of the ACCEPT or DISPLAY statement must have zero byte records, which can be specified through the -FZ suffix.

Another way to specify Z type records for a file is through the -P suffix, which indicates the file is to be formatted for printing. When -P is appended to the implementor-name, the compiler forces zero-byte records and reserves an extra character at the beginning of each print line for carriage control. WRITE with the ADVANCING option should be used for these files to control carriage functions.

USE PROCEDURES

USE procedures allow the user to define processing in addition to that provided by the system in the areas of label handling, error checking, and key manipulation. The user provides these procedures, which are entered whenever the condition these procedures, which are entered whenever the condition specified in the USE statement is encountered.

The USE procedures must be defined in the DECLARATIVES section of the Procedure Division. Each USE procedure consists of a section header followed by a USE statement, followed by paragraphs containing the procedure to be executed.

Input/output statements can be included in a USE procedure, but must not cause any input or output to be executed for a file named in the USE statement. A file named in a USE statement cannot be a sort file.

A USE procedure must not cause execution of itself or another USE procedure.

Four variations of the USE statement are discussed here; the remainder are discussed in the COBOL 4 Reference Manual:

USE AFTER ERROR PROCEDURE	Follows input/output procedures provided by CYBER Record Manager when an error occurs.
USE BEFORE/ AFTER LABEL PROCEDURES	Follow or precede standard and nonstandard label writing and checking procedures provided by CYBER Record Manager.
USE FOR DUPLICATE KEY	Executed on encountering duplicate primary key values on indexed sequential files.
USE FOR HASHING	Used to compute randomizing key for direct files.

USE AFTER ERROR

USE AFTER ERROR PROCEDURE can be specified to indicate one or more routines to be executed following an input/output error. When such an error occurs, CYBER Record Manager returns an error code that indicates the type of error. COBOL places this error code in the special register ERROR-CODE before the USE procedure is entered. The USE statement specifies the files for which the error procedures are to be executed, as shown in figure 2-6.

```
USE AFTER ERROR PROCEDURE
file-name-1  [file-name-2]  ...
```

Figure 2-6. USE Statement for Error Processing

USE procedures are not executed for errors causing INVALID KEY execution. The numbers assigned to ERROR-CODE are the numbers associated with CYBER Record Manager diagnostics; these numbers are listed in the Record Manager reference manual.

USE BEFORE/AFTER LABEL

By specifying USE BEFORE or AFTER LABEL PROCEDURE, the user can provide additional processing to be performed before or after input labels are checked or output labels are prepared. The time when the additional processing takes place and the files to be processed are specified in the USE statement, as shown in figure 2-7.

BEFORE LABEL procedures can be used to modify the values from which the label is created, or against which it is checked. AFTER LABEL procedures can be used to process label data not included in a VALUE OF clause.

BEGINNING and ENDING specify the type of labels to be processed. If neither BEGINNING nor ENDING is specified, the USE procedures are executed for both beginning and ending labels. If neither REEL nor FILE is specified, the procedures are executed for both reel and file labels. UNIT is synonymous with REEL.

USE FOR DUPLICATE KEY

If a USE FOR DUPLICATE KEY procedure is specified for an indexed sequential file, and a duplicate primary key value is encountered during file creation or updating, the record containing the duplicate key is not written to the file;

```
USE    {BEFORE}          STANDARD
       {AFTER }

       {BEGINNING}       {REEL  }          {PROCEDURE }
       {ENDING   }       {FILE  }  LABEL   {PROCEDURES}
                         {UNIT  }

ON     file-name-1  [file-name-2]  ...
```

Figure 2-7. USE Statement for Labels

instead, the USE procedure is executed, allowing the user to decide what action is to be taken concerning the duplicate key record.

The USE statement either lists the files to which the procedure applies, or specifies that it applies to all indexed sequential files, as shown in figure 2-8.

```
USE FOR DUPLICATE KEY ON

{ALL                                  }
{file-name-1 [file-name-2] ...        }
```

Figure 2-8. USE Statement for Duplicate Keys

USE FOR HASHING

Records in direct files are stored randomly in numbered home blocks. The key for the record is mapped onto a number, and the record is stored in the corresponding block. The mapping process is called hashing; the routine which executes the mapping process is called the hashing routine. The USE FOR HASHING procedures allow the user to provide his own hashing routines. COBOL does not provide a hashing routine; if no USE FOR HASHING procedures are specified, the CYBER Record Manager default hashing routine is used. Sample program 5 (section 6) contains an example of a USE FOR HASHING procedure.

The USE statement either lists the files to which the procedure applies, or specifies that it applies to all direct files (see figure 2-9).

```
USE FOR HASHING ON

{ALL                                  }
{file-name-1  [file-name-2]  ...      }
```

Figure 2-9. USE Statement for Hashing Procedure

The hashing procedure computes a number from the key to be used by the system in locating the correct home block. When the value is computed, it is placed in the special register HASHED-VALUE, which is system-defined as a COMPUTATIONAL-1 item. The value must lie between the limits of zero and one less than the number of blocks in the file. (The number of blocks is given in the NUMBER OF BLOCKS clause of the Environment Division.) If the value is not within these limits, the INVALID KEY option is taken and the special register ERROR-CODE is set.

When a direct file is opened for INPUT or I-O (that is, the file already exists and is not being created), CYBER Record Manager checks that the hashing routine supplied is the same hashing routine that was used to write the file. A random record is read from the file, and the hashing routine supplied is called to compute the hashed value of the key. This value is compared to the actual hashed value of the key, and a diagnostic is issued if they do not agree. Therefore, the hashing routine must be executable independent of the rest of the program; it must use data only from the record itself. No data should be picked up from any other work area of the program unless the contents of that area can be assured later, when the file is opened.

In a file with sequential organization, fixed or variable length records reside on magnetic tape or mass storage in the same physical order in which they were written. Each WRITE statement causes the specified record to be stored immediately after the record written last; an end-of-information is stored following the last record on the file after it is closed. Since records are always read and written one record at a time in successive order, keys are not used on sequential files.

Sequential file organization is best suited to files that are to be read from the beginning and have records added only at the end of the file. Space is not allotted in advance for additions; the end-of-information moves as new records are appended. Record skips are not possible; a record cannot be rewritten. Tape, punch card, printer, and some mass storage files are classified as sequential; all files not resident on mass storage must have sequential organization.

RECORD TYPES

All six record types defined in COBOL can be specified for sequential files:

- D - Decimal count
- F - Fixed length
- R - Record mark
- T - Trailer item
- W - Control word
- Z - Zero byte

Specifications needed to obtain the various record types are described in section 2.

DEVICE TYPES

All files reside on one of two types of device: tape or mass storage (punch card and line printer files reside on mass storage while they are being processed by COBOL programs). In addition, tape files are divided into two categories, which differ primarily in the manner in which the size of physical records is defined:

X and I tapes (NOS 1 only), and SI tapes (NOS/BE 1 and NOS 1), as well as mass storage files, have formats in which the physical record size is predefined by the operating system. I tapes are the system default under NOS 1, and SI tapes are the default under NOS/BE 1. For these files, physical record size is the physical record unit (PRU) size for the device as shown in table 3-1.

S (stranger) and L (long record stranger) tapes have formats in which the physical record size is defined by the user; physical records on S tapes must be less than or equal to 5120 characters.

Differences between these two formats are of importance to the user only in the effect they have on the length and type of blocks (discussed below).

TABLE 3-1. PRU SIZE BY DEVICE

Device	PRU Size
Mass storage	640 characters
Coded SI tapes	1280 characters
Binary SI tapes	5120 characters
X or I tapes (NOS 1 only)	5120 characters

FILE DELIMITERS

Three kinds of boundary conditions are recognized by CYBER Record Manager on sequential files. Their exact physical representation depends on device type and other considerations (see the Record Manager reference manual), but boundaries in each category are treated identically by CYBER Record Manager. The three kinds of boundaries, in descending order of inclusiveness, are as follows:

End-of-information (EOI). Every file has exactly one end-of-information; it occurs immediately after the last data record in the file. Trailer labels on magnetic tape are past the end-of-information. On the file named INPUT, end-of-information is equivalent to a 6/7/8/9 card. End-of-information is written after the last record in a file when the file is closed with rewind.

End-of-partition (EOP). This boundary is sometimes referred to as end-of-file. For all files, an EOP or EOI encountered on input causes the AT END clause of the READ statement to be executed. End-of-partition is written when a file is closed with no rewind.

End-of-section (EOS). This boundary corresponds to the end of a system logical record. On the file named INPUT, an end-of-section is equivalent to a 7/8/9 card; for this file, an EOS encountered on input causes the AT END clause of the READ statement to be executed. On other files, an end-of-section is ignored by COBOL.

BLOCK TYPES

In sequential files, records are grouped into larger units called blocks to increase the efficiency of transfer between memory and storage. Four block types are supported by CYBER Record Manager and COBOL; the compiler derives the block type from the BLOCK CONTAINS clause or from a suffix attached to the implementor-name in the ASSIGN clause. If neither of these indications is present, a default block type is assumed which depends on the device type. The block types selected by COBOL for different combinations of device type and BLOCK CONTAINS specification are summarized in table 3-2. In general, block types C and I are more efficient than K and E because they are designed specifically to take advantage of the characteristics of the hardware on which COBOL 4 runs.

TABLE 3-2. BLOCK TYPES

BLOCK CONTAINS Clause Format	SI, I and X Tapes; Mass Storage	S and L Tapes
Clause omitted	C	K; 1 record per block
BLOCK CONTAINS integer CHARACTERS	C	E
BLOCK CONTAINS integer RECORDS	K	K
BLOCK CONTAINS integer-1 TO integer-2 RECORDS	C	E
BLOCK CONTAINS integer-1 TO integer-2 CHARACTERS	C	E

Although the BLOCK CONTAINS clause is also applicable to file organizations other than sequential, as discussed in the appropriate sections, it is used for a different purpose; the four specific block types discussed in this section are applicable only to sequential files.

K TYPE BLOCKS

K type blocks always contain a fixed number of records, regardless of record length; thus the length of a block varies according to the length of the records contained in the block, rather than the number of records. K blocking is specified in COBOL as follows:

BLOCK CONTAINS integer RECORDS

where integer is the number of records in the block. The number of records in the block is known as the blocking factor.

K block type is the default block type in COBOL for S and L tapes; if the BLOCK CONTAINS clause is omitted for files on these devices, the compiler specifies a block type of K with one record per block.

C TYPE BLOCKS

C type blocks contain a fixed number of characters per block determined by the device type. For SI, X, and I tapes, and for mass storage, the block size is the same as the PRU size (table 3-1); the COBOL compiler does not select C type blocks for S or L tapes. The size of individual records is irrelevant for C type blocks. On a write, if the record being written is longer than the space left in the current block, the block is filled with the first part of the record, and the remainder of the record is written to the next block, or in as many blocks as required.

C type blocking is selected by COBOL whenever the file resides on an SI, I, or X tape, or mass storage device, and one of the following clauses is included in the File Description entry:

BLOCK CONTAINS integer-1 CHARACTERS

BLOCK CONTAINS integer-1 TO integer-2 RECORDS

BLOCK CONTAINS integer-1 TO integer-2 CHARACTERS

as well as when the BLOCK CONTAINS clause is omitted.

Since the number of characters per block is fixed according to device type, the integer-1 and integer-2 specifications in the BLOCK CONTAINS clause are documentary only and should correspond to the actual value used by CYBER Record Manager to avoid confusion.

E TYPE BLOCKS

E type blocks, allowed on S and L tapes only, contain as many complete records as can fit within a specified range. When the COBOL compiler selects E type blocks, it defines a minimum block size in characters and a maximum block size in characters. On a write, if a block being constructed is already as large as the minimum block size, but writing the current record to the block would cause it to exceed the maximum block size, the block is terminated and the record is written to the next block. A single record never spans more than one block; thus, maximum record length must not exceed maximum block length.

When a file resides on an S or L tape, and one of the following clauses is included in the File Description entry, the COBOL compiler specifies E type blocks and calculates the minimum block size and the maximum block size:

BLOCK CONTAINS integer-1 CHARACTERS

Minimum block size is 0; maximum block size is integer-1.

BLOCK CONTAINS integer-1 TO integer-2 RECORDS

Minimum and maximum block size are calculated from integer-1, integer-2, and the maximum record size according to the record type (section 2).

BLOCK CONTAINS integer-1 TO integer-2 CHARACTERS

Minimum block size is integer-1; maximum block size is integer-2.

I TYPE BLOCKS

I type blocks always contain 5120 characters, regardless of device type, and are used only for binary records. Records in I type blocks can span more than one block, as with C type blocks. Only W type records are allowed with I type blocks.

I type blocking is specified in COBOL by suffixing -FIW to the implementor-name in the ASSIGN clause:

```
SELECT PAYROLL-FILE
ASSIGN TO PAYFILE-FIW
```

The components of the suffix -FIW have the following meanings:

F	Format is being changed from the default
I	The file is to have I type blocks
W	The file is to have W type records

Because only W type records are allowed with I type blocks, the suffix -FI is equivalent to -FIW.

If a file with I type blocks resides on magnetic tape, the clause

```
RECORDING MODE IS BINARY
```

is necessary to ensure that the correct parity is used (since the default recording mode is decimal).

BINARY RECORDING MODE

Whenever records containing binary zeros are to be written to or read from magnetic tape, the programmer must specify:

```
RECORDING MODE IS BINARY
```

If it is not specified, the COBOL compiler defaults to decimal conversion mode and the files might be read or written incorrectly.

Binary zeros occur under the following circumstances:

When type I blocks are specified. Block headers in I type blocks might contain binary zeros.

When W type records are used. The control words generated by CYBER Record Manager might contain binary zeros.

When COMPUTATIONAL-1 or COMPUTATIONAL-2 data items are specified in a record. COMPUTATIONAL-1 and COMPUTATIONAL-2 items might contain binary zeros.

When any colons (display code 00) are contained in the data.

RECORDING MODE IS BINARY is always required when the file resides on an X or I tape. Specifying RECORDING MODE IS BINARY for SI tapes is not required but is recommended because packing density and transfer rate of data are improved.

SPECIAL SYSTEM FILES

The files whose implementor-names are INPUT, OUTPUT, and PUNCH have special, predefined characteristics which differ from the default file structure assumed for other files. These characteristics are summarized in table 3-3.

Carriage control applies only to print files. For any file that is to be printed, whether or not the implementor-name is OUTPUT, the first character in every line is interpreted by the system as carriage control, and is not printed. (The first character is listed, however, when a file is listed at a terminal under NOS 1).

Any file whose block type is C and whose record type is Z can be printed or punched in coded format; if it is to be printed, allowance must be made for carriage control.

CREATING SEQUENTIAL FILES

Sequential files are created with the clauses and statements described below. Clauses and statements such as CLOSE, which must be specified for all files, and those that do not require further explanation, are omitted from the descriptive paragraphs.

ENVIRONMENT DIVISION

The Environment Division clauses used to establish a sequential file are shown in figure 3-1.

ORGANIZATION IS	Optional, default is SEQUENTIAL
RESERVE ALTERNATE AREAS	Optional

Figure 3-1. Environment Division for Creating Sequential Files

TABLE 3-3. FILE STRUCTURE ATTRIBUTES FOR SPECIAL SYSTEM FILES

File Name	Block Type	Record Type	Maximum Record Length	Carriage Control Assumed	Recording Mode
INPUT	C	Z	80	no	decimal
OUTPUT	C	Z	140	yes	decimal
PUNCH	C	Z	80	no	decimal

ORGANIZATION IS

The ORGANIZATION IS clause is optional for sequential files, since sequential files are assumed when the clause is omitted. If it is included, it should specify ORGANIZATION IS SEQUENTIAL.

RESERVE ALTERNATE AREAS

Default buffer size can and should be increased with the RESERVE ALTERNATE AREAS clause to improve sequential file processing:

```
RESERVE integer ALTERNATE AREAS
```

If input/output activity on the system is heavy, smaller buffers are more efficient; larger sizes are preferable if memory space is not at a premium. The optimum number of alternate areas to be reserved can only be determined by comparing performance results for a particular application when different buffer sizes are used. Four or five additional alternate areas generally increase efficiency for S or L tapes; more have minimal effect.

DATA DIVISION

The Data Division clauses applicable to sequential file specification are shown in figure 3-2.

BLOCK CONTAINS	Optional (see discussion of block types above)
RECORD CONTAINS	Optional (see record types, section 2)
LABEL RECORD	Required

Figure 3-2. Data Division for Sequential File

LABEL RECORD

The LABEL RECORD clause is required for all files, even though only tape files can have labels.

For unlabeled tape files and all mass storage files, this clause should specify:

```
LABEL RECORDS ARE OMITTED
```

Tape labels are either standard or nonstandard. A standard label is one whose contents conform to the American National Standards Institute format for Magnetic Tape Labels for Information Interchange, X3.27-1969. This format is described fully in the Record Manager Reference Manual. Any other label format is considered nonstandard.

If a tape has standard labels, the clause should specify:

```
LABEL RECORD IS STANDARD
```

Some fields in a standard label can be altered by means of a VALUE OF clause; for example, the specification:

```
LABEL RECORD IS STANDARD
VALUE OF DATE-WRITTEN
IS 741002
```

sets the value of the DATE-WRITTEN field.

Nonstandard labels must be provided by means of the specification

```
LABEL RECORDS ARE
data-name-1 [data-name-2] ...
```

and are discussed fully in the COBOL 4 Reference Manual.

Additional label processing may be performed through USE procedures, described in section 2.

PROCEDURE DIVISION

The statements used for sequential file creation are shown in figure 3-3.

OPEN OUTPUT	Required
WRITE	Required
CLOSE	Required
USE BEFORE/ AFTER LABEL	Optional

Figure 3-3. Procedure Division for Sequential File Creation

OPEN OUTPUT is the only form of the OPEN statement that can be used for sequential file creation:

```
OPEN OUTPUT file-name
```

When this form is specified, the file is made available for creation through writing. For tape files, any beginning label procedures specified by USE statements are executed and label writing is performed.

Beginning label procedures are not executed if the NO REWIND option of OPEN OUTPUT is specified. In this case, the system default label is written instead of a user provided label. (This option is applicable only to multiple files on one tape reel.)

The WRITE statement transfers a record from the record area in memory to a buffer for output to an external device and controls vertical spacing on printed output pages through the ADVANCING option with BEFORE or AFTER:

```
WRITE record-name AFTER
ADVANCING identifier-1 LINES
```

Every file that was opened with an OPEN statement should be closed with a CLOSE statement. However, the STOP RUN statement automatically closes up to 53 files that were opened by the COBOL program. During execution of the CLOSE statement, an end-of-partition is written after the last record in the file.

The USE BEFORE/AFTER LABEL PROCEDURE statement allows the user to provide a routine to check or prepare labels in conjunction with the procedures provided by CYBER Record Manager. USE procedures are discussed in section 2.

EXTENDING SEQUENTIAL FILES

The same Environment Division and Data Division clauses are used for extending sequential files as for creating them. Procedure Division statements used are shown in figure 3-4.

OPEN EXTEND	Required
WRITE	Required
CLOSE	Required
USE BEFORE/ AFTER LABEL	Optional

Figure 3-4. Procedure Division for Extending Sequential Files

Rewriting in place cannot be performed on an existing sequential file. Although new records cannot be inserted, they can be added at the end of an existing mass storage file. This is accomplished by opening the file with an OPEN EXTEND statement.

When the OPEN EXTEND is executed, the file is positioned following the end-of-partition written by the last CLOSE operation. This end-of-partition immediately follows the last record that was written before the file was closed. WRITE operations after the OPEN EXTEND continue from this position.

OPEN EXTEND cannot be specified for tape files, or for mass storage files that were closed with the NO REWIND option.

READING SEQUENTIAL FILES

The same Environment Division and Data Division clauses are used for reading sequential files as for their creation. Procedure Division statements that can be employed are shown in figure 3-5.

OPEN INPUT	Required
READ NEXT AT END	Required
CLOSE	Required
USE BEFORE/AFTER LABEL	Optional

Figure 3-5. Procedure Division for Reading Sequential Files

When a file is to be read only, reading starts at the current position, accesses the next record in sequence, and continues accessing records until an end-of-partition (end-of-section for the file INPUT) is encountered. AT END must be specified for every READ statement to indicate action to be taken when this boundary is encountered. NEXT can optionally be specified for documentation purposes:

```
READ NEXT file-name
AT END GO TO paragraph-name.
```

SAMPLE PROGRAM 1: USING SEQUENTIAL FILES

Sample program 1, shown in figure 3-6, is a simple illustration of a program using sequential files. The program reads 10 records from the card file CARD-A (implementor-name INPUT) and writes them to the tape file FILE-A (implementor-name TAPE01). FILE-A is then read, and the record numbers of the first and tenth records are displayed to verify that all 10 records were read. The input data for sample program 1 is shown in figure 3-7 and the output in figure 3-8.

Several lines in the program illustrate concepts related to sequential files (line numbers refer to the compiler-assigned numbers to the left of the source lines).

Line 13 — Line 14

The SELECT clauses link the file-names used by the COBOL program (CARD-A and FILE-A) with the logical file names (implementor-names) used by CYBER Record Manager and the operating system (INPUT and TAPE01).

Line 21

Labels for the tape file FILE-A are defined as being in ANSI standard format.

Line 22

The BLOCK CONTAINS clause defines the block type as C (fixed length character count). In the control statements for this job, the REQUEST control statement specifying device type residence for TAPE01 (FILE-A) defines it to be a tape file in SI format. Since the RECORDING MODE clause is omitted, the recording mode is assumed to be decimal. Block size for a decimal tape file in SI format is automatically set by CYBER Record Manager to 1280 characters; the specification of 640 characters in the BLOCK CONTAINS clause is ignored.

Line 23

The RECORD CONTAINS clause, in conjunction with the Record Description entry, defines the record type as F (fixed length) with a record length of 256 characters. The record length is calculated from the Record Description entry; the fact that the correct length is specified in the RECORD CONTAINS clause is irrelevant.

Line 30

The file CARD-A, which is a mass storage file, is unlabeled. The block and record specifications for this file are automatically determined by the fact that the implementor-name is INPUT; the record type is defined as Z (zero byte records) and the block type is C (fixed length character count), with a block size of 640 characters for mass storage.

Line 49

Since the implementor-name of CARD-A is INPUT, the AT END clause is executed when the 7/8/9 card is encountered.

Line 45

When the OPEN OUTPUT statement is executed, ANSI standard beginning labels are written to FILE-A, and the file is opened with processing restricted to writing.

Line 56

When the CLOSE statement is executed, an end-of-information followed by ANSI standard ending labels is written to FILE-A.

```
00001           IDENTIFICATION DIVISION.
00002
00003           PROGRAM-ID.  SEQUENTIAL-FILE-I-O.
00004
00005           ENVIRONMENT DIVISION.
00006
00007           CONFIGURATION SECTION.
00008           SOURCE-COMPUTER.  CYBER.
00009           OBJECT-COMPUTER.  CYBER.
00010
00011           INPUT-OUTPUT SECTION.
00012           FILE-CONTROL.
00013               SELECT CARD-A  ASSIGN TO INPUT.
00014               SELECT FILE-A  ASSIGN TO TAPE01.
00015
00016           DATA DIVISION.
00017
00018           FILE SECTION.
00019
00020           FD  FILE-A
00021               LABEL RECORDS ARE STANDARD
00022               BLOCK CONTAINS 640 CHARACTERS
00023               RECORD CONTAINS 256 CHARACTERS
00024               DATA RECORD IS REC-A.
00025           01  REC-A.
00026               03  REC-ID       PICTURE X(3).
00027               03  REC-NO       PICTURE 9(3).
00028               03  FILLER       PICTURE X(250).
00029
00030           FD  CARD-A
00031               LABEL RECORDS ARE OMITTED
00032               DATA RECORD IS DATA-A.
00033           01  DATA-A PICTURE X(80).
00034
00035           WORKING-STORAGE SECTION.
00036           77  COUNT PICTURE 9(3).
00037           01  CARD-IMAGE.
00038               03  CARD-ID      PICTURE X(3).
00039               03  CARD-NO      PICTURE 9(3).
00040               03  FILLER       PICTURE X(74).
00041
00042           PROCEDURE DIVISION.
00043
00044           START-WRITE.
00045               OPEN INPUT CARD-A OUTPUT FILE-A.
00046               MOVE 0 TO COUNT.
00047
00048           WRITE-REC.
00049               READ CARD-A INTO CARD-IMAGE AT END GO TO END-WRITE.
00050               WRITE REC-A FROM CARD-IMAGE.
00051               ADD 1 TO COUNT.
00052               IF COUNT IS LS 10
00053                   GO TO WRITE-REC.
00054
00055           END-WRITE.
00056               CLOSE FILE-A.
00057
00058           START-READ.
00059               OPEN INPUT FILE-A.
00060
00061           READ-REC.
00062               READ FILE-A AT END
00063                   GO TO END-READ.
00064               IF REC-NO = 1 DISPLAY ≠FIRST RECORD = ≠REC-NO.
00065               IF REC-NO = 10 DISPLAY ≠LAST  RECORD = ≠REC-NO.
00066               GO TO READ-REC.
00067
00068           END-READ.
00069               CLOSE FILE-A.
00070               STOP RUN.
```

Figure 3-6. Sample Program 1: Processing Sequential Files

```
AAA001
AAA002
AAA003
BBB004
BBB005
CCC006
CCC007
CCC008
DDD009
DDD010
```

Figure 3-7. Input Data for Sample Program 1

```
FIRST RECORD = 001
LAST  RECORD = 010
```

Figure 3-8. Output from Sample Program 1

Line 59

When the OPEN INPUT statement is executed, the beginning labels on FILE-A are checked to ensure that they conform to ANSI standard format. If they do, FILE-A is opened with processing restricted to reading. If they do not, a fatal diagnostic is issued.

Line 69

When the CLOSE statement is executed, the ending labels on FILE-A are checked to ensure that they conform to ANSI standard format. If they do not, a fatal diagnostic is issued.

SAMPLE PROGRAM 2: C TYPE BLOCKS, Z TYPE RECORDS

Sample program 2, shown in figure 3-9, illustrates the use of files with C type blocks and Z type records, referred to here as CZ files. These files are important because their structure corresponds to that of the punch card files INPUT and PUNCH and the print line file OUTPUT. Thus, a file created with C type blocks and Z type records can be printed or punched in a later job step. (It must be remembered that the first character of every line in a print file is used for carriage control and is not printed.) The files whose implementor-names are INPUT, OUTPUT, and PUNCH are defined as CZ files by COBOL.

The purpose of sample program 2 is to compare records in two files (CARD-FILE and CEE-ZEE-FILE), listing the equal records on FIRST-PRINT-FILE and the unequal records on SECOND-PRINT-FILE. The operator's console is used to display the total number of records in each category.

The input data for sample program 2 is shown in figure 3-10, and the output is shown in figure 3-11.

Several lines of the program show alternative methods of processing CZ files.

Line 17 — Line 20

The special implementor-names INPUT and OUTPUT are automatically interpreted by COBOL to mean that these files have Z type records and C type blocks. For the file OUTPUT, which is printed by default at the end of the job, the first character of each line is used by the system for carriage control; it is the user's responsibility to ensure that this character position contains the desired carriage control character, and not a character which is part of the information to be printed.

The implementor-name suffix -P after PRINT2 is another way to specify Z type records and C type blocks. In addition, -P instructs COBOL to add an additional character in front of each record to contain the carriage control character. When -P is specified, every character position in the data record is to be printed. The user cannot access the carriage control character directly, but must specify carriage control through some form of the ADVANCING option of the WRITE statement. The implementor-name suffix -FZ after DISC01 specifies that the record type for DISC01 is to be Z. No assumptions are made by COBOL about carriage control for this file.

Line 36

PRINT-CONTROL, the first elementary item in the Record Description for FIRST-PRINT-FILE, specifies a character position that is to be set to indicate carriage control.

Line 47

The BLOCK CONTAINS specification for CEE-ZEE-FILE, in conjunction with the file's mass storage residence (as determined by COBOL at execution time), defines the block type as C; thus the file is a CZ file (since Z type records are specified by the ASSIGN clause).

Lines 78, 80, 85, 94, 102, 103

All WRITE statements using the ADVANCING option imply carriage control. The difference between the implementor-names PRINT2-P (whose data record is REPORT-LINE) and OUTPUT (whose data record is PRINT-LINE) is that for PRINT2-P, the carriage control character is not within the record, while for OUTPUT, the first character of the record is set by COBOL to the appropriate character at the time the WRITE statement is executed, regardless of its previous contents.

Lines 104, 107, 108, 109

DISPLAY . . . UPON TUBE results in the specified information being displayed at the operator's console, since TUBE was equated to CONSOLE in the SPECIAL-NAMES paragraph.

Lines 124, 125

The mnemonic NEW-PAGE, defined in the SPECIAL-NAMES paragraph, is used in these WRITE statements to specify page ejection before writing.

```
00001            IDENTIFICATION DIVISION.
00002            PROGRAM-ID.  CZ-FILES.
00003
00004            ENVIRONMENT DIVISION.
00005
00006            CONFIGURATION SECTION.
00007            SOURCE-COMPUTER.  CYBER.
00008            OBJECT-COMPUTER.  CYBER.
00009
00010            SPECIAL-NAMES.
00011                CONSOLE IS TUBE
00012                ≠1≠ IS NEW-PAGE
00013                  TERMINA IS REMOTE.
00014
00015            INPUT-OUTPUT SECTION.
00016            FILE-CONTROL.
00017                SELECT CARD-FILE ASSIGN TO INPUT.
00018                SELECT FIRST-PRINT-FILE ASSIGN TO OUTPUT.
00019                SELECT  SECOND-PRINT-FILE ASSIGN TO PRINT2-P.
00020                SELECT  CEE-ZEE-FILE ASSIGN TO DISC01-FZ.
00021
00022            DATA DIVISION.
00023
00024            FILE SECTION.
00025
00026            FD CARD-FILE
00027                LABEL RECORDS OMITTED
00028                DATA RECORD CARD-IMAGE.
00029            01  CARD-IMAGE.
00030                02 CARD-COLUMN PIC X OCCURS 80 TIMES.
00031
00032            FD FIRST-PRINT-FILE
00033                LABEL RECORDS OMITTED
00034                DATA RECORD PRINT-LINE.
00035            01 PRINT-LINE.
00036                02 PRINT-CONTROL  PIC X.
00037                02 PRINT-BODY   PIC X(135).
00038
00039            FD SECOND-PRINT-FILE
00040                LABEL RECORDS OMITTED
00041                DATA RECORD REPORT-LINE.
00042            01 REPORT-LINE.
00043                02 REPORT-BODY PIC X(135).
00044
00045            FD CEE-ZEE-FILE
00046                LABEL RECORDS OMITTED
00047                BLOCK CONTAINS 640 CHARACTERS
00048                DATA RECORD Z-RECORD.
00049            01 Z-RECORD.
00050                02 Z-CHARACTER PIC X OCCURS 80 TIMES.
00051
00052            WORKING-STORAGE SECTION.
00053                77 PAIR-COUNT PIC 9(5) VALUE ZERO.
00054                77 MISS-COUNT PIC 9(5) VALUE ZERO.
00055                77 HIT-COUNT  PIC 9(5) VALUE ZERO.
00056                77 SUB        PIC 99 COMP-1.
00057
00058            PROCEDURE DIVISION.
00059
00060            OPEN-ALL-FILES.
00061                OPEN INPUT CARD-FILE CEE-ZEE-FILE.
00062                OPEN OUTPUT FIRST-PRINT-FILE SECOND-PRINT-FILE.
00063                PERFORM PAGE-EJECTION.
00064
```

Figure 3-9. Sample Program 2: C Blocks, Z Records (Sheet 1 of 2)

```
00065           READ-TWO-RECORDS.
00066               READ CARD-FILE AT END GO TO WRAP-UP-1.
00067               READ CEE-ZEE-FILE AT END GO TO PREMATURE-EOF.
00068               ADD 1 TO PAIR-COUNT.
00069               IF CARD-IMAGE NOT EQUAL Z-RECORD
00070                   ADD 1 TO MISS-COUNT
00071                   PERFORM UNEQUAL-REPORT
00072               ELSE    ADD 1 TO HIT-COUNT
00073                       PERFORM EQUAL-REPORT.
00074               GO TO READ-TWO-RECORDS.
00075
00076           UNEQUAL-REPORT.
00077               MOVE CARD-IMAGE TO REPORT-BODY.
00078               WRITE REPORT-LINE AFTER ADVANCING 3 LINES.
00079               MOVE Z-RECORD TO REPORT-BODY.
00080               WRITE REPORT-LINE AFTER ADVANCING 1 LINES.
00081               PERFORM CHARACTER-COMPARE
00082                   VARYING SUB FROM 1 BY 1
00083                   UNTIL SUB > 80.
00084               MOVE CARD-IMAGE TO REPORT-LINE.
00085               WRITE REPORT-LINE AFTER ADVANCING 1 LINES.
00086
00087           CHARACTER-COMPARE.
00088               IF CARD-COLUMN (SUB) = Z-CHARACTER (SUB)
00089                   MOVE SPACE TO CARD-COLUMN (SUB)
00090               ELSE MOVE ≠↑≠ TO  CARD-COLUMN (SUB).
00091
00092           EQUAL-REPORT.
00093               MOVE CARD-IMAGE TO PRINT-BODY.
00094               WRITE PRINT-LINE AFTER ADVANCING 1 LINES.
00095
00096           WRAP-UP-1.
00097               READ CEE-ZEE-FILE AT END GO TO FINAL-WRAP-UP.
00098
00099           PREMATURE-EOF.
00100               MOVE  ≠FILES ARE OF UNEQUAL LENGTH * * * * * * * * * * ≠
00101                   TO PRINT-BODY REPORT-BODY.
00102               WRITE PRINT-LINE AFTER ADVANCING 5 LINES.
00103               WRITE REPORT-LINE AFTER ADVANCING 5 LINES.
00104               DISPLAY ≠ FILES UNEQUAL IN LENGTH ≠ UPON TUBE.
00105
00106           FINAL-WRAP-UP.
00107               DISPLAY PAIR-COUNT ≠ SETS COMPARED≠ UPON TUBE.
00108               DISPLAY HIT-COUNT  ≠ EQUAL PAIRS≠   UPON TUBE.
00109               DISPLAY MISS-COUNT ≠ UNEQUAL PAIRS≠ UPON TUBE.
00110               CLOSE
00111                   CARD-FILE
00112                   CEE-ZEE-FILE
00113                   FIRST-PRINT-FILE
00114                   SECOND-PRINT-FILE.
00115               STOP RUN.
00116
00117           PAGE-EJECTION.
00118               MOVE
00119               ≠ L I S T I N G  O F  E Q U A L  I M A G E S ≠
00120                   TO PRINT-BODY.
00121               MOVE
00122               ≠ L I S T I N G  O F  U N E Q U A L  I M A G E S ≠
00123                   TO REPORT-BODY.
00124               WRITE PRINT-LINE AFTER ADVANCING  NEW-PAGE.
00125               WRITE REPORT-LINE AFTER ADVANCING NEW-PAGE.
```

Figure 3-9. Sample Program 2: C Blocks, Z Records (Sheet 2 of 2)

```
4679VQ4CT       995233324100000745662187000000065   BBQ/////,TC
46799693488 99355279 52766 3146680000002564319973   7347107 (415)
VQ4CT 852334611379520001452336B   7522913345 99999///,DF X  (415)
VQ4CT 4087347440  95220 622302214  99///, NBG NBG ,,, 01665
VQ4CT 4087347440  95220 625663-14  99///, NBG NBG ,,, 01665
W6BGT 0113 6652 42035 4421 88520 66523 00014 00233 66 2241 /// ,SD4
W6STC 013  52033 652 // ,, BGD 0123 K6632 4123 95236 4000 9999 NBG NBG
7/8/9 Card
4679VQ4CT       995233324100000745662187000000065   BBQ/////,TC
46799693488 99355279 52766 3146680000002564319973   7347107 (415)
VQ4CT 852334611379520001452336B   7522013345 99999///,DF X  (415)
VQ4CT 4087347440  95220 622302214  99///, NBG NBG ,,, 01665
VQ4CT 4087347440  95220 625663-14  99///, NBG NBG ,,, 01665
W6BGT 0113 6652 42035 4421 88520 66523 00014 00233 66 2241 /// ,SD4
W6STC 013  52033 652 // ,, BGD 0123 K6632 4123 95236 4000 9999 NBG NBG
```

Figure 3-10. Input Data for Sample Program 2

```
 L I S T I N G  O F  E Q U A L  I M A G E S
4679VQ4CT       995233324100000745662187000000065   BBQ/////,TC
46799693488 99355279 52766 3146680000002564319973   7347107 (415)
VQ4CT 4087347440  95220 622302214  99///, NBG NBG ,,, 01665
VQ4CT 4087347440  95220 625663-14  99///, NBG NBG ,,, 01665
W6STC 013  52033 652 // ,, BGD 0123 K6632 4123 95236 4000 9999 NBG NBG


 L I S T I N G  O F  U N E Q U A L  I M A G E S

VQ4CT 852334611379520001452336B   7522013345 99999///,DF X  (415)
VQ4CT 852334611379520001452336B   7522913345 99999///,DF X  (415)
                                        ↑

W6BGT 0113 6652 42035 4421 88520 66523 00014 00233 66 2241 /// ,SD4
W6BGT 0113 6652 42035 4421I 88520 66523 00014 00233 66 2241 /// ,SD4
                          ↑
```

Figure 3-11. Output from Sample Program 2

Relative files are mass storage files with fixed-length records in which a record key gives the ordinal of a record and therefore its location in the file. A record ordinal is a positive integer which identifies the relative physical location of a record in the file. The first record in the file is identified by ordinal 1, the second by ordinal 2, and so forth. Relative files may be accessed sequentially, without reference to the key, or they may be accessed randomly by key.

Relative files are implemented through CYBER Record Manager word addressable file organization. They are unblocked; if the BLOCK CONTAINS clause occurs in the File Description for a relative file, it is ignored.

Only F type records (fixed length records) are specified by COBOL for relative files; the record length for all records in a file is the longest length specified in any Record Description entry for the file. The RECORD CONTAINS specification is not used.

For any program using a relative file, access mode is either sequential or random. The file must be open for input, output, or input/output. The Procedure Division input/output statements permitted under the various access modes and open conditions are shown in table 4-1 and described below.

RANDOM ACCESS

Random creation, reading, or updating of relative files requires the inclusion of the clause ACCESS MODE IS RANDOM in the Environment Division FILE-CONTROL paragraph. The ACTUAL KEY clause in the same paragraph is also required to specify the data item used to hold key values for reading and writing records. The data item specified by the ACTUAL KEY clause may be part of the record, but does not have to be. On a write, only the data record itself is written to the file; no index is maintained. A record key in a relative file is simply the ordinal of that record, as defined above. Key values must be integers greater than zero. The first record in the file has ordinal 1 and key 1, and so forth. The exact location of a record is calculated by COBOL from the key.

Under random access, when a record is written to a file, the value contained in the data item referenced in the ACTUAL KEY clause is used as the record key. If the key is greater than the key of any existing record, sufficient mass storage space is allocated for the file to accommodate the record being written and all intervening records. Thus a relative file can contain unused mass storage space. For this reason, relative files are inefficient when key values are widely dispersed; they are more efficient when keys tend to be clustered and the total range of keys is not great.

The FILE-LIMITS clause (discussed below) can be used to place an upper limit on the number of records in a relative file.

SEQUENTIAL ACCESS

Any relative file can be accessed randomly or sequentially regardless of the mode in which it was created. Sequential access is indicated by the clause ACCESS MODE IS SEQUENTIAL in the Environment Division FILE-CONTROL paragraph.

When relative files are written in sequential mode, the ACTUAL KEY clause is ignored, and the contents of the key item it references are not used to determine the location of records. The ordinal values of records, however, are used to determine if processing has exceeded the ranges specified in the FILE-LIMITS clause, and the INVALID KEY clause (for WRITE) or AT END clause (for READ) is activated if so.

File reading and writing proceed just as for files with sequential organization. When a file is written, records are added only at the end of the file. When the file is read, records are returned to the user program in ordinal order according to successive positions in the file. The key item is updated to reflect the ordinal of the record read.

TABLE 4-1. ACCESS MODE AND OPEN STATUS COMBINATIONS, RELATIVE FILES

Statements	Random Access			Sequential Access		
	Open Input	Open Output	Open I-O	Open Input	Open Output	Open I-O
READ NEXT AT END	Y	N	Y	Y	N	Y
READ INVALID KEY	Y	N	Y	N	N	N
WRITE	N	N	N	N	Y	Y
WRITE INVALID KEY	N	Y	Y	N	Y	Y
REWRITE INVALID KEY	N	N	Y	N	N	Y
DELETE INVALID KEY	N	N	Y	N	N	Y
Create New File	N	Y	N	N	Y	N
Y = Allowed	N = Not allowed					

As table 4-1 shows, there is some overlap of function between random and sequential access modes for relative files. When the access mode is random, sequential reads are possible through READ NEXT AT END. These reads proceed the same as when access mode is sequential; keys are updated but not used. When the access mode is sequential, random writes, rewrites, and deletes are allowed (using the INVALID KEY clause in each case). Random reads are not possible when the access mode is sequential. In every case, the open status must be correct for these operations to proceed.

CREATING RELATIVE FILES

COBOL clauses and statements particularly applicable to creation of relative files are described below. Clauses that would normally be included in any COBOL program, as well as those that are discussed comprehensively elsewhere in this user guide, are not included here.

ENVIRONMENT DIVISION

The Environment Division clauses used to establish a relative file are shown in figure 4-1.

Specifying Relative File Organization

Relative file organization is indicated to COBOL when any of the following conditions is true:

1. The clause ORGANIZATION IS RELATIVE is included in the FILE-CONTROL paragraph.

2. The suffix -A is appended to the implementor-name in the ASSIGN clause:

   ```
   SELECT BLARGH ASSIGN TO QC-A
   ```

3. The FILE-LIMITS or ACTUAL KEY clause is included, the ORGANIZATION IS clause does not specify any other file organization, and the Z parameter is omitted from the COBOL control statement.

4. The suffix -X is appended to the implementor-name in the ASSIGN clause:

   ```
   SELECT REL ASSIGN TO RELAT-X
   ```

The -X suffix additionally instructs COBOL to write each record in the file beginning at a 64-word PRU boundary. This option provides faster processing in random access mode, but can result in unused mass storage space, depending on record size. Once a file is created with the -X option, it must always be used with the option, and always with random access. A file created without the -X option cannot be used with the option.

RESERVE ALTERNATE AREAS

RESERVE ALTERNATE AREAS can be included to increase buffer size and speed processing for relative files to be read or written sequentially:

```
RESERVE integer ALTERNATE AREAS
```

Additional buffer areas should not be specified for files to be accessed randomly, however, because of the increase in time required in locating and accessing a record. The NO option of the RESERVE ALTERNATE AREAS clause can be used to avoid an increase in buffer size, or the clause can be omitted.

FILE-LIMITS

The FILE-LIMITS clause can be used either in a program that creates a file or in a program that updates an existing file. Processing of the file is restricted to records with keys in the intervals specified by pairs of operands associated with the key word THROUGH:

```
FILE-LIMITS ARE 1 THROUGH 10
21 THROUGH 30  41 THROUGH 50
```

When a file is being created, only records with keys in the specified intervals are written to the file; however, space is allocated for intervening records if necessary to establish the position of a subsequent record.

If the FILE-LIMITS clause shown above is specified when a file is created sequentially, the first 10 writes create records 1 through 10. On the next write, space is allocated for records 11 through 20, and record 21 is written. After record 30 is written, the next write allocates space for records 31 through 40, and writes record 41.

Whether a file is being created or updated, an attempt to randomly access a record whose key does not fall within an interval specified in the FILE-LIMITS clause results in activation of the INVALID KEY clause.

If desired, the FILE-LIMITS clause can be used to establish a maximum value for record keys, with no other restriction:

```
FILE-LIMIT IS 1 THRU 1000
```

In this example, only keys greater than 1000 cause execution of the INVALID KEY clause. On a sequential write (a write without the INVALID KEY option), records exceeding the file limits are not written to the file, but no indication is made to the user.

SELECT . . . ASSIGN	Required
ORGANIZATION IS RELATIVE	Optional
RESERVE ALTERNATE AREAS	Optional
FILE-LIMITS	Optional
ACCESS MODE	Required
PROCESSING IS SEQUENTIAL	Documentary only
ACTUAL KEY	Required for random access

Figure 4-1. Environment Division Clauses for Creating Relative Files

ACCESS MODE

The ACCESS MODE clause is required for relative files. When a file is created, the clause should specify the correct access mode. If creation is to be random, the clause should specify:

```
ACCESS MODE IS RANDOM
```

If creation is to be sequential, the clause should specify:

```
ACCESS MODE IS SEQUENTIAL
```

ACTUAL KEY

A record accessed randomly is located through the key specifying its relative position in the file; thus, the ACTUAL KEY clause must be included for relative files for which ACCESS MODE IS RANDOM is specified:

```
ACTUAL KEY IS data-name
```

The data-name in the clause should be in COMPUTATIONAL-1 (integer) format; it must be an integer greater than zero, and within the values specified by FILE-LIMITS, and specifies the record ordinal. Specifying a key value of 72 on the first WRITE, for instance, creates a file with 71 empty entries followed by record 72, unless otherwise restricted by the FILE-LIMITS clause.

DATA DIVISION

Only one Data Division clause is specifically required for relative files:

```
LABEL RECORD IS
```

Labels can optionally be used for relative file creation if ACCESS MODE IS SEQUENTIAL is specified; any form of the LABEL RECORD clause is permitted (see section 3).

If labels are not used, LABEL RECORD IS OMITTED is required.

PROCEDURE DIVISION

Once the file specifications are established with Environment and Data Division clauses, the statements shown in figure 4-2 are used to create a relative file.

USE AFTER ERROR	Optional
USE BEFORE/AFTER LABEL	Optional
OPEN OUTPUT	Required
WRITE INVALID KEY WRITE	One required
CLOSE	Required

The programmer can specify USE AFTER ERROR PROCEDURE to indicate one or more routines to be executed following an error in file creation. The USE BEFORE/AFTER LABEL PROCEDURE statement allows the user to provide a routine to check or prepare labels in conjunction with the procedures provided by CYBER Record Manager. USE procedures are discussed in section 2.

OPEN OUTPUT opens the file for creation with the WRITE statement:

```
OPEN OUTPUT file-name
```

No other version of the OPEN statement can be used for relative file creation.

Relative files are created either randomly or sequentially, depending on the form of the WRITE statement used. If WRITE is specified with the INVALID KEY option, creation is random; the ACCESS MODE clause can specify either RANDOM or SEQUENTIAL. If the INVALID KEY option is not present, creation is sequential; in this case, ACCESS MODE IS RANDOM must not be specified.

USE AFTER ERROR	Optional
USE BEFORE/AFTER LABEL	Optional
OPEN OUTPUT	Required
WRITE INVALID KEY WRITE	One required
CLOSE	Required

Figure 4-2. Procedure Division Statements for Creating Relative Files

For random creation, the user must specify an actual key through the clause ACTUAL KEY IS data-item. The location a record is written to is determined by the contents of the key item when the WRITE statement is executed. The user program is responsible for setting the data-item to the desired record ordinal. If the contents of the data-item are invalid or not within the ranges specified by the FILE-LIMITS clause, the INVALID KEY clause is executed. The INVALID KEY clause is also executed if an attempt is made to write a record using the key of an existing record. (If the P option of the COBOL control statement, requiring ANSI compatibility, is in effect, the record is rewritten and INVALID KEY is not executed.)

For sequential creation, any key specified by the ACTUAL KEY clause is not used. Each record is written in the next physical location after the last record written; the user has no control over the location. An internal key is generated and updated after each write, but the user cannot access this key. If the FILE-LIMITS clause is used, writing is restricted to the range it specifies. If an attempt is made to write a record past the last record allowed by the FILE-LIMITS clause, the record is not written, but no indication is made to the user. Therefore, to ensure that all desired records are actually written to the file, the user program must keep track of the number of records written, to see that file limits are not exceeded.

PROCESSING EXISTING RELATIVE FILES

Records in existing relative files can be read, written, rewritten, or deleted. Access mode can be random or sequential, and the file can be open for INPUT, OUTPUT, I-O, or EXTEND. The various combinations of these options that are permitted are shown in table 4-1.

The same Environment Division clauses are used for updating as for creating files. FILE-LIMITS can be specified to restrict file processing to particular portions of an existing file.

The LABEL RECORD clause in the Data Division is required. If the file is already labeled, and is opened for I-O, labels are checked. Other label processing takes place just as for sequential files (see section 3).

The Procedure Division input/output statements used for processing existing relative files are shown in figure 4-3.

USE AFTER ERROR	Optional
USE BEFORE/AFTER LABEL	Optional
OPEN	Required
READ NEXT AT END	Optional
READ INVALID KEY	Optional
WRITE	Optional
WRITE INVALID KEY	Optional
REWRITE INVALID KEY	Optional
DELETE INVALID KEY	Optional
CLOSE	Required

Figure 4-3. Procedure Division Statements for Existing Relative Files

The USE statement allows the COBOL programmer to define special processing for errors and labels. USE statements are discussed in section 2.

The form of OPEN statement used depends on the type of processing to take place (see table 4-1).

OPEN EXTEND allows records to be added after the last record on a sequential update operation; it can be used only with files for which ACCESS MODE IS SEQUENTIAL is specified:

```
OPEN EXTEND file-name
```

Relative files are read sequentially by using the AT END option of the READ statement:

```
READ file-name AT END
imperative-statement
```

This format is valid whether ACCESS MODE IS SEQUENTIAL or ACCESS MODE IS RANDOM has been specified. Sequential reading results in the transfer to the input record area of records in the order of their physical occurrence on the file. The key item defined by ACTUAL KEY IS is updated after each read to the ordinal of the record just read, whether access is random or sequential. If gaps exist where no records have been written (as a result of random writing or FILE-LIMITS restrictions), they are skipped; only valid data is returned to the user.

Random access reading of relative files takes place only when ACCESS MODE IS RANDOM is specified. The format used is:

```
READ file-name
INVALID KEY imperative-statement
```

The user specifies the record to be read by setting the key item to its key. The INVALID KEY clause is executed if the contents of the key are invalid or not within the ranges specified by the FILE-LIMITS clause.

When ACCESS MODE IS RANDOM is specified, sequential and random reads may be intermixed. Each random read reads a record according to the contents of the key item; each sequential read reads the next record after the record most recently referenced.

Writing of records on an update run is identical to writing records on a creation run (discussed above), except that if the file is open for I-O, and a sequential WRITE follows a READ, the record just read is rewritten.

Existing records in a file opened for I-O can be replaced by a REWRITE statement or deleted with a DELETE statement. Both REWRITE and DELETE require an INVALID KEY clause; the record to be rewritten or deleted is located by its key. LAST can be specified with REWRITE or DELETE to indicate that the contents of the record most recently accessed are to be replaced; the contents of the actual key must not have been changed since the last access. For both REWRITE and DELETE, the INVALID KEY clause is executed if the contents of the key item are invalid or not within the limits specified by the FILE-LIMITS clause.

SAMPLE PROGRAM 3: PROCESSING RELATIVE FILES

Sample program 3, shown in figure 4-4, creates a relative file (FILE1) containing 10 records, then closes the file and reopens it to read the records in order. The first and last records are displayed. There is no input for the program; the output is shown in figure 4-5.

Line 13 — Line 14

Relative organization is defined for FILE1 in two ways in this program: through the suffix -A to the implementor-name, and through the clause ORGANIZATION IS RELATIVE. Either specification could have been omitted, but one of the two is necessary.

Line 15

ACCESS IS RANDOM indicates that records in FILE1 are to be read and written only by key in this program. Future programs can process FILE1 sequentially. The PROCESSING MODE clause is documentary only.

Line 16 — Line 17

The FILE-LIMITS clause is optional; in this format it limits

```
00001            IDENTIFICATION DIVISION.
00002
00003            PROGRAM-ID.  RELATIVE-I-O.
00004
00005            ENVIRONMENT DIVISION.
00006
00007            CONFIGURATION SECTION.
00008            SOURCE-COMPUTER. CYBER.
00009            OBJECT-COMPUTER. CYBER.
00010
00011            INPUT-OUTPUT SECTION.
00012            FILE-CONTROL.
00013                SELECT FILE1 ASSIGN TO LFN1-A
00014                ORGANIZATION IS RELATIVE
00015                 ACCESS IS RANDOM  PROCESSING IS SEQUENTIAL
00016                      FILE-LIMIT IS 4094
00017                        ACTUAL KEY IS ACT-KEY.
00018
00019            DATA DIVISION.
00020
00021            FILE SECTION.
00022
00023            FD  FILE1 RECORDING MODE IS DECIMAL
00024                    RECORD CONTAINS 2000 CHARACTERS
00025                    LABEL RECORDS ARE OMITTED.
00026            01 REC.
00027                03  FILLER       PICTURE X.
00028                03  REC-NO       PICTURE 9(17).
00029                03  FILLER       PICTURE X(1982).
00030
00031            WORKING-STORAGE SECTION.
00032            77  SO-MANY PICTURE 9(14) VALUE IS 10.
00033            77  ACT-KEY PICTURE 9(4) VALUE IS 0.
00034
00035            PROCEDURE DIVISION.
00036
00037            START-WRITE.
00038                OPEN OUTPUT FILE1.
00039                MOVE SPACES TO REC.
00040                DISPLAY ≠DISPLAY FIRST AND LAST RECORD ONLY.≠.
00041                PERFORM WRITES SO-MANY TIMES.
00042                CLOSE FILE1.
00043                DISPLAY ≠FINISHED CREATING FILE≠.
00044                GO TO START-READ.
00045.
00046            WRITES.
00047                ADD 1 TO ACT-KEY.  MOVE ACT-KEY TO REC-NO.
00048                IF REC-NO EQ 1 DISPLAY REC.
00049                IF REC-NO EQ SO-MANY DISPLAY REC.
00050                WRITE REC INVALID KEY
00051                    DISPLAY ≠**INVALID KEY**≠
00052                    DISPLAY ≠KEY = ≠ ACT-KEY
00053                        STOP RUN.
00054
00055            START-READ.
00056                MOVE 0 TO ACT-KEY.
00057                MOVE SPACES TO REC.
00058                OPEN INPUT FILE1.
00059                DISPLAY ≠START READING FILE≠.
00060                DISPLAY ≠DISPLAY FIRST AND LAST RECORD ONLY.≠.
00061
00062            READ-IT.
00063                ADD 1 TO ACT-KEY.
00064                READ FILE1 INVALID KEY GO TO DONE-RUN.
00065                IF ACT-KEY NOT = REC-NO
00066                  DISPLAY ≠**ERROR**≠
00067                  DISPLAY ≠KEY = ≠ ACT-KEY
00068                   DISPLAY ≠DISPLAY RECORD ON NEXT LINE≠
```

Figure 4-4. Sample Program 3: Processing Relative Files (Sheet 1 of 2)

```
00069                    DISPLAY REC
00070                   STOP RUN.
00071               IF REC-NO EQ 1 DISPLAY REC.
00072               IF REC-NO EQ SO-MANY DISPLAY REC.
00073               GO TO READ-IT.
00074
00075           DONE-RUN.
00076               CLOSE FILE1.
00077               DISPLAY ≠TOTAL RECORDS READ = ≠ ACT-KEY.
00078               STOP RUN.
```

Figure 4-4. Sample Program 3: Processing Relative Files (Sheet 2 of 2)

```
DISPLAY FIRST AND LAST RECORD ONLY.
 00000000000000001



 00000000000000010



FINISHED CREATING FILE
START READING FILE
DISPLAY FIRST AND LAST RECORD ONLY.
 00000000000000001



 00000000000000010



TOTAL RECORDS READ = 0011
```

Figure 4-5. Output from Program 3

the number of records the file can contain to 4094.

The ACTUAL KEY clause is required to define the key item for random reads and writes.

Line 24 — Line 25

The RECORD CONTAINS clause specifies the same record length as the Record Description entry, 2000 characters. COBOL selects F type (fixed length) records, and calculates the record length from the Record Description entry; the RECORD CONTAINS clause is ignored. The LABEL RECORD clause specifies an unlabeled file.

Line 37 — Line 53

The first part of the Procedure Division opens FILE1 for output (line 38) and then creates the file by writing 10 records to it. The contents of each record are a 17-character key, and 1983 characters of filler. The keys are generated by successively incrementing ACT-KEY (line 47), the value of the key is moved into the record (line 47), and the record is written by a random write (line 50). The INVALID KEY clause of the WRITE statement is activated, the key displayed, and processing halted only if the file limit of 4094 records is exceeded. Because all key values from 1 to 10 are used, there are no gaps in the file after it is created. In this particular case, the file could just as well be created sequentially; no key item would then be used. In line 42 the file is closed.

Line 55 — Line 78

In line 58, FILE1 is reopened for input. The records are then read from the file by a random read (line 64) after the key item is incremented (line 63). Because the values the key item successively takes are the same as the values it took when the file was created, the records are read in the order they were written; this could also be accomplished through sequential processing.

Standard files are mass storage files whose records can be accessed only randomly; access is by a key that uniquely identifies the record. Sequential access is not possible. An index is used to correlate record keys with locations in the file; each record in the file requires one index entry.

Standard files are supported only for compatibility with previous versions of COBOL, and should not be used for new applications.

The index is constructed dynamically by COBOL execution time routines, and is maintained in central memory. When the file is closed, the index is written to the file.

Standard files are implemented through CYBER Record Manager word addressable file organization; the files are unblocked, and the BLOCK CONTAINS clause is ignored.

Records may vary in length, but since COBOL always specifies W type records for standard files (whether or not the -FW suffix is used in the ASSIGN clause), the RECORD CONTAINS clause is ignored. The OCCURS clause may be used in the Record Description entry.

Standard file organization is efficient insofar as mass storage space is not wasted, and only one access is needed per record. On the other hand, the necessity for one index entry per record means that the index can require an arbitrarily large amount of central memory space; therefore, the file organization is best for small files.

FILE STORAGE

Records occur on a standard file in the order they are written. Key values need not be sorted; the key value has no bearing on either the file size or the order in which records are stored. The sixth record written lies in the sixth record position in both the index and the file; the ninth written lies in the ninth position, and so forth. In the example shown in figure 5-1, records with actual key values of 1, 3, 7, and 4 were written to a standard file in that order and stored as shown. Their keys were stored in an index and associated with a word address as shown at the right.

CREATING STANDARD FILES

Creation of standard files involves the COBOL clauses described below. Clauses and statements such as CLOSE, which must be specified for all files, and those that do not require further explanation, are omitted from the descriptive paragraphs. Clauses and statements that apply only to existing files are described under Updating Standard Files and Reading Standard Files.

ENVIRONMENT DIVISION

The Environment Division clauses used in establishing a standard file are shown in figure 5-2.

Defining File Organization

Standard file organization is normally specified by including the clause ORGANIZATION IS STANDARD in the FILE-CONTROL paragraph of the Environment Division. However, if the Z option is specified on the COBOL control statement, requiring compatibility with COBOL 3, and the ACTUAL/-SYMBOLIC KEY clause is specified (as required for standard files), standard file organization is assumed by COBOL as the default in the absence of the ORGANIZATION clause. However, the Z option might cause undesired side effects (see the COBOL 4 Reference Manual).

RESERVE ALTERNATE AREAS

Use of the RESERVE ALTERNATE AREAS clause is not recommended because CYBER Record Manager handles buffer requirements efficiently in its absence, and random access requires more time to locate and access a record when larger buffers are specified. If the clause is used, however, each additional area specified is allotted either an additional 640 characters or the number of characters in the largest record, whichever is greater.

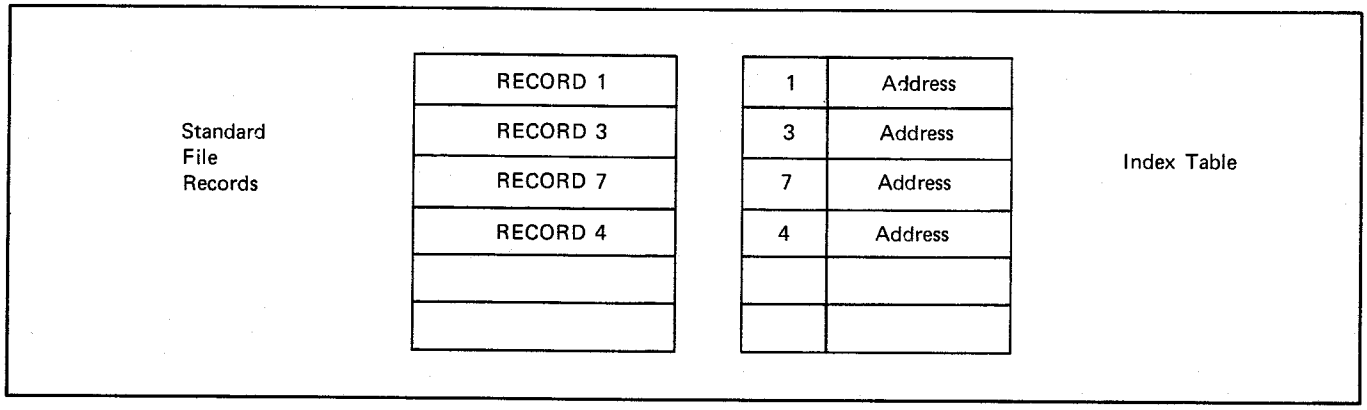

Figure 5-1. Standard File Structure

SELECT . . . ASSIGN	Required
ORGANIZATION IS STANDARD	Required in absence of Z control statement option
RESERVE ALTERNATE AREAS	Optional
FILE-LIMITS	Optional
ACCESS IS RANDOM	Required
ACTUAL/SYMBOLIC/RECORD KEY	Required

Figure 5-2. Environment Division for Creating Standard Files

FILE-LIMITS

A FILE-LIMITS clause should be specified for all standard files to indicate the total number of records. If it is omitted, no more than 4095 records can be written. FILE-LIMITS cannot be used to restrict processing to a portion of a standard file; only the following format can be used:

FILE-LIMITS IS integer

ACTUAL/SYMBOLIC/RECORD KEY

A key defined by the ACTUAL KEY, SYMBOLIC KEY, or RECORD KEY clause must be specified for record access:

ACTUAL KEY IS data-name

The key can be either within the record or not. An actual key is more efficient than a record or symbolic key; it must be an integer greater than zero and less than or equal to the FILE-LIMITS specification, and must consist of 14 or fewer digits. It should be described as COMPUTATIONAL-1 (integer); it must not be described as COMPUTATIONAL-2 (floating point). A symbolic key or record key is a string of one to seven characters. It should not be described as either COMPUTATIONAL-1 or COMPUTATIONAL-2.

DATA DIVISION

Only one Data Division clause is specifically required in declaring standard files:

LABEL RECORD IS OMITTED

Labels cannot be used with standard files, but the clause specifying that they are to be omitted is required.

PROCEDURE DIVISION

The input/output statements used to create a standard file are shown in figure 5-3.

USE AFTER ERROR PROCEDURE can be used with OUTPUT or I-O or with one or more file names to indicate routines to be executed following an error in file creation. USE procedures are described in section 2.

USE AFTER ERROR	Optional
OPEN	Required
WRITE INVALID KEY	Required
CLOSE	Required

Figure 5-3. Procedure Division Statements to Create Standard Files

Although OPEN OUTPUT is usually required for opening a file for creation, OPEN I-O can be used for standard files; this option is provided to allow compatibility with earlier versions of COBOL. The format is:

OPEN OUTPUT file-name

or:

OPEN I-O file-name

Only WRITE INVALID KEY can be used to write a new record:

WRITE record-name
INVALID KEY imperative-statement.

The INVALID KEY clause is executed if an attempt is made to write more records than allowed by the total specified in FILE-LIMITS, or if a bad key value is encountered.

UPDATING STANDARD FILES

Standard files can be updated with record additions, insertions, and overwrites. Records cannot be deleted. The Environment and Data Division clauses are the same as for file creation.

Procedure Division statements used for file update are shown in figure 5-4.

USE AFTER ERROR PROCEDURE can be specified with I-O.

INVALID KEY is required with the READ and WRITE statements. Any attempt to read a record with a key that does not exist or to access a record beyond the FILE-LIMITS specification activates the INVALID KEY clause. An existing record can be rewritten (overwritten) with a new record having the same key value through the WRITE statement; the new record must not be longer than the old record.

USE AFTER ERROR	Optional
OPEN I-O	Required
READ INVALID KEY	Required
WRITE INVALID KEY	Required
CLOSE	Required

Figure 5-4. Procedure Division Statements for Updating Standard Files

READING STANDARD FILES

Read-only processing for standard files is achieved by specifying the same Environment Division clauses as for updating and using the Procedure Division statements shown in figure 5-5.

USE AFTER ERROR	Optional
OPEN	Required
READ INVALID KEY	Required
CLOSE	Required

Figure 5-5. Procedure Division Statements for Reading Standard Files

READ functions the same as for updating. USE procedures may be used to define additional processing for errors. USE statements are described in section 2.

SAMPLE PROGRAM 4: USING STANDARD FILES

Sample program STANDARD-I-O-EXAMPLE, shown in its entirety in figure 5-6, creates and then updates a standard file, DSKFILE. The original records for the file are contained on a card file, CRDFILE. When a code indicating the end of creation is encountered on CRDFILE, DSKFILE is closed and reopened for updating. A third file, PRTFILE, is maintained to print a record of transactions during file creation and updating. Sample input for this program is shown in figure 5-7, and the output that would result from this input is shown in figure 5-8.

Line 19

The ORGANIZATION IS STANDARD clause is the normal way of specifying standard files.

Line 20 — Line 22

The FILE-LIMIT clause used here could have been omitted, since the maximum number of records it specifies, 4095, is the default. ACCESS IS RANDOM is required for standard files. The ACTUAL KEY clause identifies the data item used for reading and writing.

Line 29

This form of the LABEL RECORDS clause is required for standard files.

Line 52 — Line 54

The record format for DSKFILE includes one character to indicate the last operation on the record, four characters for the record key, and 75 characters for the actual data itself.

Line 57

The most efficient keys for standard files are actual keys described as COMPUTATIONAL-1.

Line 58 — Line 68

MESSAGE-AREA contains various messages to be included in the printed record of transactions for the program.

Line 80 — Line 87

Records are read one at a time from CRDFILE into CRDREC and moved into PRTREC (for PRTFILE) and DSKREC (for DSKFILE). The format of records in DSKFILE is the same as the format of records in CRDFILE.

Line 88 — Line 91

In order to write a record to DSKFILE, the proper key value must be moved to the actual key ACT-KEY. The INVALID KEY option of the WRITE statement is required for standard files.

Line 106 — Line 116

To update the file DSKFILE, transaction requests are read from the card file CRDFILE. The transactions requested are either to read a record and replace it with the new record contained on the card (code 2) or simply to read the record (any other code). All transactions are written to PRTFILE (which will be printed at the end of the job, since its implementor-name is OUTPUT).

Line 117 — Line 126

It is not necessary to move a key value to ACT-KEY just before this write request because the record that was just read is being rewritten, and ACT-KEY is therefore already set to the correct value.

```
00001           IDENTIFICATION DIVISION.
00002
00003           PROGRAM-ID.  STANDARD-I-O-EXAMPLE.
00004
00005           ENVIRONMENT DIVISION.
00006
00007           CONFIGURATION SECTION.
00008           SOURCE-COMPUTER. CYBER.
00009           OBJECT-COMPUTER. CYBER.
00010
00011           INPUT-OUTPUT SECTION.
00012           FILE-CONTROL.
00013           SELECT CRDFILE
00014               ASSIGN TO INPUT.
00015           SELECT PRTFILE
00016               ASSIGN TO OUTPUT.
00017           SELECT DSKFILE
00018               ASSIGN TO DISK01
00019               ORGANIZATION IS STANDARD
00020               FILE-LIMIT IS 4095
00021               ACCESS IS RANDOM
00022               ACTUAL KEY IS ACT-KEY.
00023
00024           DATA DIVISION.
00025
00026           FILE SECTION.
00027
00028           FD  CRDFILE
00029               LABEL RECORDS ARE OMITTED
00030               DATA RECORD IS CRDREC.
00031           01  CRDREC.
00032               05  STDCODE          PICTURE 9.
00033               05  STDKEY           PICTURE 9(4).
00034               05  STDATA           PICTURE X(75).
00035
00036           FD  PRTFILE
00037               LABEL RECORDS ARE OMITTED
00038               DATA RECORD IS PRTREC.
00039           01  PRTREC.
00040               05  SPACE-CONTROL    PICTURE X.
00041               05  MESS-TYPE        PICTURE X(30).
00042               05  STDCODE          PICTURE 9.
00043               05  FILLER           PICTURE X(5).
00044               05  STDKEY           PICTURE 9(4).
00045               05  FILLER           PICTURE X(5).
00046               05  STDATA           PICTURE X(75).
00047
00048           FD  DSKFILE
00049               LABEL RECORDS ARE OMITTED
00050               DATA RECORD IS DSKREC.
00051           01  DSKREC.
00052               05  STDCODE          PICTURE 9.
00053               05  STDKEY           PICTURE 9(4).
00054               05  STDATA           PICTURE X(75).
00055
00056           WORKING-STORAGE SECTION.
00057           77  ACT-KEY              PICTURE 9(4)    COMP-1.
00058           01  MESSAGE-AREA.
00059               05  INV-READ-KEY     PICTURE X(28)   VALUE IS
00060                                    ≠*** INVALID KEY ON READ ***≠.
00061               05  INV-WRITE-KEY    PICTURE X(28)   VALUE IS
00062                                    ≠*** INVALID KEY ON WRITE ***≠.
00063               05  NEW-REC          PICTURE X(14)   VALUE IS
00064                                    ≠    NEW RECORD≠.
00065               05  OLD-REC          PICTURE X(14)   VALUE IS
00066                                    ≠    OLD RECORD≠.
00067               05  UPD-REC          PICTURE X(18)   VALUE IS
00068                                    ≠    UPDATED RECORD≠.
00069
```

Figure 5-6. Sample Program 4: Processing Standard Files (Sheet 1 of 2)

```
00070           PROCEDURE DIVISION.
00071
00072           CREATION SECTION.
00073
00074           BEGIN.
00075               OPEN INPUT  CRDFILE
00076                    OUTPUT PRTFILE DSKFILE.
00077               MOVE ≠0≠ TO PRTREC.
00078
00079           CREATE-REC.
00080               READ CRDFILE
00081                   AT END
00082                       GO TO END-OF-JOB.
00083               IF STDCODE OF CRDREC NOT = 0
00084                   CLOSE DSKFILE
00085                   GO TO UPDATE.
00086               MOVE CORRESPONDING CRDREC TO PRTREC DSKREC.
00087               MOVE NEW-REC TO MESS-TYPE OF PRTREC.
00088               MOVE STDKEY OF CRDREC TO ACT-KEY.
00089               WRITE DSKREC
00090                   INVALID KEY
00091                       MOVE INV-WRITE-KEY TO MESS-TYPE OF PRTREC.
00092               WRITE PRTREC.
00093               GO TO CREATE-REC.
00094
00095           UPDATE SECTION.
00096
00097           REBEGIN.
00098               OPEN I-O DSKFILE.
00099               GO TO UPDATE-REC.
00100
00101           GET-NEXT.
00102               READ CRDFILE
00103                   AT END
00104                       GO TO END-OF-JOB.
00105
00106           UPDATE-REC.
00107               MOVE STDKEY OF CRDREC TO ACT-KEY.
00108               READ DSKFILE
00109                   INVALID KEY
00110                       MOVE CORRESPONDING CRDREC TO PRTREC
00111                       MOVE INV-READ-KEY TO MESS-TYPE OF PRTREC
00112                       WRITE PRTREC
00113                       GO TO GET-NEXT.
00114               MOVE CORRESPONDING DSKREC TO PRTREC.
00115               MOVE OLD-REC TO MESS-TYPE OF PRTREC.
00116               WRITE PRTREC.
00117               IF STDCODE OF CRDREC = 2
00118                   MOVE CORRESPONDING CRDREC TO PRTREC DSKREC
00119                   WRITE DSKREC
00120                       INVALID KEY
00121                           MOVE INV-WRITE-KEY TO MESS-TYPE OF PRTREC
00122                           WRITE PRTREC
00123                           GO TO GET-NEXT.
00124               MOVE UPD-REC TO MESS-TYPE OF PRTREC.
00125               WRITE PRTREC.
00126               GO TO GET-NEXT.
00127
00128           END-OF-JOB SECTION.
00129
00130           END-EXAMPLE.
00131               CLOSE CRDFILE
00132                     PRTFILE
00133                     DSKFILE.
00134               STOP RUN.
```

Figure 5-6. Sample Program 4: Processing Standard Files (Sheet 2 of 2)

```
01000  OF ALL THE SHIPS UPON THE SEA.
01001  THIS ONE ISN≠T THE ONE FOR ME.
01003  TESTING 1-2-3
01004  FOUR SCORE AND SEVEN YEARS AGO
01005  WAS 87 YEARS AGO.
01006  TEST DATA LINE 1
01007  TEST DATA LINE 2
01008  RHUBARB
01009  RUEBARB
01010  RUDEBARB
99999
21001  THIS ONE AIN≠T THE ONE FOR ME≠.
21009  RUE DE LA BARB
21013  RUBBISH
11015  RUBBISH
211016 RUBBISH
11017  THIS IS AN ADDED RECORD
```

Figure 5-7. Standard File Program Input

```
     NEW RECORD                0      1000       OF ALL THE SHIPS UPON THE SEA.
     NEW RECORD                0      1001       THIS ONE ISN≠T THE ONE FOR ME.
     NEW RECORD                0      1003       TESTING 1-2-3
     NEW RECORD                0      1004       FOUR SCORE AND SEVEN YEARS AGO
     NEW RECORD                0      1005       WAS 87 YEARS AGO.
     NEW RECORD                0      1006       TEST DATA LINE 1
     NEW RECORD                0      1007       TEST DATA LINE 2
     NEW RECORD                0      1008       RHUBARB
     NEW RECORD                0      1009       RUEBARB
     NEW RECORD                0      1010       RUDEBARB
*** INVALID KEY ON READ ***    9      9999
     OLD RECORD                0      1001       THIS ONE ISN≠T THE ONE FOR ME.
     UPDATED RECORD            2      1001       THIS ONE AIN≠T THE ONE FOR ME≠.
     OLD RECORD                0      1009       RUEBARB
     UPDATED RECORD            2      1009       RUE DE LA BARB
*** INVALID KEY ON READ ***    2      1013       RUBBISH
*** INVALID KEY ON READ ***    1      1015       RUBBISH
*** INVALID KEY ON READ ***    2      1101     6 RUBBISH
*** INVALID KEY ON READ ***    1      1017       THIS IS AN ADDED RECORD
```

Figure 5-8. Standard File Program Output

A direct file is a mass storage file in which the location of a record is determined by a randomizing computation, called hashing, performed on its primary key. This randomizing process results in the dispersal of records throughout the file, so that the physical order of records in the file is different from any logical order by key, as well as from the order in which they were written. No primary key index is maintained for direct files; in order to locate a record, given its key, the same computation is performed as was performed when the record was written. Direct files are implemented through CYBER Record Manager direct access file organization.

Any of the record types described in section 2 can be used with direct files; they cannot be labeled.

A direct file always has one primary key and can optionally have from 1 to 255 alternate keys. The data item defined as the primary key must have a fixed location and length within the record format. Each record must have a unique primary key value; duplicate primary key values are not allowed for direct files.

Every direct file contains a user-specified number of blocks known as home blocks. A unique positive integer is assigned to each home block. Whenever a record is accessed randomly, the hashing computation is performed on the primary key value specified by the user. The result of the computation is an integer that identifies one of the home blocks. This block is the one the record is written to or read from. Alternate keys are not hashed.

A hashing routine can be provided by the user as a USE FOR HASHING procedure (section 2); otherwise, the system provides a hashing routine by default.

CYBER Record Manager provides a key analysis utility that can be used to test a user-defined hashing routine by determining the distribution of records obtained with that hashing routine. The key analysis utility is discussed in appendix C.

Any program using direct files must specify either ACCESS IS RANDOM or ACCESS IS SEQUENTIAL (SEQUENTIAL is default). The file must be open for input, output, or input/output. The Procedure Division input/output statements permitted under the various access modes and open conditions are shown in table 6-1 and are described in more detail below.

FILE STORAGE

A direct file, as it exists on mass storage after creation, consists of a file statistics table, a user-defined number of home blocks, and any overflow blocks established by the system.

FILE STATISTICS TABLE

The file statistics table (FSTT) is an internal table created and used by CYBER Record Manager. The table is preserved on the file after the file is closed, so that information about the file is available to CYBER Record Manager in subsequent runs using the file. The information that is maintained includes statistics about transactions on the file and file structure specifications such as number of home blocks, size of blocks, key length and location, and maximum and minimum record size. The user should not attempt to change any of these specifications once the file has been created; COBOL allows some of these specifications to be omitted in subsequent programs, as described under Processing Existing Direct Files.

TABLE 6-1. ACCESS MODE AND OPEN CONDITION COMBINATIONS, DIRECT FILES

Statements	Random Access			Sequential Access		
	Open Input	Open Output	Open I-O	Open Input	Open Output	Open I-O
Create New File	N	Y	N	N	N	N
READ INVALID KEY	Y	N	Y	N	N	N
READ KEY IS INVALID KEY	YM	N	YM	N	N	N
READ AT END	Y	N	Y	Y	N	Y
START KEY INVALID KEY	YM	N	YM	YM	N	YM
WRITE INVALID KEY	N	Y	Y	N	N	N
REWRITE INVALID KEY	N	N	Y	N	N	Y
DELETE INVALID KEY	N	N	Y	N	N	Y

Y = Allowed N = Not allowed YM = Allowed multiple index only

HOME BLOCKS

Home blocks are mass storage areas reserved for a direct file when it is created. The user must specify the number of home blocks the file is to contain through the NUMBER OF BLOCKS clause. Once this number is defined, it cannot be changed.

All home blocks for a file are the same size. The user can define the length of the blocks through the BLOCK CONTAINS or RECORD-BLOCK CONTAINS clause. If both these clauses are omitted, COBOL will calculate a default block size calculated from the size of the largest record defined in the File Description entries, and from other file structure specifications. COBOL increases the size of user-defined blocks, if necessary, to an integral multiple of physical record unit (PRU) size, which is 64 words for mass storage, less one word. This length is calculated to utilize mass storage most efficiently.

Record distribution among the blocks is determined by hashing the keys. The number of records allotted to a block depends on the results of hashing. Because of the randomness of the hashing procedure, some blocks will have unused space at the end, as shown in figure 6-1.

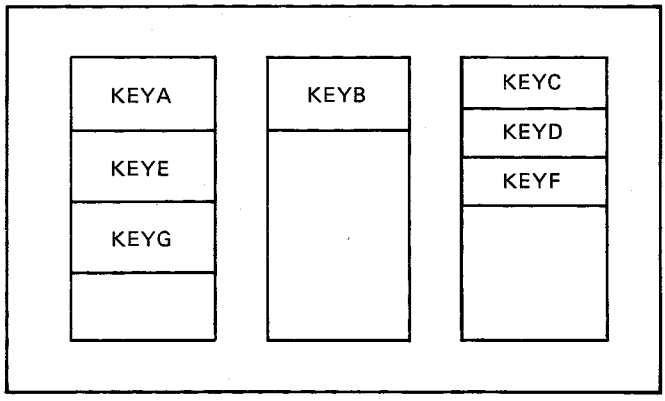

Figure 6-1. Unused Space in Home Blocks

Records whose keys hash to the same value are chained together to ensure ultimate access by primary key to any desired record. If the home block determined by the hashing routine cannot accommodate another record, an additional block, called an overflow block, is allocated and a pointer to the corresponding overflow block is placed in the home block. Each additional overflow block requires an additional access; thus, it is usually more efficient to specify a large number of small home blocks, rather than a small number of large home blocks, so that the need for overflow blocks is reduced.

Even though the home block containing a record or a pointer to the record is located by a hashed key, final access to the record is by its unhashed key. Therefore, two records with identical unhashed primary keys cannot coexist on one file.

OVERFLOW BLOCKS

Overflow blocks are automatically allocated and written by CYBER Record Manager; the COBOL program does not need to acknowledge their existence in any way. They are the same size as home blocks. When an overflow block is required, it is assigned in the same manner that home blocks are assigned. Leftover records from more than one home block can occupy an overflow block. If the unhashed key of a record accessed by primary key cannot be found in its home block, the overflow block is searched. Overflow blocks can point to further overflow blocks, if necessary. Home blocks and the overflow blocks associated with them are illustrated in the diagram in figure 6-2.

The diagram shows that excess records from home blocks 1 and 2 are stored in overflow block 3, along with excess records from overflow block 2. Excess records from home block 4 are assigned to overflow block 2. Excess records from home blocks 3, 5, and 6 are allocated to overflow block 1.

Actual residence of overflow records depends on the setting of an installation parameter. The three possible options are:

1. Abort - no overflow allowed
2. Overflow to other home blocks only. In this case, the amount of mass storage allocated remains constant throughout the life of the file.
3. Overflow to other home blocks and overflow blocks, the choice to be made by CYBER Record Manager

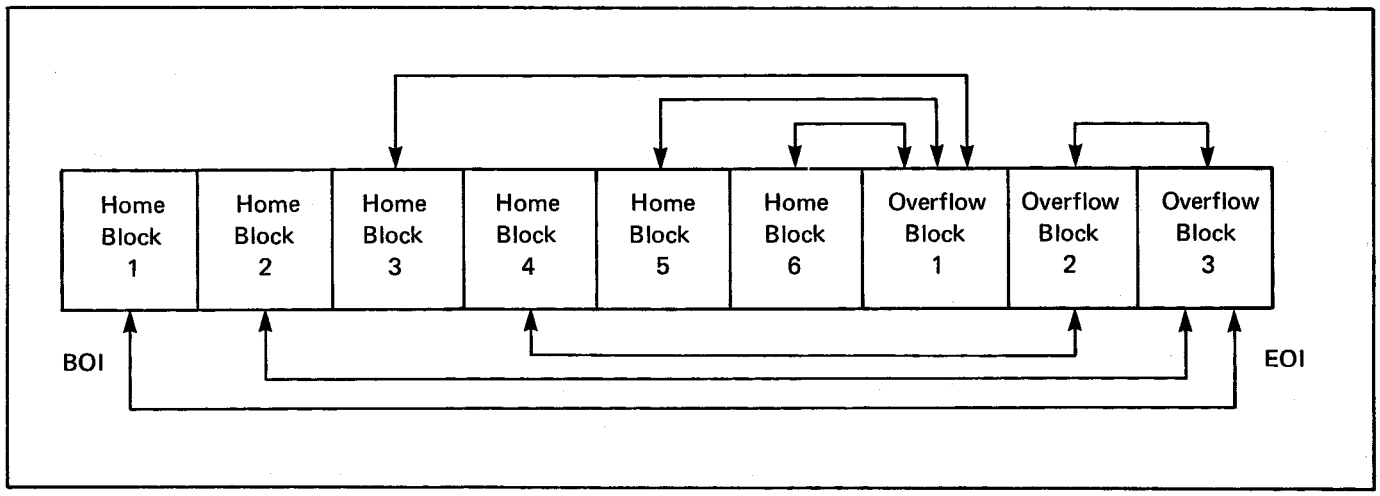

Figure 6-2. Overflow Blocks

CREATING DIRECT FILES

Direct files can be created in one of two ways: by COBOL language statements, or through the CREATE utility. When COBOL statements are used, home blocks are moved from a central memory buffer to mass storage each time a record is written. When the CREATE utility is used, all records that hash to a given home block can be created at once, and the amount of writing time required is thus reduced. Since the amount of time saved with CREATE is significant only when the file contains 1000 or more records, the utility is inefficient for files with fewer records. CREATE is called through control statements (see appendix C).

Direct file creation through COBOL involves the clauses described below. COBOL statements whose operation does not depend on file organization are not included. Clauses and statements that apply only to existing files are described under Processing Existing Files.

ENVIRONMENT DIVISION

The Environment Division clauses used to establish a direct file with COBOL language statements are shown in figure 6-3.

SELECT . . . ASSIGN

The SELECT . . . ASSIGN clause is required for all files:

```
SELECT file-name
ASSIGN TO implementor-name
```

If the file is a multiple index file, the file used to hold the alternate key indexes must also be defined in the ASSIGN clause:

```
SELECT file-name ASSIGN TO
data-file-name index-file-name
```

The second form might appear in a program as:

```
SELECT EMPLOY-FILE
ASSIGN TO EMPLOY EMPLMIP
```

in which EMPLOY would be the direct file created and EMPLMIP would be the index file that contains the index of alternate keys.

Preservation of the index file associated with every multiple index data file is the user's responsibility. In particular, since both files are normally permanent files, control statements to save both files should be included in the job that creates the files, and control statements to attach both files are needed in subsequent jobs that reference the data file. The index file must be attached by every job using the data file, whether or not alternate keys are used in the job.

RESERVE ALTERNATE AREAS

The RESERVE ALTERNATE AREAS clause provides buffer space for one additional home block to be assigned, regardless of the value specified. Thus the clause is meaningful only in the following format:

```
RESERVE 1 ALTERNATE AREA
```

If the clause is omitted or RESERVE NO ALTERNATE AREAS is specified, the minimum buffer space allotted is derived from the block and record information provided by the user.

FILE-LIMITS

The FILE-LIMITS clause can be used to set the maximum number of records that can exist in the file at any time. FILE-LIMITS cannot be used to restrict processing to a portion of a file. Only the format:

```
FILE-LIMIT IS integer
```

can be used. If the FILE-LIMITS clause is omitted, there is no precise limit to the number of records that can be written.

ACCESS IS RANDOM

Direct files can only be written randomly; thus ACCESS IS RANDOM is required on a creation run.

ACTUAL/SYMBOLIC/RECORD KEY

Every direct file has exactly one primary key defined by one of the following clauses:

```
ACTUAL KEY IS data-name
```

SELECT . . . ASSIGN	Required
ORGANIZATION IS DIRECT	Required
RESERVE ALTERNATE AREAS	Optional
FILE-LIMITS	Optional
ACCESS IS RANDOM	Required
ACTUAL/RECORD/SYMBOLIC KEY	Required
ALTERNATE RECORD KEY	Required for multiple-index files
NUMBER OF BLOCKS	Required
RECORD-BLOCK CONTAINS	Optional

Figure 6-3. Environment Division for Direct Files Creation

SYMBOLIC KEY IS data-name

RECORD KEY IS data-name

where data-name defines the key item. The key item must be entirely within the record as specified by the Record Description entry. The length and relative location of the key must be the same for every program using the file. Length of a record or symbolic key can be from 1 to 255 characters; length of an actual key can be from 1 to 9 characters. An actual key must be a COMPUTATIONAL-1 item.

If the system hashing routine is used (no USE FOR HASHING procedure is provided by the user), the usage of the item is irrelevant; for purposes of hashing, only the length of the item and its contents, considered as a bit string, are of importance to CYBER Record Manager. If the user does provide a hashing routine, the usage of the item can be taken into account, as all processing of the key is then under user control. If the direct file is a multiple index file, the key defined by the ACTUAL/SYMBOLIC/RECORD/KEY clause is the unique primary key; alternate keys must be defined through the ALTERNATE RECORD KEY clause.

ALTERNATE RECORD KEY

The ALTERNATE RECORD KEY clause can be used only if an index file, in addition to the direct file, is defined through the ASSIGN clause, and then it is required. When alternate keys are specified, records in the file can be accessed on a key other than the primary key specified in the ACTUAL/-SYMBOLIC/RECORD KEY clause. At least one alternate key must be defined for multiple index files, and as many as 255 can be specified. The alternate key specifications must immediately follow the primary key specification:

```
RECORD KEY IS data-name-1
ALTERNATE RECORD KEY IS data-name-2
ALTERNATE RECORD KEY IS data-name-3
.
.
.
```

The index file specified in the ASSIGN clause is used to hold alternate key indexes which are established and maintained automatically by CYBER Record Manager for multiple index files. One index is created for each alternate key field. Within the index, one entry is made for each alternate key value encountered as records are written. A primary key value is associated with each alternate key value for each data record with that alternate key value. The alternate key entries are kept in sorted order by CYBER Record Manager. Every time a record is added or deleted, the index file is updated to reflect all the alternate key values of the record in question. Index file structure is shown in figure 6-4.

If the DUPLICATES option of the ALTERNATE RECORD KEY clause is not specified, alternate key values must be different for each record in the file. In this case, a duplicate value encountered on a write causes execution of the INVALID KEY clause. If DUPLICATES is specified, duplicate values are allowed. The order of primary key values within a

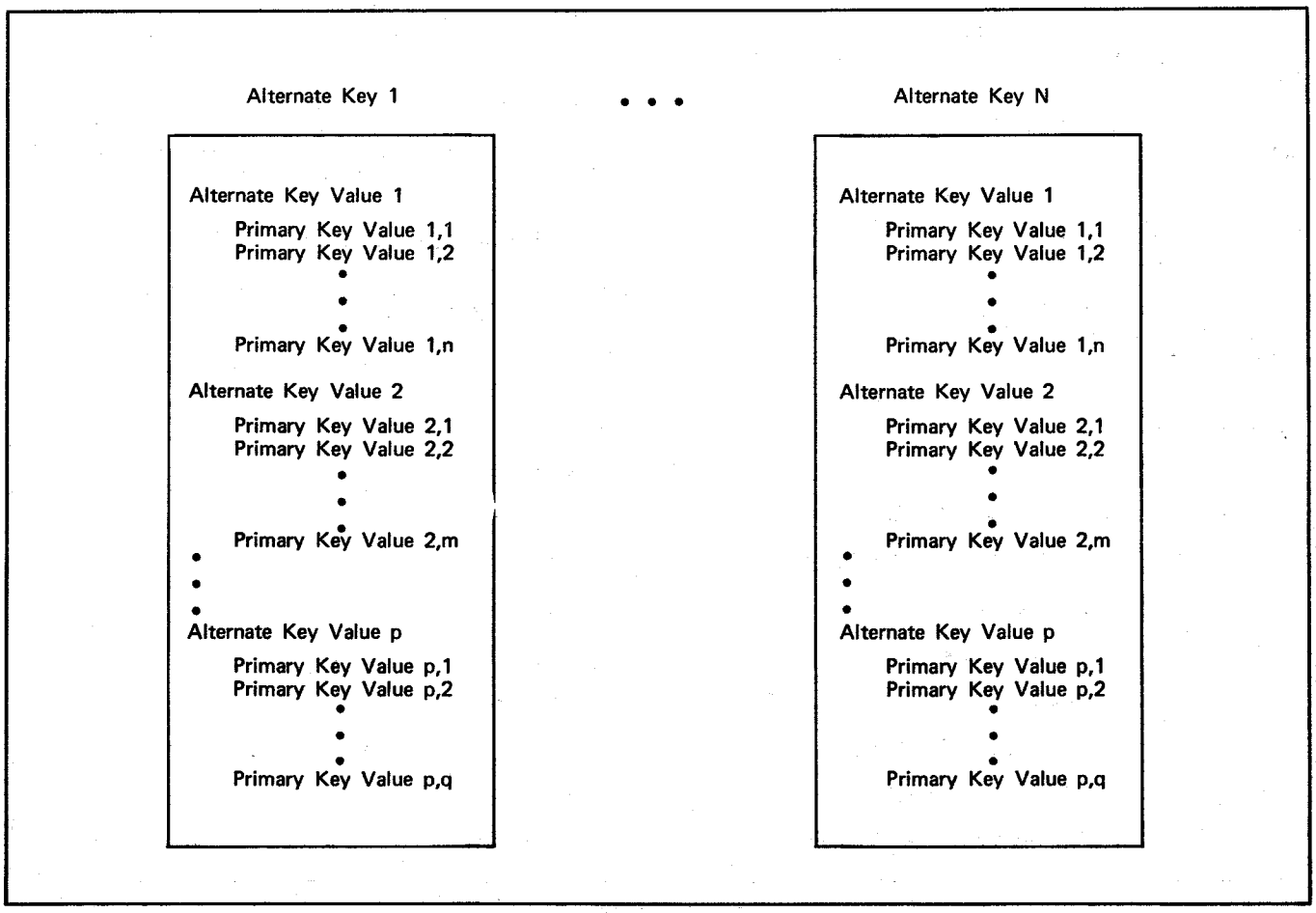

Figure 6-4. Index File Structure

set of duplicates depends on the presence or absence of the INDEXED option. If INDEXED is omitted, primary key values associated with a given alternate key value are maintained in the order in which they are written to the file (first in, first out). If INDEXED is specified, the primary keys are kept in the same order as primary keys in an indexed sequential file (section 7). Performance in file updating is considerably enhanced when INDEXED is specified.

Alternate keys can overlap and differ in length, but none can begin in the same location as the primary key or any other alternate key. The alternate key fields defined at file creation are normally retained for the life of the file, but they can be changed or altered, at user option, by using the IXGEN utility (see appendix C).

If the ALTERNATE RECORD KEY clause is omitted, the file does not have multiple index status and access can be only through the primary key.

NUMBER OF BLOCKS

The NUMBER OF BLOCKS clause must be specified for direct file creation to establish the number of home blocks to be used and to reserve the blocks on mass storage. With most hashing schemes, a better distribution is achieved if the number of home blocks is defined as a prime number:

```
NUMBER OF BLOCKS IS 73
```

If the available space in a home block is exceeded, an overflow block is assigned and performance is degraded. For this reason, block sizes should be small and the number of home blocks should be large.

RECORD-BLOCK CONTAINS

The RECORD-BLOCK CONTAINS clause can be used in lieu of the BLOCK CONTAINS clause to specify the block size of a direct file at creation:

```
RECORD-BLOCK CONTAINS integer RECORDS
```

or:

```
RECORD-BLOCK CONTAINS integer CHARACTERS
```

If the RECORDS option is used in the clause, COBOL converts the number of records to characters based on the maximum record length specified. If the CHARACTERS option is used, the size must be greater than zero and must not exceed 327,670 characters. In both cases, COBOL adjusts the block size up to a multiple of 640 characters (PRU size) less 10 characters (one word). If the RECORD-BLOCK CONTAINS clause is omitted, the BLOCK CONTAINS clause in the Data Division can be used to specify record block size. If both clauses are used, BLOCK CONTAINS takes precedence. If both clauses are omitted, a default block size calculated from other clauses is specified.

DATA DIVISION

The clauses used in the Data Division to declare direct files are shown in figure 6-5.

BLOCK CONTAINS	Optional
LABEL RECORD IS OMITTED	Required

Figure 6-5. Data Division Clauses for Direct File Creation

BLOCK CONTAINS

BLOCK CONTAINS should be used if the RECORD-BLOCK CONTAINS clause is not used:

```
BLOCK CONTAINS integer RECORDS
```

or:

```
BLOCK CONTAINS integer CHARACTERS
```

If RECORDS is specified, COBOL converts the number of records to characters based on the maximum record size and rounds the block size up to a multiple of 640 characters less 10. The same rounding occurs if CHARACTERS is specified. The calculated block size is then used for both home and overflow blocks. If the clause is omitted and a RECORD-BLOCK CONTAINS clause is not defined for the file, the default size is calculated from the other clauses provided.

LABEL RECORD

Labels cannot be used with direct files; the clause LABEL RECORD IS OMITTED is required.

PROCEDURE DIVISION

The statements used to create a direct file are shown in figure 6-6.

USE AFTER ERROR	Optional
USE FOR HASHING	Optional
OPEN OUTPUT	Required
WRITE INVALID KEY	Required
CLOSE	Required

Figure 6-6. Procedure Division Statements for Creating Direct Files

USE AFTER ERROR PROCEDURE can be used with OUTPUT or with one or more file names to indicate routines to be executed following an error in file creation.

The programmer can specify the USE FOR HASHING clause to define a hashing routine other than the system routine to hash the record key.

USE procedures are discussed in section 2.

Direct files must be opened for creation with OPEN OUTPUT:

```
OPEN OUTPUT file-name
```

WRITE must be used with the INVALID KEY option:

```
WRITE record-name
INVALID KEY imperative-statement
```

The key item must be set before each WRITE is executed. When the WRITE statement is processed, the record is added to the block indicated by the hashed key value. If the block is full, an overflow block is assigned to hold the leftover record. If the key already exists or the total number of records in the file exceeds the specifications of the FILE-LIMITS clause, the INVALID KEY clause is executed.

PROCESSING EXISTING DIRECT FILES

Records can be added, deleted, rewritten, or read when an existing direct file is open for I-O. Access can be by alternate key (if alternate keys have been defined) or by primary key. An existing direct file open for INPUT can only be read. Operations permitted for each open condition are shown in table 6-1.

Because of the file statistics table described above, it is not necessary to repeat specification of block size and number of home blocks; the clauses NUMBER OF BLOCKS, RECORD-BLOCK CONTAINS, and BLOCK CONTAINS can be omitted on an update run. If they are included, however, they must specify the same values as those with which the file was created. Except for optional omission of these clauses, the Data and Environment Division clauses used when updating a direct file are the same as those used when the file is created.

Procedure Division statements used for updating a direct file are shown in figure 6-7.

USE AFTER ERROR	Optional
USE FOR HASHING	Optional
OPEN INPUT OPEN I-O	One Required
READ INVALID KEY	Optional
WRITE INVALID KEY	Optional
REWRITE INVALID KEY	Optional
DELETE INVALID KEY	Optional
READ KEY IS INVALID KEY	Optional; used for multiple index only
START KEY INVALID KEY	Optional; used for multiple index only
READ NEXT AT END	Optional
CLOSE	Required

Figure 6-7. Procedure Division for Updating Direct Files

PRIMARY KEY ACCESS

In order to access a record in a direct file by primary key, the INVALID KEY option must be included in the input/output statement. When this option is used, the contents of the key item defined in the ACTUAL/SYMBOLIC/RECORD KEY clause are hashed and used to determine the location of the desired record. Access takes place in the same manner whether or not the file is a multiple index file. The INVALID KEY clause is executed when the key specified does not match the key of any record in the file (READ, REWRITE, and DELETE only), when writing the record would cause the FILE-LIMITS specified to be exceeded (WRITE only), or when the key provided duplicates the key of an existing record (WRITE only). Whenever an input/output error occurs, the special-register ERROR-CODE is set to the number of the CYBER Record Manager error that has occurred. These numbers are listed in the Record Manager Reference Manual. It is advisable for a program to check the contents of ERROR-CODE whenever the INVALID KEY clause is executed.

READ INVALID KEY locates a record according to the contents of the primary key item and returns it to the input record area.

WRITE INVALID KEY adds a new record to the file; the number of the home block it is written to is determined by hashing the contents of the primary key item.

REWRITE INVALID KEY replaces an existing record by a new record with the same primary key. Following the rewrite, the new record is no longer available in the output record area.

DELETE INVALID KEY removes an existing record from the file and frees the space it occupied for records written subsequently.

ALTERNATE KEY ACCESS

READ KEY IS INVALID KEY reads a multiple index file record by alternate key:

```
READ file-name KEY IS data-name
INVALID KEY imperative-statement
```

The data-name specified as the object of the KEY IS option must be an item previously defined as an alternate key by the ALTERNATE RECORD KEY clause. When this statement is executed, the index file is searched until the index entries for the alternate key value that matches the contents of data-name are located. The record whose primary key value occurs first in the entries for the alternate key value is returned to the input record area. If DUPLICATES was not specified in the ALTERNATE RECORD KEY clause, only one primary key value is associated with the alternate key value. If DUPLICATES was specified, the record returned by READ KEY IS INVALID KEY is the record whose primary key occurs first in the group of primary keys associated with the alternate key value. The data-name referenced in the KEY IS option can be a leading portion of an alternate key item, rather than the whole item. In this case, the first key value in the index file whose leading portion matches that provided by the user determines the record read.

START KEY INVALID KEY is used to position a multiple index file without reading a record:

```
START file-name
KEY relational-operator data-name
INVALID KEY imperative-statement
```

Relational-operator must be one of the following:

IS EQUAL TO
IS =
IS GREATER THAN
IS >
IS NOT LESS THAN
IS NOT <

The data-name must be an alternate key item, or the leading portion of an alternate key item.

Execution of the START statement establishes the key of reference by positioning the index file to the first alternate key value that meets the specified condition. The search begins either from the current key of reference or from the beginning of the index file if the key of reference has not yet been established. The key that satisfies the condition becomes the new key of reference. If the comparison is not satisfied by any alternate key value, the INVALID KEY clause is executed.

Successful execution of READ INVALID KEY or START establishes a key of reference for purposes of future access to the file. The key of reference is the primary or alternate key of the record read or located. Once the key of reference has been established, it can only be changed by execution of another START or READ INVALID KEY statement.

If the READ is by primary key (KEY IS is omitted), the new key of reference is the primary key value read. Subsequent sequential reads return records in their order in the data file; any index file positioning established by START or READ KEY IS INVALID KEY is lost. If the READ is by alternate key, the new key of reference is the alternate key value, and subsequent sequential reads return records in the order their keys occur in the index file entries.

The importance of the key of reference is that it determines the order in which records are read by subsequent READ NEXT AT END statements. READ NEXT AT END specifies sequential reading of records. If a key of reference has been established, records are returned to the user program in the order in which their primary keys occur in the alternate key index. When the last record with a given alternate key value has been read, the special-register ERROR-CODE is set to 1000. No other indication is made that the end of the list for that alternate key value has been reached. The next time READ NEXT AT END is executed, if the key of reference has not been changed, the first primary key in the list for the next alternate key value is used to read a record from the data file.

If READ NEXT AT END is executed before a key of reference has been established, reading of the records in the data file takes place according to the physical order of records in the file; the alternate key index is not referenced.

READ ONLY PROCESSING

When a direct file is opened for INPUT, only READ INVALID KEY and READ NEXT AT END can be used; if the file is a multiple index file, READ KEY IS INVALID KEY and START can also be used. In other words, any of the information in the file can be accessed, but no changes can be made to the file itself. This kind of processing is called read-only processing; it is very useful to protect the file from accidental alteration.

For a file without multiple index structure, if the access mode is sequential, only READ NEXT AT END can be executed. In this case, records are read in the physical order in which they occur on the file. Since direct file records are scattered, the order in which they are returned by sequential reads bears no relation either to the order in which they were written, or to any logical, key-oriented order. Therefore, this method of access is primarily useful when all the records in the file are to be read, and the order in which they are read is not important.

For a multiple index file, when the access mode is sequential, the START statement can be executed as well as READ NEXT AT END. The START statement positions the file at the first primary key value or alternate key value satisfying the specified comparison. Subsequent execution of READ NEXT AT END returns records in an order that depends on whether the key specified by START was a primary or alternate key. If primary, records are returned in physical order (the same as for a non-multiple-index file). If alternate, records are returned in order by alternate key; within each alternate key value, records are returned in the order written, or, if INDEXED was specified in the ALTERNATE KEY clause, in order by primary key.

SAMPLE PROGRAM 5: USING DIRECT FILES

Sample program 5, shown in figure 6-8, uses the direct file RANFILE to maintain a table of word occurrences. Each record in RANFILE contains a count of the number of occurrences of a given word in the card file TEXT-FILE, as well as a list of those occurrences. TEXT-FILE contains the text of the program itself; word occurrences are identified by the number of the line that the word occurs in. Once the table is built, it is interrogated by queries from a second card file, QUERY-FILE. An example of output from this program is shown in figure 6-9.

Line 13 — Line 18

Among the FILE-CONTROL clauses for RANFILE are the SELECT and ASSIGN clauses, required for all files. The clause RESERVE 1 AREA specifies that one extra area is to be reserved for use as a buffer; this is the maximum number of extra areas that can be reserved for direct files. ORGANIZATION IS DIRECT is required for direct files; ACCESS IS RANDOM is required for direct file creation. The SYMBOLIC KEY clause indicates that the key item is SOURCE-WORD, which is within the fixed length portion of the record format for RANFILE, as required. ACTUAL or SYMBOLIC KEY could have been used instead; all three are the same for direct files. The NUMBER OF BLOCKS clause (required) specifies a prime number, as recommended above.

Line 19 — Line 20

The ASSIGN clauses for QUERY-FILE and TEXT-FILE both use the -FZ suffix, since both files originated as card files but neither currently has the implementor-name INPUT. -FZ ensures Z type records for these files; the BLOCK CONTAINS clauses used ensure C type blocks.

Line 27 — Line 28

The clause LABEL RECORDS OMITTED is required. The BLOCK CONTAINS clause could have been omitted, in which case COBOL would compute a block size based on largest record size. Since 5110 characters is specified (eight PRUs less one word), the block size is not rounded up.

```
00001            IDENTIFICATION DIVISION.
00002
00003            PROGRAM-ID. WORD-LOCATOR.
00004
00005            ENVIRONMENT DIVISION.
00006
00007            CONFIGURATION SECTION.
00008            SOURCE-COMPUTER. CYBER.
00009            OBJECT-COMPUTER. CYBER.
00010
00011            INPUT-OUTPUT SECTION.
00012            FILE-CONTROL.
00013                SELECT RANFILE ASSIGN TO DISC01
00014                RESERVE 1 AREA
00015                ORGANIZATION IS DIRECT
00016                ACCESS MODE IS RANDOM
00017                SYMBOLIC KEY IS SOURCE-WORD
00018                NUMBER OF BLOCKS IS 101.
00019            SELECT  QUERY-FILE ASSIGN  QUERIES-FZ.
00020            SELECT  TEXT-FILE ASSIGN SOURCE-FZ.
00021
00022            DATA DIVISION.
00023
00024            FILE SECTION.
00025
00026            FD RANFILE
00027                LABEL RECORDS OMITTED
00028                BLOCK CONTAINS 5110 CHARACTERS
00029                DATA RECORD IS RANREC.
00030            01 RANREC.
00031                02  SOURCE-WORD  PIC X(30).
00032                02  OCCURRENCE-COUNT  PIC 999.
00033                02  LINE-OCCURRENCE  PIC 9(5) OCCURS
00034                    1 TO 100 TIMES DEPENDING ON OCCURRENCE-COUNT.
00035
00036            FD  TEXT-FILE
00037                LABEL RECORDS OMITTED
00038                BLOCK CONTAINS 640 CHARACTERS
00039                DATA RECORD SOURCE-REC.
00040            01 SOURCE-REC.
00041                02  SEQ-NO PIC 9(6).
00042                02  FILLER PIC X.
00043                02  TEXT-CHARACTERS.
00044                    03 IN-CHAR PIC X OCCURS 65 TIMES.
00045                02  FILLER PIC X(8).
00046
00047            FD  QUERY-FILE
00048                LABEL RECORDS OMITTED
00049                BLOCK CONTAINS 640 CHARACTERS
00050                DATA RECORD QUERY-REC.
00051            01  QUERY-REC.
00052                02  QUERY PIC X(30).
00053                02  FILLER PIC X(50).
00054
00055            WORKING-STORAGE SECTION.
00056                77  SUB-A PIC 99 COMP-1 VALUE 70.
00057                77  SUB-B PIC 99 COMP-1.
00058                77  LINE-COUNT PIC 9(5) VALUE 0.
00059                77  SUB-C PIC 99 COMP-1.
00060                77  GARBAGE PIC 9(7)  COMP-1.
00061            01 WORD.
00062                02 W-CHARACTER PIC X OCCURS 30 TIMES.
00063            01  RE-WORD REDEFINES WORD.
00064                    02 36-BITS PIC X(6) OCCURS 5 TIMES.
```

Figure 6-8. Sample Program 5: Direct Files (Sheet 1 of 3)

```
00065           01  WORK-FIELD.
00066               02 BINARY-ITEM PIC 9(7) COMP-1.
00067               02 SPLITTER REDEFINES BINARY-ITEM.
00068                   03 EXPONENT PIC X(4).
00069                   03 MANTISSA PIC X(5).
00070
00071           PROCEDURE DIVISION.
00072
00073           DECLARATIVES.
00074
00075           HASHER SECTION.
00076
00077               USE FOR HASHING ON RANFILE.
00078
00079           HASH-PARA.
00080               MOVE SOURCE-WORD TO WORD.
00081               MOVE ZEROES  TO BINARY-ITEM HASHED-VALUE.
00082               PERFORM 36-BIT-ADDER  VARYING SUB-C
00083                     FROM 1 BY 1 UNTIL  SUB-C = 6.
00084               DIVIDE HASHED-VALUE BY 101
00085                   GIVING  GARBAGE REMAINDER BINARY-ITEM.
00086               MOVE BINARY-ITEM TO HASHED-VALUE.
00087               GO TO DECLARATIVE-EXIT.
00088
00089           36-BIT-ADDER.
00090               MOVE 36-BITS (SUB-C) TO MANTISSA.
00091                  ADD BINARY-ITEM TO  HASHED-VALUE.
00092
00093           DECLARATIVE-EXIT.
00094               EXIT.
00095           END DECLARATIVES.
00096
00097           START.
00098               OPEN INPUT TEXT-FILE.
00099               OPEN OUTPUT RANFILE.
00100
00101           CREATE-MORE-RECS.
00102               PERFORM CREATE-INITIAL-REC  2 TIMES
00103               CLOSE RANFILE.
00104               OPEN I-O RANFILE.
00105
00106           MAIN-BUILD-LOOP.
00107               PERFORM GET-A-WORD  THRU GAW-EXIT.
00108               MOVE WORD TO SOURCE-WORD.
00109               READ RANFILE  INVALID KEY GO TO CREATE-NEW.
00110
00111           UPDATE-EXISTING.
00112               IF OCCURRENCE-COUNT = 99
00113                   DISPLAY WORD ≠ OCCURS TOO OFTEN - OCCURRENCE DROPPED≠
00114                   GO TO MAIN-BUILD-LOOP.
00115               ADD 1 TO OCCURRENCE-COUNT
00116               MOVE LINE-COUNT TO LINE-OCCURRENCE (OCCURRENCE-COUNT).
00117               REWRITE RANREC INVALID KEY  DISPLAY ≠ FAULT 1 ≠ STOP RUN.
00118               GO TO MAIN-BUILD-LOOP.
00119
00120           CREATE-NEW.
00121               MOVE WORD TO SOURCE-WORD.
00122               MOVE 1 TO OCCURRENCE-COUNT.
00123               MOVE LINE-COUNT TO LINE-OCCURRENCE (1).
00124               WRITE RANREC INVALID KEY DISPLAY RANREC STOP RUN.
00125
00126           DONE-CREATING.
00127               GO TO MAIN-BUILD-LOOP.
00128
00129           QUERY-PHASE.
00130               CLOSE RANFILE.
00131               OPEN INPUT RANFILE.
```

Figure 6-8. Sample Program 5: Direct Files (Sheet 2 of 3)

```
00132               CLOSE TEXT-FILE.
00133               OPEN INPUT QUERY-FILE.
00134
00135           QUERY-LOOP.
00136               READ  QUERY-FILE AT END  GO TO WRAP-UP.
00137               MOVE  QUERY TO SOURCE-WORD.
00138               READ  RANFILE INVALID  KEY GO TO  NO-SUCH-WORD.
00139               EXAMINE QUERY REPLACING ALL SPACES BY ≠*≠.
00140               DISPLAY
00141                   QUERY
00142                   ≠ IS FOUND ≠
00143               OCCURRENCE-COUNT
00144               ≠ TIMES  AS FOLLOWS ≠.
00145               PERFORM LINE-DISPLAY  VARYING SUB-A
00146                   FROM 1 BY 1 UNTIL  SUB-A >  OCCURRENCE-COUNT.
00147               GO TO QUERY-LOOP.
00148
00149           LINE-DISPLAY.
00150               IF  SUB-A = 1
00151                     DISPLAY  ≠ LINES ≠ LINE-OCCURRENCE (SUB-A)
00152               ELSE DISPLAY ≠      ≠ LINE-OCCURRENCE(SUB-A).
00153
00154           NO-SUCH-WORD.
00155               EXAMINE QUERY REPLACING ALL SPACES BY ≠*≠.
00156               DISPLAY ≠ NO OCCURRENCES FOR ≠ QUERY.
00157               GO TO QUERY-LOOP.
00158
00159           GET-A-WORD.
00160               IF SUB-A > 65
00161               PERFORM CARD-READ
00162                   MOVE 1 TO SUB-A.
00163               IF  IN-CHAR (SUB-A) = SPACE
00164                   ADD 1 TO SUB-A
00165                GO TO GET-A-WORD.
00166               MOVE 1 TO SUB-B.
00167               MOVE SPACES TO WORD.
00168
00169           PEEL.
00170               IF SUB-B > 30
00171                   PERFORM SKIP-WORD UNTIL IN-CHAR (SUB-A) = SPACES
00172                   GO TO GET-A-WORD.
00173               MOVE IN-CHAR (SUB-A) TO W-CHARACTER  (SUB-B).
00174               ADD 1 TO SUB-A SUB-B.
00175              IF IN-CHAR (SUB-A) NOT = SPACES  GO TO PEEL.
00176                IF SUB-B > 1 AND W-CHARACTER (SUB-B - 1) = ≠.≠
00177                   MOVE SPACE TO W-CHARACTER (SUB-B - 1).
00178
00179           GAW-EXIT.
00180               EXIT.
00181
00182           SKIP-WORD.
00183               ADD 1 TO SUB-A.
00184
00185           CARD-READ.
00186               READ TEXT-FILE AT END GO TO QUERY-PHASE.
00187               ADD 1 TO LINE-COUNT.
00188               EXAMINE TEXT-CHARACTERS REPLACING ALL ≠(≠ BY SPACE.
00189               EXAMINE TEXT-CHARACTERS REPLACING ALL ≠)≠ BY SPACE.
00190
00191           CREATE-INITIAL-REC.
00192               PERFORM GET-A-WORD THRU GAW-EXIT.
00193               PERFORM CREATE-NEW.
00194
00195           WRAP-UP.
00196               CLOSE RANFILE QUERY-FILE.
00197               STOP RUN.
```

Figure 6-8. Sample Program 5: Direct Files (Sheet 3 of 3)

```
INVALID********************** IS FOUND 004 TIMES  AS FOLLOWS
 LINES 00109
       00117
       00124
       00138
BLOCKS************************ IS FOUND 001 TIMES  AS FOLLOWS
```

Figure 6-9. Output from Sample Program 5

Line 30 — Line 34

The DEPENDING ON option in the Record Description entry for RANFILE instructs COBOL to select T type records. The fixed length header for these records consists of SOURCE-WORD and OCCURRENCE-COUNT; each instance of LINE-OCCURRENCE is one trailer item. The record length varies from 38 to 533 characters. SOURCE-WORD contains the word whose occurrences are tabulated; it also serves as the primary key, to be hashed on record access. OCCURRENCE-COUNT keeps a count of the number of times the word occurs in the file, and LINE-OCCURRENCE contains the line number of each occurrence. Only the first 99 occurrences of each word are tallied.

Line 36 — Line 53

The File Description entries for TEXT-FILE and QUERY-FILE indicate that both files are to contain 80-character records. Type C blocking is selected by COBOL for both files.

Line 55 — Line 69

The Working-Storage Section defines various entities used during program execution. SUB-A, SUB-B, and SUB-C are used as subscripts in the program. LINE-COUNT keeps a count of the records in TEXT-FILE in case the program deck is punched without sequence numbers. This count is then used to fill in LINE-OCCURRENCE in the output file RANFILE. WORD, redefined by RE-WORD, is a work area used in the hashing routine. WORK-FIELD is used to compute the hashed value of the key.

Line 73 — Line 95

A simple example of a hashing routine is provided here. The user does not have to provide a hashing routine; the CYBER Record Manager default hashing routine is used if none is provided. Whatever hashing routine is used must be used by every program accessing the direct file.

After the hashed value is computed, it must be placed in HASHED-VALUE, a COBOL special-register defined as an eight-digit COMPUTATIONAL-1 (integer) item. The computed value must lie between zero and the number of blocks specified in the NUMBER OF BLOCKS clause minus 1.

Line 97 — Line 104

TEXT-FILE is opened as an input file and RANFILE is opened as an output file (required for creation). Execution of CREATE-MORE-RECS causes two initial records to be written to RANFILE with TEXT-FILE used as the input source file. Two is an arbitrary number selected to ensure that the initial creation terminates before duplicate words are encountered on TEXT-FILE.

Line 106 — Line 109

Now that two records exist on RANFILE, the file is reopened as an input/output file to allow checking for multiple occurrences of words on TEXT-FILE. When the INVALID KEY clause of the READ statement is executed (the record does not already exist on RANFILE), additional records are written to RANFILE by executing the CREATE-NEW paragraph.

Line 111 — Line 118

UPDATE-EXISTING is performed when the READ of RANFILE is successful. This paragraph updates existing records on RANFILE by incrementing OCCURRENCE-COUNT and storing another entry in LINE-OCCURRENCE. Only 99 entries for LINE-OCCURRENCE can be stored in each record; if a word occurs more than 99 times the excess record entries are dropped and a message is displayed on the console indicating the word.

Line 120 — Line 124

When CREATE-NEW is performed, a new record is added to RANFILE.

Line 126 — Line 147

When TEXT-FILE is exhausted, RANFILE is closed as an input/output file and reopened as an input file, and QUERY-FILE is opened to select and display information from RANFILE.

Line 149 — Line 157

When a QUERY entry matches an entry on RANFILE, the word is displayed along with the number of times it occurred and the first line number on which it occurred. If no match is found, a message to that effect is displayed and followed by a display of the missing word.

Line 159 — Line 189

The paragraphs from GET-A-WORD to CARD-READ read input from TEXT-FILE, character by character, and format words to be used in RANFILE. Terminators and some special characters are changed to blanks before words are placed on RANFILE.

Line 191 — Line 197

Executing CREATE-INITIAL-REC causes a record to be placed on RANFILE.

When all processing is finished, RANFILE is closed and the run is completed.

SAMPLE PROGRAM 6: USING MULTIPLE INDEX DIRECT FILES

Sample program 6, shown in figure 6-10, creates a multiple index direct file (NAME-FILE), closes it, and reopens it in read-only mode. One primary key, KEY-1, and three alternate keys, PERS-NO, COLOR, and TEAM-TI, are defined for NAME-FILE; COLOR and TEAM-TI are not used for record access in this program. The input for the program is shown in figure 6-11, and the output in figure 6-12.

Line 13 — Line 14

CRDFILE and CARD-FILE are successive sections on the card file INPUT; their file structure characteristics are implicitly established by the implementor-name INPUT. CRDFILE contains the original records that NAME-FILE is created from; CARD-FILE contains a key used to position NAME-FILE for reading.

```
00001            IDENTIFICATION DIVISION.
00002
00003            PROGRAM-ID. DIRECT-ACCESS-MIP.
00004
00005            ENVIRONMENT DIVISION.
00006
00007            CONFIGURATION SECTION.
00008            SOURCE-COMPUTER. CYBER.
00009            OBJECT-COMPUTER. CYBER.
00010
00011            INPUT-OUTPUT SECTION.
00012            FILE-CONTROL.
00013                SELECT CRDFILE ASSIGN TO INPUT.
00014                SELECT CARD-FILE ASSIGN TO INPUT.
00015                SELECT NAME-FILE ASSIGN OFILE MIP2
00016                ORGANIZATION IS DIRECT
00017                ACCESS IS RANDOM
00018                NUMBER OF BLOCKS IS 20
00019                SYMBOLIC KEY IS KEY-1
00020                ALTERNATE RECORD KEY IS PERS-NO
00021                ALTERNATE RECORD KEY IS COLOR  DUPLICATES
00022                ALTERNATE RECORD KEY IS TEAM-TI  DUPLICATES.
00023
00024            DATA DIVISION.
00025
00026            FILE SECTION.
00027
00028            FD CRDFILE
00029                LABEL RECORDS ARE OMITTED
00030                DATA RECORDS ARE CARD-REC.
00031            01  CARD-REC.
00032                02  IN-KEY1      PIC X(18).
00033                02  IN-AMT       PIC 9(8).
00034                02  IN-COLOR     PIC X(8).
00035                02  I-DATE-A     PIC 9(5).
00036                02  IN-RANK1     PIC X(20).
00037
00038            FD  CARD-FILE
00039                LABEL RECORDS ARE OMITTED
00040                DATA RECORDS ARE CARD-RECORD.
00041            01  CARD-RECORD.
00042                02  INPERS-NO    PIC 9(4).
00043                02  FILLER       PIC X(76).
00044
00045            FD  NAME-FILE
00046                LABEL RECORDS ARE OMITTED
00047                BLOCK CONTAINS 640 CHARACTERS
00048                DATA RECORDS ARE PERS-REC.
00049            01  PERS-REC.
00050                02  KEY-1.
00051                    03  FILLER  PIC X(6).
00052                    03  PERS-NO PIC 9(4).
00053                    03  CITY    PIC X(8).
00054                02  AMT         PIC 9(6)V99.
00055                02  COLOR       PIC X(8).
00056                02  DATE-A      PIC 9(5).
```

Figure 6-10. Sample Program 6: Direct Access Multiple Index (Sheet 1 of 2)

```
00057                02  RANK.
00058                    03  TITLEA  PIC X(11).
00059                    03  TEAM-TI   PIC X(9).
00060
00061           WORKING-STORAGE SECTION.
00062
00063           PROCEDURE DIVISION.
00064
00065           CREATE-RANDOM-FILE.
00066               OPEN INPUT CRDFILE
00067                OUTPUT NAME-FILE.
00068
00069           READ1.
00070               READ CRDFILE AT END GO TO END-CREATE.
00071               MOVE IN-KEY1 TO KEY-1.
00072               MOVE IN-COLOR TO COLOR.
00073               MOVE IN-AMT TO AMT.
00074               MOVE I-DATE-A TO DATE-A.
00075               MOVE IN-RANK1 TO RANK.
00076               WRITE PERS-REC  INVALID KEY DISPLAY
00077                 ≠INVALID KEY IS ≠ KEY-1
00078                    GO TO END-RUN.
00079               GO TO READ1.
00080
00081           END-CREATE.
00082               CLOSE  CRDFILE, NAME-FILE.
00083
00084           INITIALIZE-ALL.
00085               OPEN INPUT NAME-FILE, CARD-FILE.
00086
00087           SELECT-REC.
00088               READ CARD-FILE AT END GO TO END-RUN.
00089               MOVE  INPERS-NO TO PERS-NO.
00090               START NAME-FILE KEY IS = PERS-NO
00091                   INVALID KEY DISPLAY ≠INVALID KEY - KEY IS≠ INPERS-NO
00092                   GO TO END-RUN.
00093
00094           READ-II.
00095               READ NEXT NAME-FILE AT END GO TO END-RUN.
00096               DISPLAY PERS-REC.
00097               GO TO READ-II.
00098
00099           END-RUN.
00100               CLOSE NAME-FILE, CARD-FILE.
00101               STOP RUN.
```

Figure 6-10. Sample Program 6: Direct Access Multiple Index (Sheet 2 of 2)

```
PERSON0079BROOKFIE90497850BROWN   72123DRAFTSMAN  ASST TRS
PERSON0123CHICAGO 00300198BLUE    69043PROGRAMMER VICE PRES
PERSON0456CHICAGO 00215835PURPLE  69152CLERK      PRESIDENT
PERSON0796WINNETKA00792354BLUE    69187TYPIST     COMMITTEE
PERSON2272HINSDALE50053916YELLOW  72189TRAINEE    COMMITTEE
PERSON2459EVANSTON00254981BROWN   70219MANAGER    CHAIRMAN
PERSON3313CICERO  06061472YELLOW  732900PERATOR   V CHAIRMAN
PERSON3796BATAVIA 21856379YELLOW  70123PROGRAMMER MEMBER
PERSON4845CHICAGO 70275138YELLOW  71301ENGINEER   SECRETARY
PERSON4890BATAVIA 10213576PURPLE  70154ANALYST    MEMBER
PERSON5167LA GRANG02426554BLUE    69265SECRETARY  TREASURER
PERSON5968LYONS   08682394BROWN   74012SECRETARY  MEMBER
PERSON6594CHICAGO 01597624PURPLE  69152ENGINEER   COMMITTEE
PERSON6894LYONS   30637732PURPLE  71368CLERK-TYP  MEMBER
PERSON7156WESTERN 04848910BLUE    70077MANAGER    COMMITTEE
PERSON7598CHICAGO 01795842BROWN   692150PERATOR   COMMITTEE
```

Figure 6-11. Input for Sample Program 6

```
PERSON2459EVANSTON25498100BROWN   70219MANAGER     CHAIRMAN
PERSON3313CICERO  06147200YELLOW  73290OPERATOR    V CHAIRMA
PERSON3796BATAVIA 85637900YELLOW  70123PROGRAMMER  MEMBER
PERSON4845CHICAGO 27513800YELLOW  71301ENGINEER    SECRETARY
PERSON4890BATAVIA 21357600PURPLE  70154ANALYST     MEMBER
PERSON5167LA GRANG42655400BLUE    69265SECRETARY   TREASURER
PERSON5968LYONS   68239400BROWN   74012SECRETARY   MEMBER
PERSON6594CHICAGO 59762400PURPLE  69152ENGINEER    COMMITTEE
PERSON6894LYONS   63773200PURPLE  71368CLERK-TYP   MEMBER
PERSON7156WESTERN 84891000BLUE    70077MANAGER     COMMITTEE
PERSON7598CHICAGO 79584200BROWN   69215OPERATOR    COMMITTEE
```

Figure 6-12. Output from Sample Program 6

Line 15

The ASSIGN clause for NAME-FILE lists two implementor names. The first, OFILE, is the data file for the user records; the second, MIP2, is the index file for the alternate key indexes. Within the COBOL program, only the COBOL file-name, NAME-FILE, is used; CYBER Record Manager processes both files in a manner invisible to the user. Outside the COBOL program, the files must be treated as two separate files, with different logical file names identical to the implementor-names.

Line 16 — Line 18

The ORGANIZATION, ACCESS, and NUMBER OF BLOCKS clauses are similar to those in sample program 5; specifying 20 home blocks is inefficient, but not illegal.

Line 19 — Line 22

One primary key and three alternate keys are defined for NAME-FILE. The SYMBOLIC KEY clause is used for the primary key; RECORD KEY or ACTUAL KEY would produce identical results. Duplicate alternate key values are not allowed for PERS-NO, but are allowed for COLOR and TEAM-TI. Since INDEXED is omitted for both COLOR and TEAM-TI, primary key values within an alternate key value entry for either of these keys are stored in the order in which they are encountered (first in, first out).

Line 46 — Line 47

The LABEL RECORDS clause is required. Although BLOCK CONTAINS specifies 640 characters, the actual data block size will be 1270 instead of 640 (next highest PRU multiple, minus 10 characters).

Line 65 — Line 82

NAME-FILE is opened for output and created in a fairly straightforward manner from the records on CRDFILE, using the WRITE INVALID KEY statement. Since no hashing routine has been provided, record location is determined by the system-provided hashing routine. NAME-FILE is closed after creation.

Line 88

CARD-FILE contains a single card, which in turn contains a single key value. The output shown in figure 6-12 is the result of using the key value 2459.

Line 90

The START statement specifies an alternate key, PERS-NO. The index file is positioned within the entries for PERS-NO at the subentries for the value 2459. Alternate key values are unique for PERS-NO. The START statement positions the file but does not read any records.

Line 95

Successive execution of READ NEXT AT END results in the successive reading of the records of NAME-FILE in the order in which the values of PERS-NO (subsequent to 2459) occur in the index file. Since alternate key values are maintained in order in the index file, and since duplicate alternate key values are not allowed for PERS-NO, each execution of READ NEXT AT END returns the record with the next highest value for PERS-NO. Sequential reading begins with the value established by the START statement, 2459. The program output shown in figure 6-12 confirms the order in which the records of NAME-FILE are read.

Indexed sequential files are mass storage files in which records are maintained in sorted order by primary key. Records can be accessed individually, by key, or successive records can be accessed sequentially; the advantages of random access are therefore combined with those of sorted sequential access. Random access is possible in indexed sequential files because CYBER Record Manager creates and maintains an index linking each record's key with its location in the file. Sorted sequential access is possible because CYBER Record Manager ensures that the physical order of records is always the same as their logical order by key. The index is maintained in index blocks (separate from the data blocks containing user records), which are a part of the file. Any of the record types discussed in section 2 can be used with indexed sequential files; the files cannot be labeled. COBOL indexed sequential files are implemented through CYBER Record Manager indexed sequential file organization.

An indexed sequential file always has one primary key and can optionally have up to 255 alternate keys. The primary key is defined through the RECORD/SYMBOLIC KEY clause. The primary key index remains part of the data file itself. The primary key is a character string of any length up to an installation defined limit; it may be described as COMPUTATIONAL-1, COMPUTATIONAL-2, or alphanumeric. If the file is a multiple index file, the primary key must be part of the record, and key values cannot be duplicated. If the file is not a multiple index file, the key need not be part of the record, and key values can be duplicated if the DUPLICATES option of the RECORD/SYMBOLIC KEY clause is specified.

Indexed sequential files are multiple index files when an index file is specified in the ASSIGN clause, and the ALTERNATE RECORD KEY clause is used. The alternate keys cannot begin at the same location as the primary key or each other, but they can overlap and vary in length. Alternate keys must be within the record. They can be duplicated in value only if the DUPLICATES option of the ALTERNATE RECORD KEY clause is selected.

The access mode under which an indexed sequential file can be processed can be either random or sequential. In addition, the file can be open for input, output, or I-O. Table 7-1 shows the relationship between access mode, open status, and input/output verbs allowed.

FILE STORAGE

An indexed sequential file on mass storage consists of a file statistics table, the data blocks that contain the user records, and the index.

TABLE 7-1. ACCESS MODE AND OPEN CONDITION COMBINATIONS, INDEXED SEQUENTIAL FILES

Statements	Random Access			Sequential Access		
	Open Input	Open Output	Open I-O	Open Input	Open Output	Open I-O
Create New File	N	Y	N	N	N	N
READ INVALID KEY	Y	N	Y	N	N	N
READ MAJOR INVALID KEY	Y*	N	Y*	N	N	N
WRITE INVALID KEY	N	Y	Y	N	N	N
REWRITE INVALID KEY	N	N	Y	N	N	N
REWRITE LAST INVALID KEY	N	N	Y*	N	N	Y*
DELETE INVALID KEY	N	N	Y	N	N	N
DELETE LAST INVALID KEY	N	N	Y*	N	N	Y*
READ KEY IS INVALID KEY	YM	N	YM	N	N	N
START KEY INVALID KEY	YM	N	YM	YM	N	YM
READ NEXT AT END	Y	N	Y	Y	N	Y
SKIP	Y	N	Y	Y	N	Y

Y = Allowed N = Not allowed YM = Allowed for multiple index files only

Y* = Allowed for non-multiple-index files only

FILE STATISTICS TABLE

The file statistics table (FSTT) is an internal table created and used by CYBER Record Manager. The table is written to the file after the file is closed, so that information about the file is available to CYBER Record Manager in subsequent runs using the file. The information that is maintained includes statistics about transactions on the file and file structure specifications such as size and padding of data and index blocks, key length, and maximum and minimum record size. The user should not attempt to change any of these specifications once the file has been created; COBOL allows some of these specifications to be omitted in subsequent programs, as described under Processing Existing Indexed Sequential Files.

DATA BLOCKS

In order to transfer and store data more efficiently, records in indexed sequential files are grouped into units called data blocks.

All data blocks are the same size. They contain both user records and keys pointing to those records. The records are stored near the beginning of the block, the keys are stored at the end of the block, and data padding lies between them. Figure 7-1 shows the appearance of the blocks.

The records at the beginning of the block and the key entries at the end of the block are maintained in the same sorted order.

Padding can be specified for data blocks through the DATA-PADDING clause (described below). New data blocks might be created when existing records are added at file update or replaced with larger records. If necessary, CYBER Record Manager relocates existing records and keys so that records are always in physical and logical order for sequential access.

INDEX BLOCKS

The primary key index for an indexed sequential file is subdivided into blocks called index blocks. (The alternate key indexes, if any, are not part of the data file but are kept on a separate index file.) Division into blocks enables more efficient index manipulation and record access. All index creation and manipulation are completely taken care of by COBOL and CYBER Record Manager; the only way in which the user becomes involved is in the optional specification of certain file structure parameters when the file is created.

When an indexed sequential file is created, COBOL determines an index block size that remains constant throughout the life of the file. This size can be indicated by the user directly through the INDEX-BLOCK CONTAINS clause. If this clause is omitted, COBOL calculates an appropriate block size based on other file structure specifications. Key entry size is also calculated, thus implicitly establishing the maximum number of key entries per index block.

A percentage of padding for index blocks can also be specified, through the INDEX-PADDING clause. When this clause is specified, index blocks are created with the specified percentage of unused space, so that later file updating can add new records without requiring index reorganization.

The simplest structure for an index sequential file involves only one index block, as shown in figure 7-2. The block contains one key entry for each data block in the file; each key entry indicates the lowest key value to be found in its associated data block. A record is located by finding the two key entries that bracket its key value; the record is written to or read from the data block associated with the lower of the two key values. For example, in figure 7-2, the record whose key is 5 is known to be in the second block because its key value occurs between key 4 and key 10.

If, during creation or updating, more than one index block becomes necessary (in other words, the number of data blocks exceeds the number of key entries per block), additional levels of indexing are established. A file with several levels of indexing is shown in figure 7-3. When several levels exist, the lowest level consists of a single block. Entries in this block point to the index blocks at the next higher level, and so forth. Entries in the highest level index block point to data blocks. The same bracketing principle is used for each level of indexing as for a file with one level of indexing. For example, in figure 7-3 the record whose key value is 27 is found in the fifth block by successively bracketing the key value 27 in higher levels of indexes, until the key entry pointing to the fifth data block is found.

CREATING INDEXED SEQUENTIAL FILES

COBOL clauses applicable to indexed sequential file creation are described below. Clauses and statements such as CLOSE, which must be specified for all files, and those that do not require further explanation, are omitted from the descriptive paragraphs. Clauses and statements that apply only to existing files are described under Processing Existing Indexed Sequential Files.

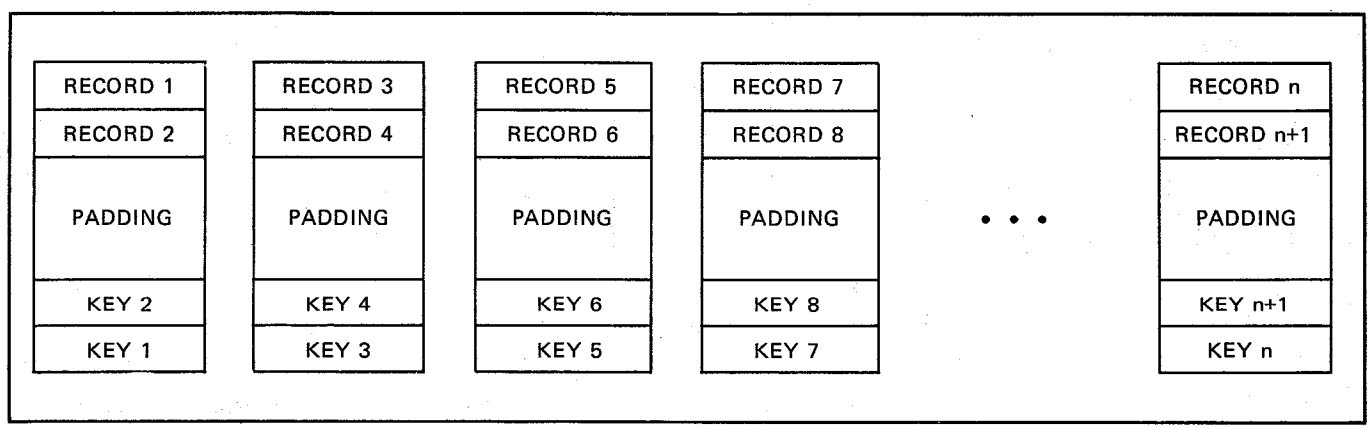

Figure 7-1. Padding of Data Blocks

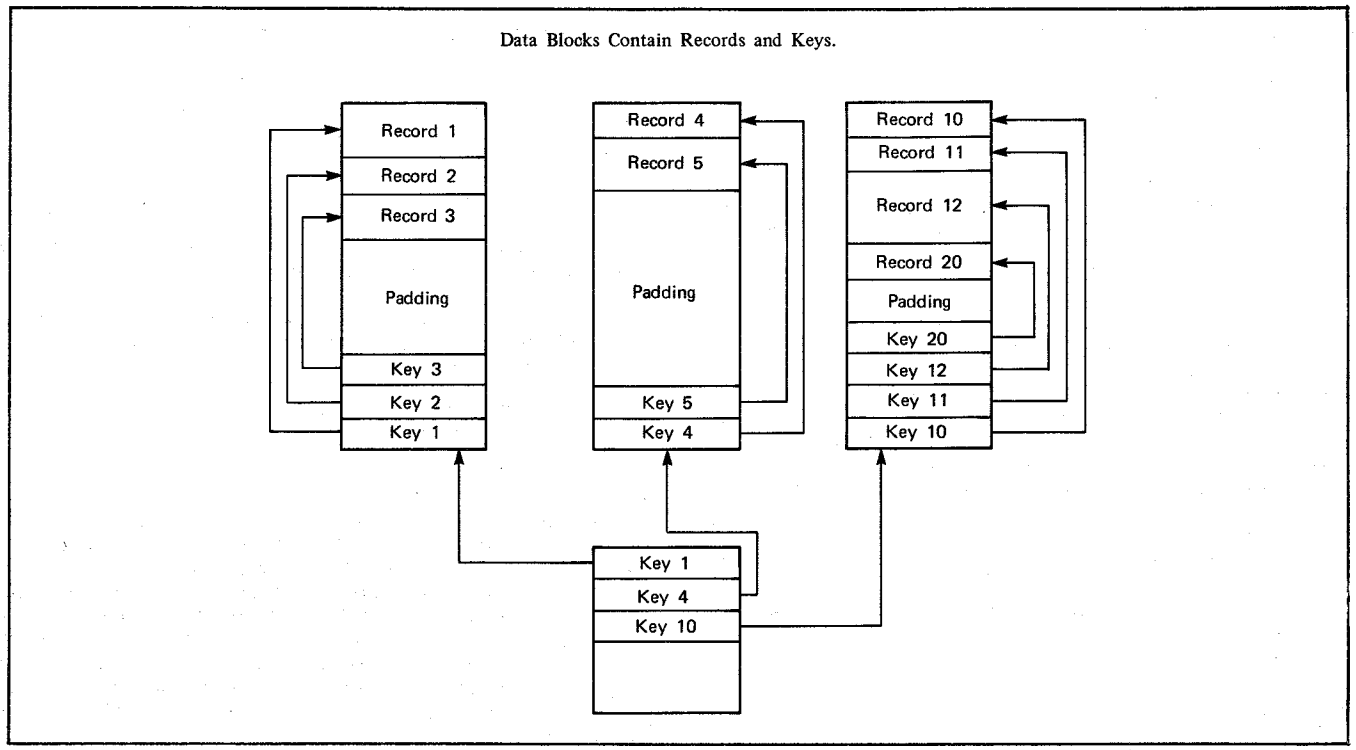

Figure 7-2. File with One Level of Index Block

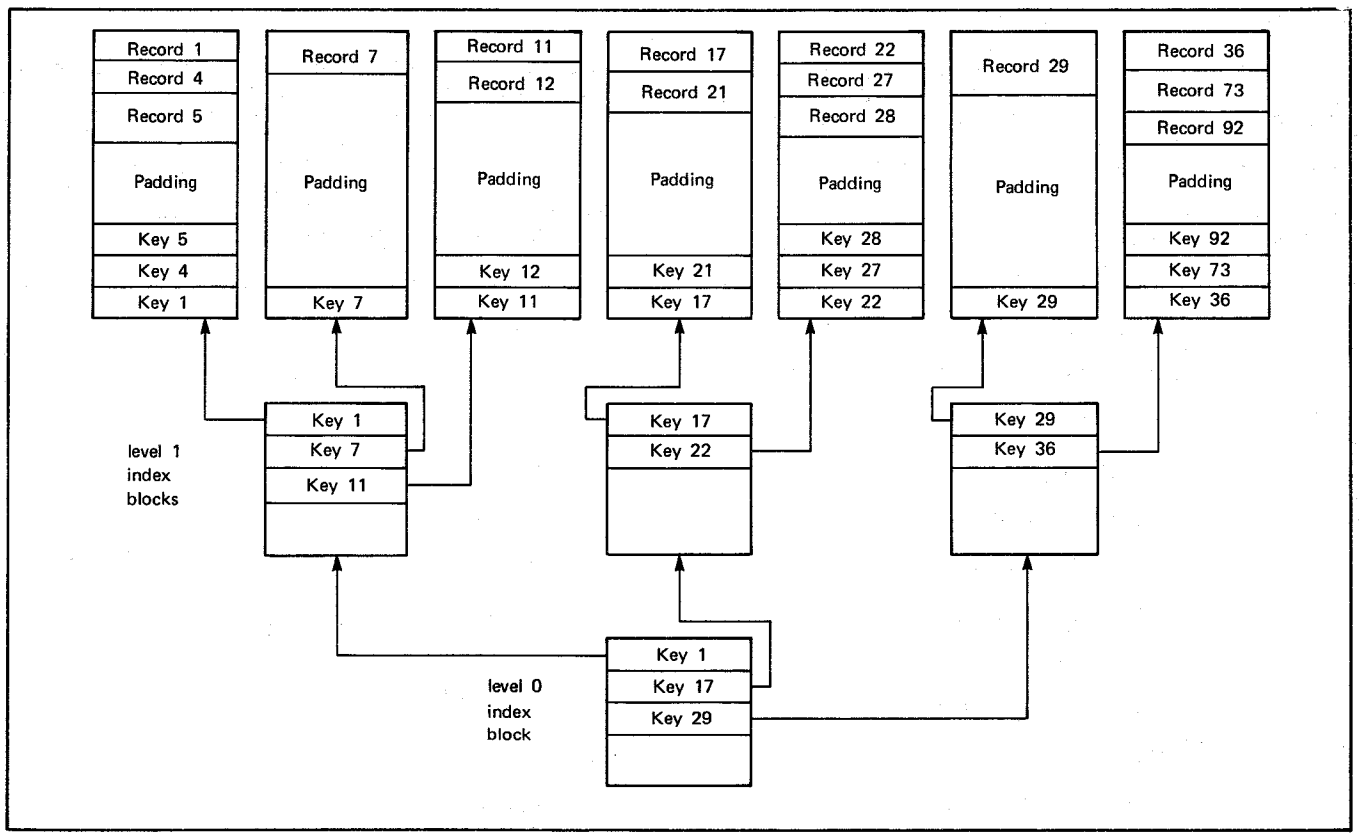

Figure 7-3. File with Two Levels of Index Block

ENVIRONMENT DIVISION

Environment Division clauses used to establish an indexed sequential file are shown in figure 7-4.

RESERVE ALTERNATE AREAS

RESERVE ALTERNATE AREAS can be used to increase the buffer area to accommodate additional index and data blocks:

```
RESERVE integer ALTERNATE AREAS
```

Buffer size information is not a fixed part of an indexed sequential file; consequently the RESERVE ALTERNATE AREAS clause can be changed after the file is created. Each additional area reserved is large enough for one index block and one data block. Up to 63 additional index and data blocks can be specified by the integer value. If the integer value is 1, one additional index block and one additional data block are assigned; an integer value of 5 adds five index blocks and five data blocks, and so on.

If RESERVE ALTERNATE AREAS is omitted or RESERVE NO ALTERNATE AREAS is specified, COBOL uses a formula based on other specifications provided to calculate the minimum area.

RECORD/SYMBOLIC KEY

A record or symbolic key must be specified for record access:

```
RECORD KEY IS data-name
```

or:

```
SYMBOLIC KEY IS data-name
```

The system does not distinguish between record and symbolic keys; all references to SYMBOLIC KEY are processed exactly the same as for RECORD KEY. The key need not be part of the data record unless the file is to be multiply indexed. It can be described as a COMPUTATIONAL-1 (integer) key:

```
77 INT-KEY PICTURE 9(6)
   COMPUTATIONAL-1.
```

or as a COMPUTATIONAL-2 (floating-point) key:

```
77 FP-KEY USAGE IS
   COMPUTATIONAL-2.
```

or as a character string:

```
77 CH-KEY PICTURE X(8).
```

The character string can be any length less than or equal to the installation-defined maximum.

For non-multiple-index files only, key values can be duplicated if the following format of the RECORD/SYMBOLIC KEY clause is specified:

```
RECORD KEY IS data-name
WITH DUPLICATES
```

When this format is used, records whose keys duplicate those of records already in the file can be written. The records with the same key value are all contiguous, and are referred to as a duplicate key group. The order of records within the group is the same as the order in which they were written. If DUPLICATES is not specified in the RECORD/SYMBOLIC KEY clause, and an attempt is made to write a record whose key duplicates that of an existing record, the INVALID KEY clause of the WRITE statement is executed, and the record is not written.

ALTERNATE RECORD KEY

The ALTERNATE RECORD KEY clause can be used only for multiple index files. An index file, in addition to the indexed sequential data file, must be defined through the ASSIGN clause, in which case the ALTERNATE RECORD KEY clause is required. When alternate keys are specified, records in the file can be accessed on a key other than the primary key specified in the RECORD or SYMBOLIC KEY clause. At least 1 alternate key must be defined for multiple index files,

SELECT . . . ASSIGN	Required
ORGANIZATION IS INDEXED SEQUENTIAL	Required
RESERVE ALTERNATE AREAS	Optional
FILE-LIMITS	Optional
ACCESS IS RANDOM	Required
RECORD/SYMBOLIC KEY	Required
ALTERNATE RECORD KEY	Required for multiple-index files
INDEX-LEVEL INDEX-BLOCK CONTAINS	One required
INDEX-PADDING	Optional
RECORD-BLOCK CONTAINS	Optional
DATA-PADDING	Optional

Figure 7-4. Environment Division Clauses for Creating Indexed Sequential Files

and as many as 255 can be specified. The alternate key specifications must immediately follow the primary key specification:

```
RECORD KEY IS data-name-1
ALTERNATE RECORD KEY IS data-name-2
ALTERNATE RECORD KEY IS data-name-3
.
.
.
```

The index file specified in the ASSIGN clause is used to hold alternate key indexes which are established and maintained automatically by CYBER Record Manager. One index is created for each alternate key field. Within the index, one entry is made for each alternate key value encountered as records are written. For each record with a given alternate key value, one primary key subentry is associated with the alternate key entry. The alternate key entries are kept in sorted order by CYBER Record Manager. Every time a record is added or deleted, the index file is updated to reflect all the alternate key values of the record in question. Index file structure is shown in figure 6-4.

If the DUPLICATES option of the ALTERNATE KEY clause is not specified, alternate key values must be unique for each record in the file. In this case, a duplicate alternate key value encountered on a write causes execution of the INVALID KEY clause. If DUPLICATES is specified, duplicate alternate key values are allowed. The order of primary key subentries within an alternate key entry depends on the presence or absence of the INDEXED option. If INDEXED is omitted, primary key values associated with a given alternate key value are maintained in the order in which they are written to the file (first in, first out). If INDEXED is specified, the primary keys are kept in sorted order by value. Performance in file updating is considerably enhanced when INDEXED is specified.

Alternate keys can overlap and differ in length, but none can begin in the same location as the primary key or any other alternate key. The alternate key fields defined at file creation are normally retained for the life of the file, but they can be changed or altered, at user option, through the IXGEN utility (see appendix C).

If the ALTERNATE RECORD KEY clause is omitted, the file does not have multiple index status and access can only be through the primary key.

Index Block Size Calculation

When an indexed sequential file is created, an index block size is calculated which remains constant throughout the life of the file. Index block size is not necessarily the same as data block size. The size of index blocks is important in the future efficient processing of the file. If the size is too small, index block splitting occurs too frequently, and a large number of index levels can become necessary. Each additional level of index requires an additional mass storage access for each record. Index blocks that are too large, on the other hand, require excessive amounts of central memory. Therefore a balance should be struck in selecting index block size. The ESTMATE utility, discussed in appendix C, can be used to calculate an optimum index block size.

The COBOL user is not required to specify index block size directly. If desired, however, the size can be specified through the clause:

```
INDEX-BLOCK CONTAINS
integer CHARACTERS
```

Any size between 1 and 327,670 characters may be specified. COBOL rounds up the value specified to the next highest multiple of PRU size (640 characters), minus 10 characters. If the INDEX-BLOCK CONTAINS clause is omitted, COBOL calculates an appropriate index block size from information provided by the clauses INDEX-LEVEL, INDEX-PADDING, and FILE-LIMITS, and from the key entry size as determined by the key length.

INDEX-LEVEL is specified as follows:

```
INDEX-LEVEL IS integer
```

The integer can be from 1 to 63. This specification is used in conjunction with the FILE-LIMITS clause; the default for FILE-LIMITS is 4094. An index block size is calculated such that, when the file contains the number of records indicated by FILE-LIMITS, it will have the number of index levels specified by the INDEX-LEVELS clause. Since this value is used only when the file is created, subsequent processing can result in the file actually having more index levels than originally specified. INDEX-LEVEL is ignored if INDEX-BLOCK CONTAINS is present, or if specified for an existing file.

FILE-LIMITS is specified as follows:

```
FILE-LIMITS IS integer
```

In addition to its use in index block size calculation, FILE-LIMITS restricts the number of records that the file can contain to the value specified by integer. This restriction is only valid for the duration of the program in which it is specified; FILE-LIMITS may be omitted, or specified with a different value, in any program.

The format of INDEX-PADDING is as follows:

```
INDEX-PADDING IS integer PERCENT
```

The integer, from 0 to 99, specifies the percentage of space which is to be left unused when an index block is created. INDEX-PADDING can only be specified once, when the file is created, but the value specified is valid throughout the life of the file. Unused space allows for the addition of records to the file without the necessity for index block splitting.

Data Block Size Calculation

Like index block size, data block size can be specified directly or indirectly. Direct specification is through the BLOCK CONTAINS clause (described under Data Division) or the RECORD-BLOCK CONTAINS clause in one of the following formats:

```
RECORD-BLOCK CONTAINS
integer RECORDS
```

or:

```
RECORD-BLOCK CONTAINS
integer CHARACTERS
```

If RECORDS is used, COBOL converts the value to characters based on the maximum record size. If CHARACTERS is used, the value specified is accepted. In both cases, block size is rounded up to the next multiple of PRU size (640 characters), less 10 characters. If both RECORD-BLOCK CONTAINS and BLOCK CONTAINS are specified, BLOCK CONTAINS takes precedence. If neither clause is specified, COBOL calculates a data block size based on average record length, key length, and the DATA-PADDING specification.

DATA-PADDING is specified as follows:

```
DATA-PADDING IS integer PERCENT
```

The integer, from 0 to 99, specifies the percentage of space which is to be left unused when a data block is created; the default value is 0 (no padding). DATA-PADDING can only be specified once, when the file is created, but the value specified remains valid throughout the life of the file. Unused space allows for the addition of records to the file without the necessity for data block splitting.

DATA DIVISION

Data Division clauses applicable to indexed sequential files are shown in figure 7-5.

BLOCK CONTAINS	Optional
LABEL RECORD IS OMITTED	Required

Figure 7-5. Data Division Clauses for Creating Indexed Sequential Files

BLOCK CONTAINS

BLOCK CONTAINS can be used to specify data block size if the RECORD-BLOCK clause is not used:

```
BLOCK CONTAINS integer RECORDS
```

or:

```
BLOCK CONTAINS integer CHARACTERS
```

If RECORDS is used, COBOL converts the number of records to characters and rounds the block size up to the next highest multiple of 640 characters less 10. If CHARACTERS is specified, the block size is rounded up in the same manner. If the clause is omitted and a RECORD-BLOCK CONTAINS clause is not used for the file, COBOL calculates data block size based on other file structure specifications.

LABEL RECORD

Indexed sequential files cannot have labels; the LABEL RECORD clause is required in the following format:

```
LABEL RECORD IS OMITTED
```

PROCEDURE DIVISION

The clauses used to create an indexed sequential file are shown in figure 7-6.

USE AFTER ERROR	Optional
OPEN OUTPUT	Required
WRITE INVALID KEY	Required
CLOSE	Required

Figure 7-6. Procedure Division Clauses for Creating Indexed Sequential Files

USE AFTER ERROR PROCEDURE can be used to define additional error processing. USE procedures are discussed in section 2.

OPEN OUTPUT is the only form of the OPEN statement that can be used for indexed sequential file creation:

```
OPEN OUTPUT file-name
```

WRITE INVALID KEY must be used to create records:

```
WRITE record-name INVALID KEY
```

INVALID KEY procedures are executed if file limits are exceeded, if key contents are invalid, or if the key duplicates that of an existing record, and DUPLICATES was not specified in the RECORD/SYMBOLIC KEY clause.

When an indexed sequential file is created, the records written to the file must be in sorted order by key value. (This restriction does not apply to records written on subsequent update runs.) The COBOL SORT facility, described in the COBOL Reference Manual, can be used to sort a file containing the records prior to creating the indexed sequential file.

PROCESSING EXISTING INDEXED SEQUENTIAL FILES

An existing indexed sequential file open for I-O can have records added, deleted, rewritten, or read. Access can be by alternate key (if alternate keys have been defined) or by primary key. An existing indexed sequential file open for INPUT can only be read.

Because of the file statistics table described above, it is not necessary in a program using an existing indexed sequential file to repeat clauses related to index and data block structure. Thus, the clauses INDEX-LEVEL, INDEX-BLOCK CONTAINS, INDEX-PADDING, RECORD-BLOCK CONTAINS, DATA-PADDING, and BLOCK CONTAINS can all be omitted; the values specified when the file was created will be used again. If they are included, however, they must specify the same values as when the file was created. Except for optional omission of these clauses, the Data and Environment Division clauses used when using an existing indexed sequential file are the same as those used when the file is created.

Procedure Division statements used for updating an indexed sequential file are shown in figure 7-7.

USE AFTER ERROR PROCEDURE can be used with I-O or with one or more file names to specify routines to be executed following an input/output error.

USE FOR DUPLICATE KEY can be specified to cause a duplicate primary key specification to be ignored and a specified procedure to be executed instead.

USE procedures are described in Section 2.

PRIMARY KEY ACCESS

In order to access a record by primary key, the INVALID KEY option must be included in the input/output statement. When this option is used, the contents of the key item defined in the RECORD/SYMBOLIC KEY clause are used to determine the location of the desired record. Access takes place in the

USE AFTER ERROR	Optional
USE FOR DUPLICATE KEY	Optional
OPEN INPUT OPEN I-O	One required
READ INVALID KEY	Optional
READ MAJOR INVALID KEY	Optional
WRITE INVALID KEY	Optional
REWRITE INVALID KEY	Optional
REWRITE LAST INVALID KEY	Optional
DELETE INVALID KEY	Optional
DELETE LAST INVALID KEY	Optional
READ KEY IS INVALID KEY	Optional; multiple index files only
START KEY INVALID KEY	Optional; multiple index files only
READ NEXT AT END	Optional
SKIP	Optional
CLOSE	Required

Figure 7-7. Procedure Division Statements for Indexed Sequential Files

same manner whether or not the file is a multiple index file. The INVALID KEY clause is executed when the key specified does not match the key of any record in the file (READ, REWRITE, and DELETE only), when writing the record would cause the FILE-LIMITS specified to be exceeded (WRITE only), and when the key provided duplicates the key of an existing record and DUPLICATES was not specified in the RECORD/SYMBOLIC KEY clause (WRITE only). Whenever an input/output error occurs, the special-register ERROR-CODE is set to the number of the CYBER Record Manager error that has occurred. These numbers are listed in the CYBER Record Manager Reference Manual. It is usually good practice for a program to check the contents of ERROR-CODE whenever the INVALID KEY clause is executed.

READ INVALID KEY locates a record according to the contents of the primary key item and returns it to the input record area.

WRITE INVALID KEY adds a new record to the file in the appropriate place in sequential order.

REWRITE INVALID KEY replaces an existing record with a new record with the same primary key. Following the rewrite, the new record is no longer available in the output record area.

DELETE INVALID KEY removes an existing record and its index entries from the file.

The format:

```
READ file-name MAJOR KEY
IS data-name INVALID KEY
imperative-statement
```

can be used for indexed sequential files that are not multiple index files to read a record by major key. The major key defined by data-name is a data item within the key item defined by the RECORD/SYMBOLIC KEY clause. The major key must be less than or equal to the full key in length, and must be a leading portion of the key item. When the READ MAJOR KEY statement is executed, the record returned is the first record in the file for which the contents of the major key portion of the key agree with the contents of data-name. Sequential reads can then be used to return all the records with the same major key value.

Duplicate Keys

If an indexed sequential file is not a multiple index file, duplicate primary key values are allowed when the DUPLICATES option is specified in the RECORD/SYMBOLIC KEY clause. A duplicate key is a key that is identical to the key of a record already in the file. If DUPLICATES is omitted, and a duplicate key is encountered on a write, the new record is not written to the file. Instead, the USE FOR DUPLICATE KEY procedure (if any) is executed, followed by the INVALID KEY clause. If DUPLICATES is specified, and a USE FOR DUPLICATE KEY procedure is provided, the USE procedure is executed, followed by the INVALID KEY clause. If no USE procedure is provided, the new record is written to the file. Because the records in the file are in sorted order by primary key, all records with the same primary key value are contiguous; their order within the group is the same as the order in which they were written.

When a READ INVALID KEY statement specifies a primary key value associated with a group of duplicates, the first record in the group is returned. This is the only record in the group that can be accessed randomly; the remaining records can only be accessed by READ NEXT AT END statements. After execution of READ NEXT AT END, the key item is set to the key of the record read; therefore, by checking the contents of this item after a read, the program can determine when the end of the group of duplicates has been passed.

A READ NEXT AT END statement may be followed by either a REWRITE or DELETE statement with the LAST option specified to rewrite or delete the record just read by the READ statement. The contents of the key item must not have been altered since the read. In this way, some or all of the records in a group of duplicates can be rewritten or deleted.

ALTERNATE KEY ACCESS

READ KEY IS INVALID KEY reads a multiple index file record by alternate key:

```
READ file-name KEY IS
data-name INVALID KEY
imperative-statement
```

The data-name specified as the object of the KEY IS option must be an item previously defined as an alternate key by the ALTERNATE RECORD KEY clause. When this statement is executed, the index file entry for the alternate key value that matches the contents of data-name is located. The record whose primary key value occurs first in the subentries for the alternate key value is returned to the input record area. If DUPLICATES is not specified in the ALTERNATE RECORD KEY clause, only one primary key value is associated with the alternate key value. If DUPLICATES is specified, the record returned by READ KEY IS INVALID KEY is the record whose primary key occurs first in the group of primary keys associated with the alternate key value. The data-name referenced in the KEY IS option can be a leading portion of an alternate key item, rather than the whole item.

START KEY INVALID KEY is used to position a multiple index file without reading a record:

```
START file-name KEY relational-operator
data-name INVALID KEY imperative-statement
```

The relational operator must be one of the following:

```
IS EQUAL TO
IS =
IS GREATER THAN
IS >
IS NOT LESS THAN
IS NOT <
```

The data-name can be a primary or alternate key item, or a leading portion of a key item.

Execution of the START statement establishes the key of reference by positioning the index file to the first alternate key value that meets the specified condition. The search begins either from the current key of reference or from the beginning of the index file if the key of reference has not yet been established. The key that satisfies the condition becomes the new key of reference. If the comparison is not satisfied by any alternate key value, the INVALID KEY clause is executed.

Successful execution of READ INVALID KEY or START establishes a key of reference for purposes of future access to the file. The key of reference is the primary or alternate key of the record read or located. Once the key of reference has been established, it can only be changed by execution of another START or READ INVALID KEY statement.

If the READ is by primary key (KEY IS is omitted), the new key of reference is the primary key value read. Subsequent sequential reads return records in their order in the data file; any index file positioning established by START or READ KEY IS INVALID KEY is lost. If the READ is by alternate key, the alternate key value is the new key of reference, and subsequent sequential reads return records in the order their keys occur in the index file entries.

The importance of the key of reference is that it determines the order in which records are read by subsequent READ NEXT AT END statements. READ NEXT AT END specifies sequential reading of records. If the key of reference is an alternate key, records are returned to the user program in the order in which their primary keys occur in the alternate key index. When the last record with a given alternate key value has been read, the special-register ERROR-CODE is set to 1000. No other indication is made that the end of the list for that alternate key value has been reached. The next time READ NEXT AT END is executed, if the key of reference has not been changed, the first primary key in the list for the next alternate key value is used to read a record from the data file.

If READ NEXT AT END is executed before a key of reference has been established, records in the data file are read according to the physical order of records in the data file; the alternate key index is not referenced.

READ ONLY PROCESSING

When an indexed sequential file is opened for INPUT, only the statements READ INVALID KEY and READ NEXT AT END can be used; if the file is a multiple index file, READ KEY IS INVALID KEY and START can also be used. In other words, any of the information in the file can be accessed, but no changes can be made to the file itself. This kind of processing is called read-only processing; it is particularly useful when the file is to be protected from accidental alteration.

For a non-multiple-index file, if the access mode is sequential, only READ NEXT AT END can be executed. In this case, records are read in the physical order in which they occur on the file (that is, in sorted order by key).

For a multiple index file, when the access mode is sequential, the START statement can be executed as well as READ NEXT AT END. The START statement positions the file at the first primary key value or alternate key value satisfying the specified comparison. Subsequent execution of READ NEXT AT END returns records in an order that depends on whether the key specified by START was a primary or alternate key. If primary, records are returned in primary key order (the same as for a non-multiple-index file). If alternate, records are returned in order by the alternate key; within each alternate key value, records are returned in the order written, or, if INDEXED was specified in the ALTERNATE KEY clause, in order by primary key.

SAMPLE PROGRAM 7: USING INDEXED SEQUENTIAL FILES

Sample program 7, shown in figure 7-8, creates an indexed sequential file (RANFILE), updates the file by accessing it randomly, and reads the file by accessing it sequentially. The purpose of the program is to create a file containing all the odd prime numbers less than a specified value (500 in this case) through a modified Eratosthenes' sieve method.

Line 15 — Line 24

The FILE-CONTROL clauses for RANFILE provide specific values for file structure parameters. ORGANIZATION IS INDEXED and ACCESS IS RANDOM are required. SYMBOLIC KEY is used; RECORD KEY could have been used, with the same results. RESERVE ALTERNATE AREAS is optional; specifying 10 areas results in allocation of a central memory buffer large enough for 10 index blocks and 10 data blocks. The INDEX-BLOCK CONTAINS and RECORD-BLOCK CONTAINS clauses specify the same block size, 10230, which will not be rounded up because it is exactly 16 PRUs minus one word. Padding is also specified for both kinds of blocks. The values selected in this program are not presented as recommendations for efficient processing, but only illustrate these clauses.

Line 39

MAX is the upper limit of the list of primes to be computed.

Line 44

The method used for generation of primes is to write all the odd numbers to RANFILE in two batches, and then to eliminate those that are discovered to be composite (non-prime).

Line 46 — Line 53

RANFILE is opened for output, as required for indexed sequential file creation. The first stage of prime number generation is to write to RANFILE all the odd numbers that are 3 greater than a multiple of 4. After this is done, the file is closed.

```
00001            IDENTIFICATION DIVISION.
00002
00003            PROGRAM-ID. ERATOSTHENES-SIEVE.
00004
00005            ENVIRONMENT DIVISION.
00006
00007            CONFIGURATION SECTION.
00008            SOURCE-COMPUTER. CYBER.
00009            OBJECT-COMPUTER. CYBER.
00010
00011            SPECIAL-NAMES.
00012                CONSOLE IS TUBE.
00013
00014            INPUT-OUTPUT SECTION.
00015            FILE-CONTROL.
00016            SELECT RANFILE ASSIGN TO DISC01
00017                    RESERVE 10 ALTERNATE AREAS
00018                    ORGANIZATION IS INDEXED
00019                    ACCESS MODE IS RANDOM
00020                    SYMBOLIC KEY IS KEY-HOLE
00021                    INDEX-BLOCK CONTAINS 10230 CHARACTERS
00022                    RECORD-BLOCK CONTAINS 10230 CHARACTERS
00023                    INDEX-PADDING IS 15 PERCENT
00024                    DATA-PADDING IS 50 PERCENT.
00025
00026            DATA DIVISION.
00027
00028            FILE SECTION.
00029
00030            FD RANFILE
00031                LABEL RECORDS OMITTED
00032                DATA RECORD IS REC.
00033            01 REC.
00034                02  KEY-HOLE PIC 9(10).
00035                02 PRIME-MARK PIC X.
00036                02 BODY       PIC X(69).
00037
00038            WORKING-STORAGE SECTION.
00039                77 MAX          PIC 9(5) VALUE 500.
00040                77  EDIT-FIELD PIC Z(9)9.
00041                77 INCREMENT  PIC 9(10).
00042                77 KEY-STORE  PIC 9(10).
00043
00044            PROCEDURE DIVISION.
00045
00046            CREATION.
00047                OPEN OUTPUT RANFILE.
00048                MOVE ALL ≠ INITIAL RECORD ≠ TO BODY.
00049                MOVE SPACE TO PRIME-MARK.
00050                PERFORM RECORD-CREATION
00051                    VARYING KEY-HOLE FROM 3 BY 4
00052                      UNTIL KEY-HOLE GREATER THAN MAX.
00053                CLOSE RANFILE.
00054                GO TO UPDATE-PORTION.
00055
00056            RECORD-CREATION.
00057                WRITE REC INVALID KEY DISPLAY ≠ FAULT 1≠ STOP RUN.
00058                MOVE KEY-HOLE TO INCREMENT.
00059
00060            UPDATE-PORTION.
00061                OPEN I-O RANFILE.
00062                SUBTRACT 2 FROM INCREMENT.
00063                MOVE ALL ≠ INSERTION RECORD ≠ TO BODY.
00064                PERFORM INSERTION
00065                    VARYING KEY-HOLE FROM INCREMENT BY -4
00066                    UNTIL KEY-HOLE < 3.
00067                GO TO START-SIEVE.
00068
```

Figure 7-8. Sample Program 7: Using Indexed Sequential Files (Sheet 1 of 2)

```
00069           INSERTION.
00070               WRITE REC INVALID KEY DISPLAY ≠  FAULT 2≠.
00071
00072           START-SIEVE.
00073               MOVE 3 TO KEY-HOLE.
00074               READ RANFILE INVALID KEY DISPLAY ≠ FAULT 3≠ STOP RUN.
00075
00076           SEARCH-FOR-BASE.
00077               READ RANFILE AT END GO TO SIEVE-COMPLETE.
00078               IF PRIME-MARK NOT = ≠*≠ GO TO BASE-FOUND.
00079               GO TO SEARCH-FOR-BASE.
00080
00081           BASE-FOUND.
00082               MOVE ≠*≠ TO PRIME-MARK.
00083               REWRITE REC INVALID KEY DISPLAY ≠ FAULT 4≠ STOP RUN.
00084               MULTIPLY KEY-HOLE BY 2 GIVING INCREMENT.
00085               MOVE KEY-HOLE  TO KEY-STORE.
00086
00087           SIEVE.
00088               ADD INCREMENT TO KEY-STORE.
00089               IF KEY-STORE GREATER THAN MAX GO TO START-SIEVE.
00090               MOVE KEY-STORE TO KEY-HOLE.
00091               DELETE RECORD FROM RANFILE INVALID KEY PERFORM NOTHING.
00092               GO TO SIEVE.
00093
00094           SIEVE-COMPLETE.
00095               CLOSE RANFILE.
00096               OPEN INPUT RANFILE.
00097               DISPLAY ≠ P R I M E  N U M B E R S  ≠.
00098
00099           SEQ-READ-LOOP.
00100               READ RANFILE AT END GO  TO WRAP-UP.
00101               MOVE KEY-HOLE TO EDIT-FIELD.
00102               DISPLAY ≠          ≠ EDIT-FIELD.
00103               GO TO SEQ-READ-LOOP.
00104
00105           WRAP-UP.
00106               CLOSE RANFILE.   STOP RUN.
00107
00108           NOTHING.
```

Figure 7-8. Sample Program 7: Using Indexed Sequential Files (Sheet 2 of 2)

Line 57

The WRITE INVALID KEY statement writes a record to RANFILE, using one of the generated odd numbers as the symbolic key. The keys are thus written to the file in sorted order by key, as required for indexed sequential files.

Line 60 — Line 67

The remainder of the odd numbers less than 500 (except for 1) are inserted in their proper places in the file. Record insertion need not be in sorted order; in fact, these records are inserted in reverse order.

Line 69 — Line 92

The sieving of the odd primes takes place from line 69 to line 92. The data item PRIME-MARK is used to flag composite numbers. The file is first read sequentially until a prime number is found, then the number found is used as a base to eliminate subsequent numbers that are multiples of the base number. The records having composite numbers as keys are deleted from the file.

Line 94 — Line 103

After all the nonprime numbers have been deleted from RANFILE, it is closed and reopened for INPUT. All the records in the file are read sequentially and displayed. The output for this program is shown in figure 7-9.

SAMPLE PROGRAM 8: MULTIPLE INDEX INDEXED SEQUENTIAL FILES

Sample program 8, shown in figure 7-10, illustrates the use of multiple index indexed sequential files. The program creates a multiple index file, NAME-FILE, closes it, and reopens it for read-only processing. The original records, as well as query records to read NAME-FILE, are on the file INPUT. The input for the program is shown in figure 7-11, and the output in figure 7-12.

Line 15

The ASSIGN clause for NAME-FILE specifies implementor-names for the data file, OFILE, as well as the index file, MIP2. Processing of MIP2 within this program is transparent

to the user, but the file must be preserved by the user through control statements subsequent to the COBOL control statement.

Line 16 — Line 18

The clauses ORGANIZATION IS INDEXED SEQUENTIAL and ACCESS IS RANDOM are required. The INDEX-BLOCK CONTAINS clause is optional; if it were omitted, INDEX-LEVEL would be required, so that COBOL could calculate index block size. Since the INDEX-PADDING and DATA-PADDING clauses are omitted, no padding is allowed for when new blocks are created.

Line 19 — Line 22

One primary key, KEY-1, and three alternate keys, PERS-NO, COLOR, and TEAM-TI are defined. DUPLICATES is specified for COLOR and TEAM-TI. For these keys, more than one record can have the same alternate key value; the primary key subentries within the alternate key value entries in the index file are arranged in the order records are written.

Line 46 — Line 47

The LABEL RECORDS clause is required. The BLOCK CONTAINS clause is optional, and when included specifies the size of data blocks.

Line 63 — Line 80

Creation of NAME-FILE begins when it is opened for OUTPUT. The records of CRDFILE (which is actually the next section on the INPUT file) are read and transferred directly to PERS-REC with no modification. The WRITE INVALID KEY statement is then used to write the records to NAME-FILE. At the end of creation, NAME-FILE and CRDFILE are closed.

Line 85 — Line 99

NAME-FILE is reopened for INPUT in order to read selected records. The file CARD-FILE consists of a single record, which contains a value for the alternate key PERS-NO. This value is used to position NAME-FILE by means of the START statement (lines 88 through 90). For this execution of the program, the value used is 2459. Since PERS-NO is an alternate key rather than a primary key, the index file, rather than the data file, is searched for the specified value of PERS-NO. When the index entry for this value is located, so is the corresponding primary key value (which is unique for PERS-NO). When READ NEXT AT END (line 93) is executed, therefore, the primary key value is used to retrieve a record from the data file. Subsequent iterations of READ NEXT AT END read records from NAME-FILE in the order in which the alternate key values for PERS-NO occur in the index file (as shown in figure 7-12). When the last entry for PERS-NO in the index file is passed, the AT END clause is executed, and the files are closed (line 98).

```
PRIME NUMBERS
              3
              5
              7
              9
             11
             13
             17
             19
             23
             29
             31
             37
             41
             43
             47
             53
             59
             61
             67
             71
             73
             79
             83
              .
              .
              .
            433
            439
            443
            449
            457
            461
            463
            467
            479
            487
            491
            499
```

Figure 7-9. Output from Program 7

```
00001            IDENTIFICATION DIVISION.
00002
00003            PROGRAM-ID. INDEX-SEQ-MIP.
00004
00005            ENVIRONMENT DIVISION.
00006
00007            CONFIGURATION SECTION.
00008            SOURCE-COMPUTER. CYBER.
00009            OBJECT-COMPUTER. CYBER.
00010
00011            INPUT-OUTPUT SECTION.
00012            FILE-CONTROL.
00013                SELECT CRDFILE ASSIGN TO INPUT.
00014                SELECT CARD-FILE ASSIGN TO INPUT.
00015                SELECT NAME-FILE ASSIGN OFILE MIP2
00016                ORGANIZATION IS INDEXED SEQUENTIAL
00017                ACCESS IS RANDOM
00018                INDEX-BLOCK CONTAINS 5110 CHARACTERS
00019                SYMBOLIC KEY IS KEY-1
00020                ALTERNATE RECORD KEY IS PERS-NO
00021                ALTERNATE RECORD KEY IS COLOR  DUPLICATES
00022                ALTERNATE RECORD KEY IS TEAM-TI  DUPLICATES.
00023
00024            DATA DIVISION.
00025
00026            FILE SECTION.
00027
00028            FD CRDFILE
00029                LABEL RECORDS ARE OMITTED
00030                DATA RECORDS ARE CARD-REC.
00031            01  CARD-REC.
00032                02  IN-KEY1 PIC X(18).
00033                02  IN-AMT  PIC 9(8).
00034                02  IN-COLOR PIC X(8).
00035                02  I-DATE-A PIC 9(5).
00036                02  IN-RANK1 PIC X(20).
00037
00038            FD  CARD-FILE
00039                LABEL RECORDS ARE OMITTED
00040                DATA RECORDS ARE CARD-RECORD.
00041            01  CARD-RECORD.
00042                02  INPERS-NO  PIC 9(4).
00043                02  FILLER     PIC X(76).
00044
00045            FD  NAME-FILE
00046                LABEL RECORDS ARE OMITTED
00047                BLOCK CONTAINS 5110 CHARACTERS
00048                DATA RECORDS ARE PERS-REC.
00049            01  PERS-REC.
00050                02  KEY-1.
00051                    03  FILLER  PIC X(6).
00052                    03  PERS-NO PIC 9(4).
00053                    03  CITY    PIC X(8).
00054                02  AMT         PIC 9(6)V99.
00055                02  COLOR       PIC X(8).
00056                02  DATE-A      PIC 9(5).
00057                02  RANK.
00058                    03  TITLEA  PIC X(11).
00059                    03  TEAM-TI  PIC X(9).
00060
00061            PROCEDURE DIVISION.
00062
00063            CREATE-RANDOM-FILE.
00064                OPEN INPUT CRDFILE
00065                    OUTPUT NAME-FILE.
00066
```

Figure 7-10. Sample Program 8: Multiple Index Indexed Sequential Files (Sheet 1 of 2)

Page 7-13 is missing from the original CDC manual.

Page 7-14 is missing from the original CDC manual.

ACTUAL KEY FILE ORGANIZATION 8

An actual key file is a mass storage file in which a primary key associated with each record identifies directly the actual location of the record on the file. This key, called the actual key, is a bit string whose contents are system-oriented rather than data-oriented. No primary key index or hashing routine is needed or used for an actual key file.

Any of the record types described in section 2 can be used with actual key files. Labels are not supported.

Logically, an actual key file can be regarded as a series of record slots, each with an associated ordinal. Record slots are in order according to their ordinals. (Although relative files can be similarly described, their structure and processing differ from actual key files.) Once a slot is established by writing a record to it, it remains in existence for the life of the file. If a record is deleted, its slot becomes available for a new record. In the lifetime of a file, it is possible that several unrelated records could successively occupy the same slot.

The actual key of a record is the ordinal of the slot the record is written to. A key item for random access is defined through the clause:

ACTUAL KEY IS data-name

The contents of the key item are checked on every random access to the file. On a write, if the value of the key item is 0, CYBER Record Manager writes the record to an unoccupied record slot, and sets the key item to the ordinal of that slot, thus providing the user with the actual key of the record. Thereafter, the user accesses the record randomly by means of the system-assigned actual key value.

The key item defined by the ACTUAL KEY clause does not have to be contained within the record, unless the file is a multiple index file.

Because records in actual key files are automatically in order by slot ordinal, access to these files can be either random or sequential.

An actual key, being an internally defined entity, has no logical connection with the record it identifies. This usage of keys differs from that in direct and indexed sequential file organizations. In these file organizations, keys are provided by the user; typically, they contain information that has external meaning, such as employee name or inventory part number. Actual keys, on the other hand, have no purpose other than unique identification and location of records. Alternate keys can be defined for actual key files, however, providing the ability to access the file by externally meaningful keys.

Physically, an actual key file is divided into a series of blocks called data blocks. A fixed number of record slots is associated with every block in the file. Each block contains a descriptor for each slot in the block, whether the slot is full or empty. The block also contains the records corresponding to each slot, unless one or more of these records is too large to fit in the block. In this case, the overflow records are written to other blocks, but their descriptors remain in the original block. The number of data blocks in a file is restricted only by the optional FILE-LIMITS clause. Mass storage is not allocated in advance; when records are written, new blocks are added to the file as necessary. Blocks containing overflow records are not different structurally or logically from other blocks in the file.

Actual key files provide relatively fast random access of records, since no time is consumed by index block searches or key hashing. Less mass storage space is used than for indexed sequential files, since no index blocks exist; if an actual key file contains a very large number of unoccupied record slots, however, mass storage use is less efficient. The principal disadvantage of actual key files is the necessity for the user to keep track of the generated actual key values. As the key has no logical connection with the contents of the record, it is an additional piece of arbitrary information that must be maintained. The user normally establishes some mechanism, such as a second file, whereby a record's actual key is linked with some more meaningful piece of data.

In particular, actual key files are especially suitable as multiple index files. In a typical application, actual key values are assigned, and alternate key indexes built, as the original records are written. Subsequent random access to the file is only through alternate key values. Since each access by alternate key requires a read of the data file by primary key, and since actual key access is the fastest form of primary key access, an alternate key read of an actual key file is more efficient than an alternate key read under any other file organization.

The access mode under which an actual key file can be processed can be either random or sequential. In addition, the file can be open for INPUT, OUTPUT, or I-O. The Procedure Division statements that can be used with various combinations of access mode and open status are shown in table 8-1 and described in more detail below.

FILE STATISTICS TABLE

The first physical part of an actual key file on mass storage is an internal table called the file statistics table (FSTT). This table is created and maintained by CYBER Record Manager. The table is preserved on the file after the file is closed, so that information about the file is available to CYBER Record Manager in subsequent runs using the file. The information that is maintained includes statistics about transactions on the file and file structure specifications such as key length, maximum and minimum record length, block size, and number of records per block. The user should not attempt to change any of these specifications once the file has been created; COBOL allows some of these specifications to be omitted in subsequent programs, as described under Processing Existing Actual Key Files.

CREATING ACTUAL KEY FILES

Actual key file creation involves the clauses described below. Clauses and statements such as CLOSE, which must be specified for all files, and those that do not require further explanation, are omitted from the descriptive paragraphs. Clauses and statements that apply only to existing files are described under Processing Existing Actual Key Files.

TABLE 8-1. ACCESS MODE AND OPEN CONDITION COMBINATIONS, ACTUAL KEY FILES

Statements	Random Access			Sequential Access		
	Open Input	Open Output	Open I-O	Open Input	Open Output	Open I-O
Create New File	N	Y	N	N	N	N
SKIP	Y	N	Y	Y	N	Y
START KEY INVALID KEY	YM	N	YM	YM	N	YM
READ NEXT AT END	Y	N	Y	Y	N	Y
READ INVALID KEY	Y	N	Y	N	N	N
READ KEY IS INVALID KEY	YM	N	YM	N	N	N
WRITE INVALID KEY	N	Y	Y	N	N	N
REWRITE INVALID KEY	N	N	Y	N	N	Y
DELETE INVALID KEY	N	N	Y	N	N	Y
Y = Allowed	N = Not allowed			YM = Allowed multiple index only		

ENVIRONMENT DIVISION

The Environment Division clauses involved in creating an actual key file are shown in figure 8-1.

SELECT . . . ASSIGN	Required
ORGANIZATION IS ACTUAL KEY	Required
FILE-LIMITS	Optional
ACCESS IS RANDOM	Required
ACTUAL KEY	Required
ALTERNATE RECORD KEY	Required for multiple-index files

Figure 8-1. Environment Division for Creating Actual Key File

FILE-LIMITS

The FILE-LIMITS clause is optional and establishes the maximum number of records that can exist at any time:

```
FILE-LIMITS IS integer
```

FILE-LIMITS cannot be used to restrict processing to a portion of a file. Only the records in the file at any given time are affected by the FILE-LIMITS clause; records written to the file and subsequently deleted are not included in the total.

ACTUAL KEY

The primary key for an actual key file must be specified through the clause:

```
ACTUAL KEY IS data-name
```

The key item must be within the data record if the file is a multiple index file, but does not have to be otherwise. The key must be a COMPUTATIONAL-1 item with a PICTURE from 1 to 8 characters.

ALTERNATE RECORD KEY

The ALTERNATE RECORD KEY clause can be used only if an index file, in addition to the actual key data file, is defined through the ASSIGN clause, and then it is required. When alternate keys are specified, records in the file can be accessed on a key other than the primary key specified in the ACTUAL KEY clause. At least 1 alternate key must be defined for multiple index files, and as many as 255 can be specified. The alternate key specifications must immediately follow the primary key specification:

```
ACTUAL KEY IS data-name-1
ALTERNATE RECORD KEY IS data-name-2
ALTERNATE RECORD KEY IS data-name-3
.
.
.
```

The index file specified in the ASSIGN clause is used to hold alternate key indexes which are established and maintained automatically by CYBER Record Manager for multiple-index files. One index is created for each alternate key field. Within the index, one entry is made for each alternate key value encountered as records are written. For each record

with a given alternate key value, one primary key subentry is associated with the alternate key entry. The alternate key entries are kept in sorted order by CYBER Record Manager. Every time a record is added or deleted, the index file is updated to reflect all the alternate key values of the record in question. Index file structure is shown in figure 6-4.

If the DUPLICATES option of the ALTERNATE KEY clause is not specified, alternate key values must be different for each record in the file. In this case, a duplicate value encountered on a write causes execution of the INVALID KEY clause. If DUPLICATES is specified, duplicate values are allowed. The order of primary key values within a set of duplicates depends on the presence or absence of the INDEXED option. If INDEXED is omitted, primary key values associated with a given alternate key value are maintained in the order in which they are written to the file (first in, first out). If INDEXED is specified, the primary keys are kept in the same order as primary keys in an indexed sequential file (section 7). Performance in file updating is considerably enhanced when INDEXED is specified.

Alternate keys can overlap and differ in length, but none can begin in the same location as the primary key or any other alternate key. The alternate keys defined at file creation are normally retained for the life of the file, but they can be changed or altered, at user option, by using the IXGEN utility (see appendix C).

If the ALTERNATE RECORD KEY clause is omitted, the file does not have multiple index status and access can only be through the primary key.

DATA DIVISION

Data Division clauses applicable to actual key files are shown in figure 8-2.

BLOCK CONTAINS	Optional
LABEL RECORD IS OMITTED	Required

Figure 8-2. Data Division for File Creation

BLOCK CONTAINS

The BLOCK CONTAINS clause for actual key files is used in the following format:

```
BLOCK CONTAINS integer RECORDS
```

COBOL uses the value specified both to establish the fixed number of record slots per data block, and to calculate the data block size, based on average record length. If the BLOCK CONTAINS clause is omitted, the number of records per block is determined by installation default (eight records per block in the release system).

LABEL RECORD

Labels are not supported on actual key files; the LABEL RECORD clause is required in the following format:

```
LABEL RECORD IS OMITTED
```

PROCEDURE DIVISION

Procedure Division statements applicable to actual key file creation are shown in figure 8-3.

USE AFTER ERROR	Optional
OPEN OUTPUT	Required
WRITE INVALID KEY	Required
CLOSE	Required

Figure 8-3. Procedure Division for File Creation

USE AFTER ERROR PROCEDURE can be used to define additional processing to be executed following an error. USE procedures are discussed in section 2.

OPEN OUTPUT is the only form of the OPEN statement that can be used for actual key file creation:

```
OPEN OUTPUT file-name
```

WRITE INVALID KEY must be used to create records:

```
WRITE record-name INVALID KEY
imperative-statement
```

During file creation, the key item is reset to 0 before each execution of the WRITE INVALID KEY statement, and the system generates successive actual key values and returns them to the key item. in sequential order during file creation. The INVALID KEY clause is executed only when file limits are exceeded.

PROCESSING EXISTING ACTUAL KEY FILES

An existing actual key file open for I-O can have records added, deleted, rewritten, or read. Access can be by alternate key (if alternate keys have been defined), by primary key, or sequentially. An existing actual key file open for INPUT can only be read.

Because of the file statistics table described above, it is not necessary to repeat the BLOCK CONTAINS clause in a program using an existing actual key file. If it is included, however, it must specify the same value as that with which the file was created. Except for optional omission of this clause, the Data Division and Environment Division clauses used when updating an actual key file are the same as those used when the file is created.

Procedure Division statements used with an existing actual key file are shown in figure 8-4.

USE AFTER ERROR	Optional
OPEN INPUT OPEN I-O	One required
READ INVALID KEY	Optional
WRITE INVALID KEY	Optional
REWRITE INVALID KEY	Optional
DELETE INVALID KEY	Optional
READ KEY IS INVALID KEY	Optional; multiple index files only
START KEY INVALID KEY	Optional; multiple index files only
READ NEXT AT END	Optional
SKIP	Optional
CLOSE	Required

Figure 8-4. Procedure Division for Existing Actual Key Files

USE AFTER ERROR PROCEDURE can be used with I-O or with one or more file names to specify routines to be executed following an input/output error.

PRIMARY KEY ACCESS

In order to access a record in an actual key file by primary key, the INVALID KEY option must be included in the input/output statement. When this option is used, the contents of the key item defined in the ACTUAL KEY clause are used to determine the location of the desired record. Access takes place in the same manner whether or not the file is a multiple index file. The INVALID KEY clause is executed when the key specified does not match the key of any record in the file (READ, REWRITE, and DELETE only), when writing the record would cause the FILE-LIMITS specification to be exceeded (WRITE only), when the key provided duplicates the key of an existing record (WRITE only), and when the key is not in a valid format for the file (any statement). Whenever an input/output error occurs, the special-register ERROR-CODE is set to the number of the CYBER Record Manager error that has occurred. These numbers are listed in the Record Manager Reference Manual. It is advisable for a program to check the contents of ERROR-CODE whenever the INVALID KEY clause is executed.

READ INVALID KEY locates a record according to the contents of the primary key item and returns it to the input record area. The primary key item must be set to the slot ordinal of the desired record before the READ statement is executed.

WRITE INVALID KEY adds a new record to the file. If the value of the primary key item is 0, CYBER Record Manager writes the record to any available vacant slot, creating a new data block if necessary, and returns the ordinal of the slot selected to the key item as the actual key. If any records have been deleted from the file, the location selected for the new record is unpredictable, as is its consequent relative position among the other records in the file.

REWRITE INVALID KEY replaces an existing record with a new record with the same primary key. The primary key item must be set to the slot ordinal of the record to be replaced. Following the rewrite, the new record is no longer available in the output record area.

DELETE INVALID KEY removes an existing record from the file and frees the space it occupied for records written subsequently. The primary key item must be set to the slot ordinal of the record to be deleted.

ALTERNATE KEY ACCESS

READ KEY IS INVALID KEY reads a multiple index file record by alternate key:

```
READ file-name KEY IS data-name
INVALID KEY imperative statement
```

The data-name specified as the object of the KEY IS option must be an item previously defined as an alternate key by the ALTERNATE KEY clause. When this statement is executed, the index file entry for the alternate key value that matches the contents of data-name is located. The record whose primary key value occurs first in the subentries for the alternate key value is returned to the input record area. If DUPLICATES is not specified in the ALTERNATE RECORD KEY clause, only one primary key value will be associated with the alternate key value. If DUPLICATES is specified, the record returned by READ KEY IS INVALID KEY is the record whose primary key occurs first in the group of primary keys associated with the alternate key value. The data-name referenced in the KEY IS option can be a leading portion of an alternate key item, rather than the whole item.

START KEY INVALID KEY is used to position a multiple index file without reading a record:

```
START file-name KEY
relational-operator data-name
INVALID KEY imperative-statement
```

The relational-operator must be one of the following:

```
IS EQUAL TO
IS =
IS GREATER THAN
IS >
IS NOT LESS THAN
IS NOT <
```

The data-name must be an alternate key item, or the leading portion of an alternate key item.

Execution of the START statement results in positioning of the index file to the first alternate key value that meets the specified condition. The search begins from the current key of reference or from the beginning of the index file if no key of reference has been established yet. The key that satisfies the condition becomes the new key of reference. If the comparison is not satisfied by any alternate key value, the INVALID KEY clause is executed.

Successful execution of READ INVALID KEY or START KEY INVALID KEY establishes a key of reference for purposes of future access to the file. The key of reference is the primary or alternate key of the record read or located. Once the key of reference has been established, it can only be changed by execution of another START or READ INVALID KEY statement.

If the READ is by primary key (KEY IS is omitted), the new key of reference is the primary key value read. Subsequent sequential reads return records in their order in the data file, which is actual key order. Any index file positioning established by START or READ KEY IS INVALID KEY is lost. If the read is by alternate key, the new key of reference is the alternate key value, and subsequent sequential reads return records in the order their keys occur in the index file entries.

The importance of the key of reference is that it determines the order in which records are read by subsequent READ NEXT AT END statements. READ NEXT AT END specifies sequential reading of records. If the key of reference is an alternate key, records are returned to the user program in the order in which their primary keys occur in the alternate key index. When the last record with a given alternate key value has been read, the special-register ERROR-CODE is set to 1000. No other indication is made that the end of the list for that alternate key value has been reached. The next time READ NEXT AT END is executed, if the key of reference has not been changed, the first primary key in the list for the next alternate key value is used to read a record from the data file.

READ ONLY PROCESSING

When an actual key file is opened for INPUT only, the statements READ INVALID KEY and READ NEXT AT END can be used; if the file is a multiple index file, READ KEY IS INVALID KEY and START can also be executed. In other words, any of the information in the file can be accessed, but no changes can be made to the file itself. This kind of processing is called read-only processing; it is useful when the file is to be protected from accidental alteration.

For a non-multiple-index file, if the access mode is sequential, only READ NEXT AT END can be executed. In this case, records are read in the physical order in which they occur on the file; this is the same as actual key order. This method of access is primarily useful when all the records in the file are read, and the order in which they are read is not important.

For a multiple index file, when the access mode is sequential, the START statement can be executed as well as READ NEXT AT END. The START statement positions the file at the first primary key value or alternate key value satisfying the specified comparison. Subsequent execution of READ NEXT AT END returns records in an order that depends on whether the key specified by START was a primary or alternate key. If primary, records are returned in physical order (the same as for a non-multiple-index file). If alternate, records are returned in order by the alternate key; within each alternate key value, records are returned in the order written, or, if INDEXED was specified in the ALTERNATE KEY clause, in order by primary key.

SAMPLE PROGRAM 9: USING ACTUAL KEY FILES

Sample program 9, shown in figure 8-5, illustrates the use of an actual key file to sort a sequential file. Records are read from an unsorted sequential file (UNSORTED-FILE) and a tag file (TAGFILE) is constructed containing the record keys. At the same time, the records are written to an actual key file (SAKFILE). Each record in the tag file links the key of a record in the sequential file with the actual key of the same record in the actual key file. The tag file is then sorted. The sorted tag file is read sequentially, and the full records are extracted from the actual key file and written to the sequential output file (SORTED-FILE). Although this method may seem like an unnecessarily complicated answer to a problem (sorting files) that is easily solved in COBOL, in fact there are cases in which this procedure is more effective than a conventional sort. In particular, a file with very long records (longer than those in this program) can be more easily sorted by using a tag file. The records in the tag file are shorter than the records in the original file, and therefore the tag file can be sorted more quickly than the original file. An interesting feature of this program is that the actual key file created here is only of temporary importance; it can be dispensed with after the program is executed.

Output for the program is shown in figure 8-6.

Line 14 — Line 19

The FILE-CONTROL clauses for SAKFILE include the required clauses SELECT . . . ASSIGN, ORGANIZATION IS, ACCESS MODE, and ACTUAL KEY. The FILE-LIMITS clause restricts the ultimate size of SAKFILE. The clause RESERVE 5 ALTERNATE AREAS specifies that enough buffer space be allocated to hold five data blocks simultaneously. Extra buffer space for actual key files is not necessarily more efficient, and is in fact less efficient when processing is very random.

Line 30 — Line 35

The Data Division clauses for SAKFILE include the required clause LABEL RECORDS OMITTED. The BLOCK CONTAINS clause uses the RECORDS format; this is more efficient than specifying the number of characters, since COBOL converts characters per block to records per block anyway.

Line 59 — Line 68

SAK-KEY is the key item for SAKFILE. It is not within the record format for SAKFILE, since it has no place in the original record format for UNSORTED-FILE, but is only used temporarily. The remainder of the items in the Working-Storage Section are used in the random generation of key values for the original records of UNSORTED-FILE. UNSORTED-FILE is created entirely within this program, but normally it would be an existing file.

Line 74 — Line 84

Lines 74 through 84 are the skeleton of the program, calling into execution all the other lines of the Procedure Division. First UNSORTED-FILE is opened for output and created. Then it is closed and reopened for input. The SORT statement (lines 80 through 83) sorts TAGFILE, and, through the input procedure FIRST-PASS, creates TAGFILE and SAKFILE. It also, through the output procedure LAST-PASS, writes the records of SAKFILE to the output file SORTEDFILE, in the order determined by TAGFILE.

Line 86 — Line 98

The input procedure reads records from UNSORTED-FILE and writes them to SAKFILE. The keys of the records in UNSORTED-FILE are extracted and written to TAGFILE along with the corresponding actual keys. SAK-KEY is set to 0 (line 90) prior to each execution of WRITE INVALID KEY (line 93) thus requesting the system to generate the actual keys. The RELEASE statement (line 95) releases the records of TAGFILE for sorting.

```
00001            IDENTIFICATION DIVISION.
00002
00003            PROGRAM-ID.  SAKDEMO.
00004
00005            ENVIRONMENT DIVISION.
00006
00007            CONFIGURATION SECTION.
00008            SOURCE-COMPUTER. CYBER.
00009            OBJECT-COMPUTER. CYBER.
00010
00011            INPUT-OUTPUT SECTION.
00012
00013            FILE-CONTROL.
00014                SELECT SAKFILE ASSIGN TO FILESAK
00015                    ORGANIZATION IS ACTUAL KEY
00016                    ACCESS MODE RANDOM
00017                    ACTUAL KEY IS SAK-KEY
00018                    FILE-LIMIT 35000
00019                    RESERVE 5 ALTERNATE AREAS.
00020                SELECT UNSORTED-FILE ASSIGN FILEIN
00021                    RESERVE 5 ALTERNATE AREAS.
00022                SELECT SORTEDFILE ASSIGN FILEOUT
00023                    RESERVE 5 ALTERNATE AREAS.
00024                SELECT TAGFILE ASSIGN TAGS.
00025
00026            DATA DIVISION.
00027
00028            FILE SECTION.
00029
00030            FD SAKFILE
00031                LABEL RECORDS OMITTED
00032                BLOCK CONTAINS 16 RECORDS
00033                DATA RECORD SAK-REC.
00034            01 SAK-REC.
00035                02 FILLER PIC X(160).
00036
00037            FD UNSORTED-FILE
00038                LABEL RECORDS OMITTED
00039                BLOCK CONTAINS 640 CHARACTERS
00040                DATA RECORD UNS-REC.
00041            01  UNS-REC.
00042                02 RANDOM-KEY    PIC 9(5).
00043                02  REST         PIC X(155).
00044
00045            FD SORTEDFILE
00046                LABEL RECORDS OMITTED
00047                BLOCK CONTAINS 640 CHARACTERS
00048                DATA RECORD SORTED.
00049            01 SORTED.
00050                02 FILLER        PIC X(160).
00051
00052            SD TAGFILE
00053                LABEL RECORDS OMITTED
00054                DATA RECORD TAG.
00055            01 TAG.
00056                02 SYMB-KEY      PIC 9(5).
00057                02 TAG-KEY       PIC 9(5).
00058
00059            WORKING-STORAGE SECTION.
00060            77  SAK-KEY PIC 9(5) COMP-1.
00061            77  LARGE-PRIME PIC 9(7) COMP-1 VALUE 8388593.
00062            77  SEED PIC 9(5)V99 COMP-1 VALUE 4679.
00063            77  RANDOM-NUMBER PIC 9(7) COMP-1.
00064            01  DOUBLE-NUMBER.
00065                02  FULL-LENGTH PIC 9(7)V9(7).
00066                02  SPLITTER REDEFINES FULL-LENGTH.
00067                    03  INTEGRAL PIC 9(7).
00068                    03 FRACTIONAL PIC 9(7).
00069
```

Figure 8-5. Sample Program 9: Using Actual Key Files (Sheet 1 of 2)

```
00070           PROCEDURE DIVISION.
00071
00072           EXECUTE SECTION.
00073
00074           START.
00075               OPEN OUTPUT UNSORTED-FILE.
00076               PERFORM RECORD-GENERATION  500  TIMES.
00077               CLOSE UNSORTED-FILE.
00078               OPEN INPUT UNSORTED-FILE.
00079               OPEN OUTPUT SAKFILE.
00080               SORT TAGFILE
00081                   ON ASCENDING KEY SYMB-KEY
00082                     INPUT PROCEDURE IS FIRST-PASS
00083                     OUTPUT PROCEDURE IS LAST-PASS.
00084               STOP RUN.
00085
00086           FIRST-PASS SECTION.
00087
00088           READ-LOOP.
00089               READ UNSORTED-FILE AT END GO TO END-OF-FIRST-PASS.
00090               MOVE ZERO TO SAK-KEY.
00091               MOVE RANDOM-KEY TO SYMB-KEY.
00092               MOVE UNS-REC TO SAK-REC.
00093               WRITE SAK-REC INVALID KEY DISPLAY ≠FAULT 1≠ STOP RUN.
00094               MOVE SAK-KEY TO TAG-KEY.
00095               RELEASE TAG.
00096               GO TO READ-LOOP.
00097           END-OF-FIRST-PASS.
00098               CLOSE UNSORTED-FILE SAKFILE.
00099
00100           LAST-PASS SECTION.
00101
00102           GET-READY.
00103               OPEN OUTPUT SORTEDFILE INPUT SAKFILE.
00104           WRITE-LOOP.
00105               RETURN TAGFILE AT END GO TO END-OF-LAST-PASS.
00106               MOVE TAG-KEY TO SAK-KEY.
00107               READ SAKFILE INVALID KEY DISPLAY ≠FAULT 2≠ STOP RUN.
00108               MOVE SAK-REC TO SORTED.
00109               WRITE SORTED.
00110               GO TO WRITE-LOOP.
00111           END-OF-LAST-PASS.
00112               CLOSE SORTEDFILE SAKFILE.
00113
00114           RECORD-GENERATION SECTION.
00115
00116           GEN-A-REC.
00117               ADD 1 TO SEED.
00118               DIVIDE LARGE-PRIME BY SEED GIVING FULL-LENGTH.
00119               ADD INTEGRAL FRACTIONAL GIVING RANDOM-NUMBER.
00120               MOVE RANDOM-NUMBER TO RANDOM-KEY.
00121               MOVE ALL ≠  THEQUICKBROWNFOX  ≠ TO REST.
00122               WRITE UNS-REC.
00123
00124           GEN-A-REC-EXIT.
00125               EXIT.
```

Figure 8-5. Sample Program 9: Using Actual Key Files (Sheet 2 of 2)

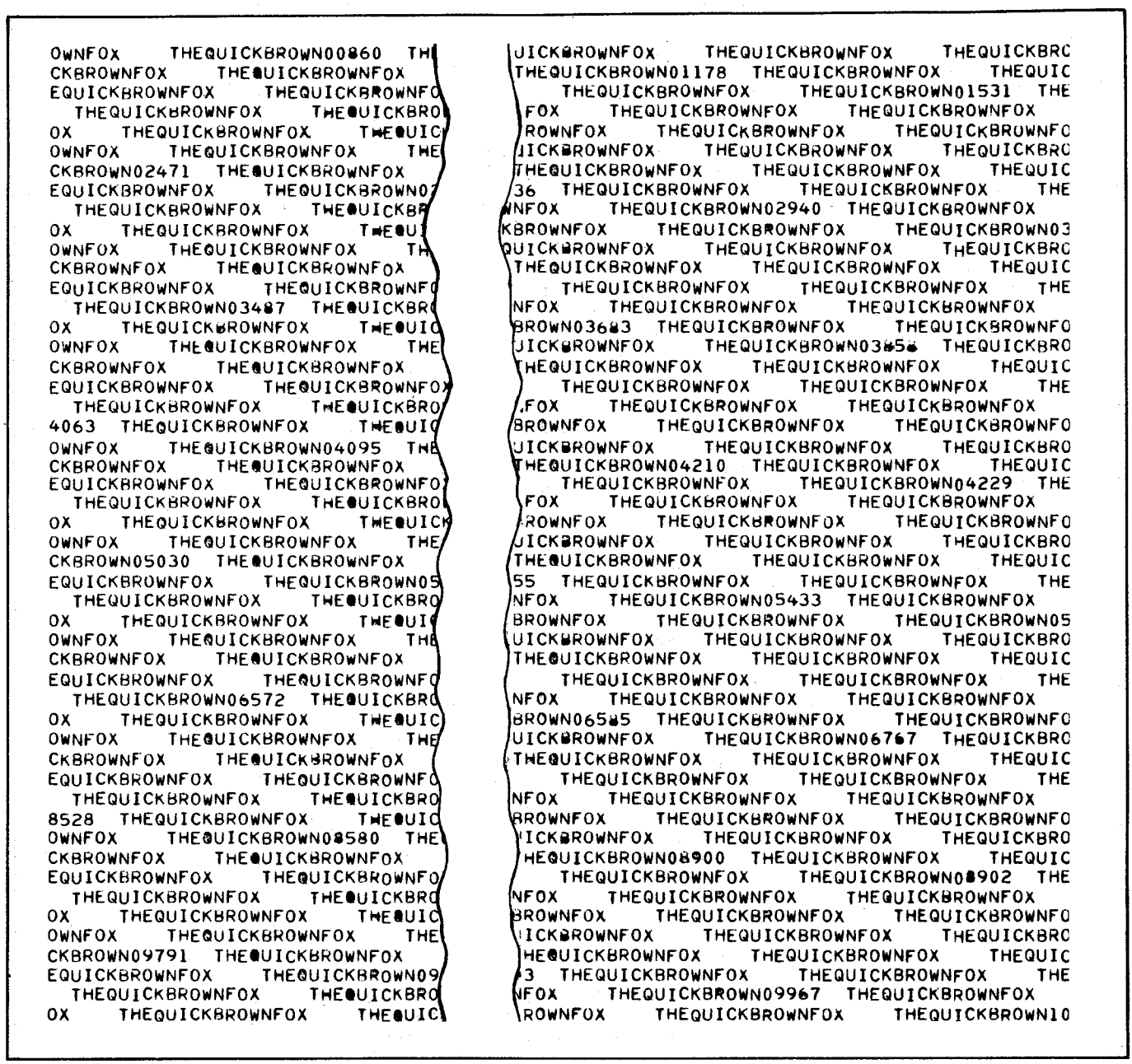

Figure 8-6. Output from Sample Program 9

Line 100 — Line 112

The output procedure accepts the sorted records of TAGFILE through the RETURN statement (line 105) and uses them to read SAKFILE randomly (line 107). The records of SAKFILE are thus read in the order of the original keys from UNSORTED-FILE. The records are then written to SORTED-FILE in this order, ensuring that the records are in order on SORTEDFILE.

Line 114 — Line 125

The original records for UNSORTED-FILE are generated in lines 114 through 125. The key of each record is generated through a random number procedure, and the rest of each record is filled with irrelevant data.

SAMPLE PROGRAM 10: MULTIPLE INDEX ACTUAL KEY FILES

Sample program 10, shown in figure 8-7, illustrates the use of actual key files with multiple index structure defined. The program creates an actual key file, NAME-FILE, based on records contained on the card file CRDFILE, and then reads selected records based on information on the card file CARD-FILE. One primary key, PERS-NO, and one alternate key, DATE-A, are defined for NAME-FILE. Sample input for program 10 is shown in figure 8-8, and sample output in figure 8-9.

```
00001            IDENTIFICATION DIVISION.
00002
00003            PROGRAM-ID. ACTUAL-KEY-MIP.
00004
00005            ENVIRONMENT DIVISION.
00006
00007            CONFIGURATION SECTION.
00008            SOURCE-COMPUTER. CYBER.
00009            OBJECT-COMPUTER. CYBER.
00010
00011            INPUT-OUTPUT SECTION.
00012            FILE-CONTROL.
00013                SELECT CRDFILE ASSIGN TO INPUT.
00014                SELECT CARD-FILE ASSIGN TO INPUT.
00015                SELECT NAME-FILE ASSIGN OFILE MIP2
00016                ACCESS IS RANDOM
00017                ORGANIZATION IS ACTUAL KEY
00018                ACTUAL KEY IS PERS-NO
00019                ALTERNATE RECORD KEY IS DATE-A DUPLICATES.
00020
00021            DATA DIVISION.
00022
00023            FILE SECTION.
00024
00025            FD CRDFILE
00026                LABEL RECORDS ARE OMITTED
00027                DATA RECORDS ARE CARD-REC.
00028            01  CARD-REC.
00029                02  IN-KEY.
00030                    03  IN-K1    PIC X(6).
00031                    03  IN-K2    PIC 9(4).
00032                    03  IN-K3    PIC X(8).
00033                02  IN-AMT       PIC 9(8).
00034                02  IN-COLOR     PIC X(8).
00035                02  I-DATE-A     PIC 9(5).
00036                02  IN-RANK1     PIC X(20).
00037
00038            FD  CARD-FILE
00039                LABEL RECORDS ARE OMITTED
00040                DATA RECORDS ARE CARD-RECORD.
00041            01  CARD-RECORD.
00042                02  INPERS-NO    PIC 9(5).
00043                02  FILLER       PIC X(75).
00044
00045            FD  NAME-FILE
00046                LABEL RECORDS ARE OMITTED.
00047                BLOCK CONTAINS 5110 CHARACTERS
00048                DATA RECORDS ARE PERS-REC.
00049            01  PERS-REC.
00050                02  KEY-1.
00051                    03 K1        PIC X(6).
00052                    03  PERS-NO PIC 9(4) USAGE COMP-1.
00053                    03  CITY     PIC X(8).
00054                02  AMT          PIC 9(6)V99.
00055                02  COLOR        PIC X(8).
00056                02  DATE-A       PIC 9(5) USAGE COMP-1.
00057                02  RANK.
00058                    03  TITLEA  PIC X(11).
00059                    03  TEAM-T1 PIC X(9).
00060
00061            PROCEDURE DIVISION.
00062
00063            CREATE-RANDOM-FILE.
00064                OPEN INPUT CRDFILE
00065                    OUTPUT NAME-FILE.
00066
00067            READ1.
00068                READ CRDFILE AT END GO TO END-CREATE.
00069                MOVE IN-K1 TO K1.
```

Figure 8-7. Sample Program 10: Multiple Index Actual Key Files (Sheet 1 of 2)

```
00070                   MOVE ZEROS TO PERS-NO.
00071                   MOVE IN-K3 TO CITY.
00072                   MOVE IN-AMT TO AMT.
00073                   MOVE IN-COLOR TO COLOR.
00074                   MOVE I-DATE-A TO DATE-A.
00075                   MOVE IN-RANK1 TO RANK.
00076                   WRITE PERS-REC  INVALID KEY DISPLAY
00077                       ≠INVALID KEY IS ≠ PERS-NO
00078                       GO TO END-RUN.
00079                   GO TO READ1.
00080
00081               END-CREATE.
00082                   CLOSE CRDFILE, NAME-FILE.
00083
00084               INITIALIZE-ALL.
00085                   OPEN INPUT NAME-FILE, CARD-FILE.
00086
00087               SELECT-REC.
00088                   READ CARD-FILE AT END GO TO END-RUN.
00089                   MOVE  INPERS-NO TO DATE-A
00090                   START NAME-FILE KEY IS = DATE-A
00091                       INVALID KEY DISPLAY ≠INVALID KEY - KEY IS  ≠ DATE-A
00092                       GO TO END-RUN.
00093
00094               READ-II.
00095                   READ NEXT NAME-FILE AT END GO TO END-RUN.
00096                   DISPLAY K1 PERS-NO CITY AMT COLOR DATE-A RANK.
00097                   GO TO READ-II.
00098
00099               END-RUN.
00100                   CLOSE NAME-FILE, CARD-FILE.
00101                   STOP RUN.
```

Figure 8-7. Sample Program 10: Multiple Index Actual Key Files (Sheet 2 of 2)

```
PERSON0079BROOKFIE90497850BROWN   72123DRAFTSMAN  ASST TRS
PERSON0123CHICAGO 00300198BLUE    69043PROGRAMMER VICE PRES
PERSON0456CHICAGO 00215835PURPLE  69152CLERK      PRESIDENT
PERSON0796WINNETKA00792354BLUE    69187TYPIST     COMMITTEE
PERSON2272HINSDALE50053916YELLOW  72189TRAINEE    COMMITTEE
PERSON2459EVANSTON00254981BROWN   70219MANAGER    CHAIRMAN
PERSON3313CICERO  06061472YELLOW  732900PERATOR   V CHAIRMAN
PERSON3796BATAVIA 21856379YELLOW  70123PROGRAMMER MEMBER
PERSON4845CHICAGO 70275138YELLOW  71301ENGINEER   SECRETARY
PERSON4890BATAVIA 10213576PURPLE  70154ANALYST    MEMBER
PERSON5167LA GRANG02426554BLUE    69265SECRETARY  TREASURER
PERSON5968LYONS   08682394BROWN   74012SECRETARY  MEMBER
PERSON6594CHICAGO 01597624PURPLE  69152ENGINEER   COMMITTEE
PERSON6894LYONS   30637732PURPLE  71368CLERK-TYP  MEMBER
PERSON7156WESTERN 04348910BLUE    70077MANAGER    COMMITTEE
PERSON7598CHICAGO 01795842BROWN   692150PERATOR   COMMITTEE
```

Figure 8-8. Input for Sample Program 10

Line 15 — Line 19

The FILE-CONTROL clauses for NAME-FILE include required clauses SELECT . . . ASSIGN, ACCESS IS RANDOM, ORGANIZATION IS, and ACTUAL KEY. Additionally, one alternate key is defined by the ALTERNATE RECORD KEY clause; duplicate values are permitted for this key. The first file defined by the ASSIGN clause, OFILE, is the actual key data file; the second, MIP2, is the index file.

Line 63 — Line 82

NAME-FILE is opened for output and created with records from CRDFILE.

```
PERSON0008BATAVIA 85637900YELLOW  70123PROGRAMMER MEMBER
PERSON0010BATAVIA 21357600PURPLE  70154ANALYST    MEMBER
PERSON0006EVANSTON25498100BROWN   70219MANAGER    CHAIRMAN
PERSON0009CHICAGO 27513800YELLOW  71301ENGINEER   SECRETARY
PERSON0014LYONS   63773200PURPLE  71368CLERK-TYP  MEMBER
PERSON0001BROOKFIE49785000BROWN   72123DRAFTSMAN  ASST TRS
PERSON0005HINSDALE05391600YELLOW  72189TRAINEE    COMMITTEE
PERSON0007CICERO  06147200YELLOW  732900PERATOR   V CHAIRMA
PERSON0012LYONS   68239400BROWN   74012SECRETARY  MEMBER
```

Figure 8-9. Output from Sample Program 10

Line 84 — Line 101

NAME-FILE is reopened for input and read. A single key is read from CARD-FILE (line 88); the key value read is 70123 in this case. The START statement positions the index file based on this key value. Subsequent iterations of the READ NEXT AT END statement read the records of NAME-FILE in the order their primary key values occur on the index file, following the alternate key value 70123 for DATE-A. The AT END clause is executed on the next read after the last key value for DATE-A has been read.

The FILE control statement establishes file structure specifications different from, or in addition to, those usually established by COBOL. This facility is implemented by assigning values to fields in the file information table (FIT).

Every file processed through CYBER Record Manager must have a valid file information table at the time the file is opened. For files referenced in COBOL programs, the COBOL compiler establishes a table for each file during compilation and sets fields in the table to appropriate values based on the clauses and statements that process the file. These tables become part of the compiled object program. Most of the file information table fields needed for a given file are set during compilation, but some might be reset by the COBOL execution time routines. For example, the block type (BT) field for a sequential file is reset if the COBOL execution time routines determine that the block type originally selected is inappropriate for the device on which the file actually resides. (Device residence cannot be determined at compile time.) In most cases, however, the values selected during compilation remain in effect throughout program execution.

If one or more FILE control statements have been provided for a file, the values they specify are placed in the file information table when the file is opened during program execution.

The FILE control statement has two uses of interest to the COBOL programmer. The first is to enable other processors or languages to use files created through COBOL programs. For example, a FORTRAN Extended 4 program can read a COBOL-created file if a FILE control statement specifies the same file structure produced by the original COBOL program. Even file structures not provided as part of standard FORTRAN input/output (such as D, R, or T type records, or K or E type blocks) can be specified through the FILE control statement. There are limitations, however, on the changes that can be made in this way. For instance, standard FORTRAN Extended input/output cannot successfully read an indexed sequential, direct, or actual key file, regardless of any parameters specified on the FILE control statement.

In order to specify the correct parameters in the FILE control statement, it is necessary to know what file information table fields are used by COBOL and how they are used. Tables 9-1 through 9-8 describe the fields and values used by COBOL for each of the six file organizations, six record types, and four block types. These tables also describe the specifications within the COBOL program that are used to set these fields.

The second use of the FILE control statement is to alter or supplement default COBOL file processing. File information table field settings provided by the FILE control statement take effect when the file is opened at execution time, and override specifications established in the table by COBOL during compilation. Thus, file structures other than those defined in the source program result when the program is executed. Although the system will not prevent it, a FILE control statement should not in general be used for this purpose. The COBOL language contains within it adequate provision for virtually any type of file processing likely to be of value to the programmer. Furthermore, unexpected consequences can result when the user overrides the file information table values already compiled in a program. However, a small number of file information table fields provide processing options not available through COBOL, and can be safely set by a FILE control statement accompanying execution of a COBOL program. These fields are shown in table 9-9.

None of the tables in this section provide much detail about the file information table fields they list. Such detail is beyond the scope of this user guide. For more information about the use, settings, and default values of these fields, the reader is referred to the Record Manager Reference Manual.

FILE CONTROL STATEMENT USAGE

The format of the FILE control statement is:

FILE(lfn,field=value,field=value, . . . , field=value)

lfn	Logical file name (COBOL implementor-name)
field	File information table field mnemonic, as listed in the tables in this section
value	Value to be placed in corresponding field

When a FILE control statement is encountered in a job stream, its parameters are stored on a temporary file for subsequent reference by processing programs. If subsequent FILE control statements are encountered in a job for a given file, the information on them is stored together with that of the previous statements. When multiple specifications appear for any specific parameter, the last specification encountered is used. When a program that references a given file is called for execution, the prestored parameters are set in the file information table when the file is opened (provided an LDSET statement has been included, as explained below).

The sequence of events can be illustrated as follows:

1. The following control statement is encountered:

   FILE(PAYFILE,BT=C,RT=Z)

   The block type (BT) and record type (RT) parameters are placed in temporary storage in association with the name PAYFILE.

2. The following control statement is encountered:

   FILE(PAYFILE,RT=F,FL=80)

   A new record type parameter (RT=F) that cancels the existing one (RT=Z) is stored, and the fixed length parameter (FL) is added to the parameters that already exist for the file named PAYFILE.

3. The following control statement is encountered:

   LDSET(FILES=PAYFILE)

   The LDSET control statement indicates that the loading operation must include the code modules required for processing of the file PAYFILE. In this case, the

modules necessary for sequential file organization are to be loaded, and those pertaining to C blocks and F records. The LDSET statement must be in the same load set as the statement calling for execution of the object program.

4. The following control statement is encountered:

   LGO.

   The loader loads the program currently on LGO (the compiled COBOL program) and consults the temporary storage area where the FILE control statement information is stored. From information found there, it determines the routines that are to handle C blocking and F records, and loads the routines. It also loads a special routine, PDF.RM, to be used in the next step in the sequence.

5. The following statement is to be executed from the loaded program:

   OPEN INPUT PAYROLL-FILE.

   The file name PAYROLL-FILE was associated with the logical file name PAYFILE in the ASSIGN statement of the COBOL program. The file is now to be opened for input processing.

   Before opening the file, CYBER Record Manager ascertains that PDF.RM has been loaded as specified by the LDSET control statement. When absolute programs (such as programs with overlay structure) are loaded, the LDSET control statement is ignored; consequently, PDF.RM must be loaded by the compiler. This is accomplished by placing the statement ENTER "PDF.RM" immediately after a STOP statement in the Procedure Division of the COBOL program. The ENTER statement is never executed. The external reference generated by the ENTER statement causes the loader to load PDF.RM. When the file is opened, the OPEN statement processing detects that PDF.RM is present. PDF.RM is then entered; it searches the temporary file that contains the FILE statement parameters. It places the parameters associated with the file named PAYFILE in the correct fields in the file information table for PAYFILE. The file is then opened. All subsequent file processing operations for PAYFILE use the information contained in the file information table.

FILE CONTROL STATEMENT EXAMPLES

The examples shown in figures 9-1 through 9-13 illustrate the correspondence between COBOL Environment and Data Division clauses and file information table fields. In each case, the COBOL clauses used to define a sequential file with specific block and record types are shown, followed by the FILE control statement that would communicate the same information to a different processor reading the file. The parameters on the FILE control statement are then explained. The block and record type combinations illustrated here are those most likely to be specified in a COBOL program. These examples assume that the file resides on an appropriate device; if not, COBOL will change the block type at execution time.

TABLE 9-1. FIT FIELDS BY RECORD TYPE

Mnemonic	Description	Setting
For D type records:		
RT	Record type	D (decimal character count)
MNR	Minimum record length	integer-1 from RECORD CONTAINS
MRL	Maximum record length	integer-2 from RECORD CONTAINS
LP	Start of count field	Length of items preceding data-name in DEPENDING ON option
LL	Length of count field	Length of data-name in DEPENDING ON option
For F type records:		
RT	Record type	F (fixed length)
FL	Fixed length	Length of longest Record Description entry
For R type records:		
RT	Record type	R (record mark)
RMK	Record mark character	Right bracket character @ (62B)
MRL	Maximum record length	Calculated from Record Description
For T type records:		
RT	Record type	T (trailer count)
MNR	Minimum record length	Calculated from integer-1 in OCCURS option
MRL	Maximum record length	Calculated from integer-2 in OCCURS option
CP	Start of trailer count field	Length of items preceding data-name in DEPENDING ON option
CL	Length of trailer count field	Length of data-name in DEPENDING ON option
HL	Header length	Length of items preceding subject of OCCURS clause
TL	Length of trailer item	Length of subject of OCCURS clause
For W type records:		
RT	Record type	W (control word)
MRL	Maximum record length	Length of longest Record Description entry
RL	Record length	Length of specified Record Description entry (reset for each WRITE)
For Z type records:		
RT	Record type	Z (zero byte terminated)
FL	Fixed length	Length of longest Record Description entry

TABLE 9-2. FIT FIELDS BY BLOCK TYPE

Mnemonic	Description	Setting
For K type blocks:		
BT	Block type	K (fixed number of records per block)
RB	Number of records per block	integer-1 value from BLOCK CONTAINS . . . RECORDS; 1 if clause omitted
For C type blocks:		
BT	Block type	C (character count)
MBL	Fixed block length	Predefined; value depends on device type
For E type blocks:		
BT	Block type	E (exact records)
MNB	Minimum block length	integer-1 from BLOCK CONTAINS . . . CHARACTERS
MBL	Maximum block length	integer-2 from BLOCK CONTAINS . . . CHARACTERS
For I type blocks:		
BT	Block type	I (internal control word)
MBL	Fixed block length	Predefined as 5120

TABLE 9-3. FIT FIELDS FOR SEQUENTIAL FILES

Mnemonic	Description	Setting
FO	File organization	SQ (sequential)
BT	Block type	From BLOCK CONTAINS
RT	Record type	From RECORD CONTAINS and other clauses (section 2)
CM	Conversion mode	Binary if so specified by RECORDING MODE; decimal otherwise
LFN	Logical file name	Implementor-name in ASSIGN
BFS	Buffer size	Calculated from RESERVE . . . ALTERNATE AREAS
PD	Processing direction	As specified by OPEN; OUTPUT if OPEN EXTEND
OF	Open action	N (NO REWIND), E (EXTEND) if specified in OPEN; R (rewind) otherwise
CF	Close action	N (NO REWIND), U (LOCK) if specified in CLOSE; R (rewind) otherwise
VF	End of volume action	U (unload)
DX	End of data exit	From AT END clause of READ statement
EX	Error exit	From USE AFTER ERROR procedure
ULP	User label processing	YES if USE BEFORE/AFTER LABEL procedures declared; NO otherwise
LX	User label routine	From USE BEFORE/AFTER LABEL procedure
LA	User label location	From data-names in LABEL RECORDS
LBL	Length of label area	Calculated from data-names in LABEL RECORDS

TABLE 9-4. FIT FIELDS FOR RELATIVE FILES

Mnemonic	Description	Setting
FO	File organization	WA (word addressable)
RT	Record type	Set from RECORD CONTAINS and other clauses (section 2)
LFN	Logical file name	Implementor-name in ASSIGN
BFS	Buffer size	Calculated from RESERVE . . . ALTERNATE AREAS
PD	Processing direction	As specified by OPEN; OUTPUT if OPEN EXTEND
OF	Open action	N (NO REWIND), E (EXTEND) if so specified in OPEN; R (rewind) otherwise
CF	Close action	R (rewind)
VF	End of volume action	U (unload)
DX	End of data exit	From AT END clause of READ statement
EX	Error exit	From USE AFTER ERROR procedure or INVALID KEY clause

TABLE 9-5. FIT FIELDS FOR STANDARD FILES

Mnemonic	Description	Setting
FO	File organization	WA (word addressable)
RT	Record type	From RECORD CONTAINS and other clauses (section 2)
LFN	Logical file name	Implementor-name in ASSIGN
BFS	Buffer size	Calculated from RESERVE . . . ALTERNATE AREAS
PD	Processing direction	As specified by OPEN
OF	Open action	R (rewind)
CF	Close action	R (rewind)
VF	End of volume action	U (unload)
EX	Error exit	From USE AFTER ERROR procedure or INVALID KEY clause

TABLE 9-6. FIT FIELDS FOR DIRECT FILES

Mnemonic	Description	Setting
FO	File organization	DA (direct access)
RT	Record type	From RECORD CONTAINS and other clauses (section 2)
LFN	Data file name	implementor-name-1 in ASSIGN
XN	Index file name	implementor-name-2 in ASSIGN
BFS	Buffer size	Calculated from RESERVE . . . ALTERNATE AREAS
PD	Processing direction	As specified by OPEN
HMB	Number of home blocks	From NUMBER OF BLOCKS
MBL	Size of home blocks	From BLOCK or RECORD-BLOCK CONTAINS, or calculated
OVF	Overflow record residence	Overflow records can be in both home and overflow blocks
RB	Records per block	From BLOCK or RECORD-BLOCK CONTAINS (used only to calculate MBL)
RKW	Beginning word of key	From Record Description
RKP	Key position within RKW	From Record Description
KL	Key length	From Record Description
HRL	Hashing routine	From USE FOR HASHING procedure
DX	End of data exit	From AT END clause of READ statement
EX	Error exit	From USE AFTER ERROR procedure or INVALID KEY clause
KA	Key location on access	From ACTUAL/RECORD/SYMBOLIC KEY
REL	Alternate key comparison	From relational-operator in START statement
NDX	Index record indicator	Index or data file search for START statement

TABLE 9-7. FIT FIELDS FOR INDEXED SEQUENTIAL FILES

Mnemonic	Description	Setting
FO	File organization	Indexed sequential
RT	Record type	From RECORD CONTAINS and other clauses (section 2)
LFN	Data file name	implementor-name-1 in ASSIGN
XN	Index file name	implementor-name-2 in ASSIGN
BFS	Buffer size	Calculated from RESERVE . . . ALTERNATE AREAS
PD	Processing direction	As specified by OPEN
MBL	Data block length	From BLOCK or RECORD-BLOCK CONTAINS, or calculated
RB	Records per block	From BLOCK or RECORD-BLOCK CONTAINS (used only to calculate MBL)
DP	Data block padding	From DATA-PADDING
IBL	Index block length	From INDEX-BLOCK CONTAINS or calculated
IP	Index block padding	From INDEX-PADDING
NL	Number of index levels	From INDEX-LEVEL (used only to calculate IBL)
KT	Key type	According to usage of key item
KL	Key length	From Record Description
OF	Open action	R (rewind)
DX	End of data exit	From AT END clause of READ statement
EX	Error exit	From USE AFTER ERROR procedure or INVALID KEY clause
KA	Key location on access	From RECORD/SYMBOLIC KEY
KP	Key position within KA word	From RECORD/SYMBOLIC KEY
DKI	Duplicate key permission	YES if DUPLICATES in RECORD KEY, NO otherwise
MKL	Major key length	Length of key item named in READ MAJOR INVALID KEY
RKW	Starting word of key	From Record Description (multiple index files only)
RKP	Position of key in RKW word	From Record Description (multiple index files only)
REL	Alternate key comparison	From relational-operator in START statement
NDX	Index record indicator	Index or data file search for START statement

TABLE 9-8. FIT FIELDS FOR ACTUAL KEY FILES

Mnemonic	Description	Setting
FO	File organization	AK (actual key)
RT	Record type	From RECORD CONTAINS and other clauses (section 2)
LFN	Data file name	implementor-name-1 in ASSIGN
XN	Index file name	implementor-name-2 in ASSIGN
BFS	Buffer size	Calculated from RESERVE . . . ALTERNATE AREAS
PD	Processing direction	As specified by OPEN
MBL	Data block length	From BLOCK CONTAINS or calculated
RB	Records per block	From BLOCK CONTAINS or calculated
KL	Key length	From Record Description
DX	End of data exit	From AT END clause of READ statement
EX	Error exit	From USE AFTER ERROR procedure or INVALID KEY clause
KA	Key location on access	From ACTUAL KEY clause
RKW	Starting word of key	From Record Description (multiple index files only)
RKP	Position of key in RKW	From Record Description (multiple index files only)
REL	Alternate key comparison	From relational-operator in START statement
NDX	Index record indicator	Index or data file search for START statement

TABLE 9-9. OTHER FILE CONTROL STATEMENT PARAMETERS

Mnemonic	Description	Applicability	Notes
BCK	Block checksum option	FO=IS, DA, AK	
DP	Data block padding percent	FO=IS, DA, AK	IS, DA: specified through DATA-PADDING clause; AK: specified through FILE control statement only
EO	Bad data disposition	FO=SQ	Determines system action after USE AFTER ERROR procedure
ERL	Trivial error limit	All file organizations	COBOL default is 3
EXD	Extended diagnostics	FO=SQ, WA	
FWI	Force write option	FO=IS, DA, AK	
MUL	Record length multiple	BT=K, E	
PC	Padding character	BT=K, E	
RMK	Record mark character	RT=R	Required if character other than 62B (right bracket character) is to be used as RECORD-MARK
SDS	Dayfile statistics	All file organizations	
SPR	Suppress read ahead	FO=SQ	
TRC	Trace option	FO=IS, DA	

COBOL clauses:

```
     SELECT EE-TEE ASSIGN FILEET
     RESERVE 4 ALTERNATE AREAS.
       .
       .
       .
FD   EE-TEE
     LABEL RECORDS OMITTED
     RECORD CONTAINS 10 TO 40 CHARACTERS
     BLOCK CONTAINS 5 TO 15 RECORDS.
01   ET-REC.
     02  STUFF PIC 9(5).
     02  ET-KEY PIC 9(3).
     02  VARY-PART PIC XX OCCURS 1 TO 16 TIMES DEPENDING ON ET-KEY.
```

Corresponding FILE control statement:

```
FILE(FILEET,BT=E,RT=T,MBL=600,MNB=50,HL=8,TL=2,CL=3,CP=5,CM=YES,MRL=40,LT=U)
```

FILEET	Logical file name from ASSIGN clause
BT=E	E block type given by format BLOCK CONTAINS integer-1 TO integer-2 RECORDS
RT=T	Type T records resulting from RECORD CONTAINS integer-1 TO integer-2 CHARACTERS and DEPENDING ON option in File Description entry
MBL=600	Maximum block length set at 600, since maximum record size is 40 characters and each block contains a maximum of 15 records, given by RECORD CONTAINS and BLOCK CONTAINS
MNB=50	Minimum block size set at 50 with minimum of 10 characters per record and 5 records per block, stated in RECORD CONTAINS and BLOCK CONTAINS
HL=8	Fixed-length portion of T record is 8 characters, determined by 02 level entries describing ET-REC
TL=2	Length of trailer portion of T record is 2, determined by PIC XX in 02 level entry describing VARY-PART
CL=3	Number of characters in key field is 3, defined by ET-KEY PIC 9(3)
CP=5	Beginning character position of key item. ET-KEY PIC 9(3) occupies character positions 6 through 8.
CM=YES	Conversion mode corresponds to RECORDING MODE IS DECIMAL (COBOL defaults to CM=YES)
MRL=40	Maximum record length of 40 characters specified by RECORD CONTAINS clause
LT=U	File is unlabeled, as indicated by LABEL RECORDS OMITTED clause

Figure 9-1. E Type Blocks, T Type Records (First Example)

COBOL clauses:

```
      SELECT EE-TEE ASSIGN FILEET
      RESERVE 4 ALTERNATE AREAS.
        .
        .
        .
  FD  EE-TEE
      LABEL RECORDS OMITTED
      BLOCK CONTAINS 5000 TO 5120 CHARACTERS
      DATA RECORD TEE-REC.
  01  TEE-REC.
      02  FIXED-PART.
        03  HEAD-INFO PIC X(12).
        03  FURTHER-INFO PIC X(8).
        03  OCCURRENCE-COUNT PIC 9(2).
      02  TABLE.
        03  ELEMENT OCCURS 1 TO 99 TIMES DEPENDING ON OCCURRENCE-COUNT.
          04  HOURS PIC 99.
          04  RATE PIC 99V99.
```

Corresponding FILE control statement:

```
FILE(FILEET,BT=E,MNB=5000,MBL=5120,RT=T,HL=22,TL=6,CP=20,CL=2,MRL=616,CM=YES,LT=U)
```

FILEET	Name of file, as stated in SELECT . . . ASSIGN clause
BT=E	E block type, given by format of BLOCK CONTAINS integer-1 TO integer-2 RECORDS
MNB=5000	Minimum block size is set at 5000 characters, indicated by integer-1 in BLOCK CONTAINS clause
MBL=5120	Minimum block size is set at 5120 characters, indicated by integer-2 in BLOCK CONTAINS clause
RT=T	Record type is T, determined by Record Description
HL=22	Fixed-length portion of T record is 22 characters, sum of FIXED-PART item lengths
TL=6	Length of trailer portion is 6 characters, sum of character positions in TABLE description
CP=20	Beginning character position of trailer count field of record is position 20 (counted from 0), determined by location of OCCURRENCE-COUNT
CL=2	Length of trailer count field is 2, specified by OCCURRENCE-COUNT field size
MRL=616	Maximum record length is 616 characters
CM=YES	Decimal conversion mode (COBOL default)
LT=U	Unlabeled file, declared by LABEL RECORDS OMITTED clause

Figure 9-2. E Type Blocks, T Type Records (Second Example)

COBOL clauses:

```
    SELECT KEY-DEE ASSIGN FILEKD
      .
      .
      .
FD  KAY-DEE
    LABEL RECORDS OMITTED
    BLOCK CONTAINS 15 RECORDS
    RECORD CONTAINS 5 TO 30 CHARACTERS DEPENDING ON D-KEY.
01  KD-REC.
    02  D-KEY PIC 9999.
    02  MISC PIC X(26).
```

Corresponding FILE control statement:

```
FILE(FILEKD,RT=D,BT=K,RB=15,MRL=30,MNR=5,MBL=450,MNB=75,LP=0,LL=4,CM=YES)
```

FILEKD	Name of file created, as given in ASSIGN clause
RT=D	Type D records, indicated by RECORD CONTAINS integer-1 TO integer-2 CHARACTERS DEPENDING ON data-name
BT=K	K block type, given by BLOCK CONTAINS integer RECORDS
RB=15	Number of records per block in BLOCK CONTAINS
MRL=30	Maximum record length is 30 characters, in RECORD CONTAINS clause
MNR=5	Minimum record length is 5, in RECORD CONTAINS clause
MBL=450	Maximum block length is 450 characters, product of 30 characters for each of 15 records, specified by RECORD CONTAINS and BLOCK CONTAINS
MNB=75	Minimum block length is 75 characters, product of 5 characters for each of 15 records, given by BLOCK CONTAINS and RECORD CONTAINS
LP=0	Beginning character position of length field is 0, since D-KEY is in positions 0 through 3
LL=4	Number of characters in length field is 4; D-KEY PIC 9999
CM=YES	Corresponds to RECORDING MODE IS DECIMAL

Figure 9-3. K Type Blocks, D Type Records (First Example)

COBOL clauses:

```
     SELECT KAY-DEE ASSIGN FILEKD
     RESERVE 4 ALTERNATE AREAS.
       .
       .
       .
FD   KAY-DEE
     LABEL RECORDS OMITTED
     BLOCK CONTAINS 10 RECORDS
     RECORD CONTAINS 14 TO 80 CHARACTERS DEPENDING ON LENGTH-COUNT
     DATA RECORD DEE-REC.
01   DEE-REC.
     02 FIXED-PART.
        03 HEADING-PORTION PIC X(10).
        03 LENGTH-COUNT PIC 9(4).
     02 STRING-PART.
        03 CHARACTER PIC X OCCURS 66 TIMES.
       .
       .
       .
     MOVE LENGTH TO LENGTH-COUNT.
     WRITE DEE-REC.
```

Corresponding FILE control statement:

```
FILE(FILEKD,BT=K,RB=10,MNB=140,MBL=800,MNR=14,MRL=80,RT=D,LP=10,LL=4,CM=YES)
```

FILEKD	Name of file created, given in ASSIGN clause
BT=K	K block type, given by BLOCK CONTAINS
RB=10	Number of records per block in BLOCK CONTAINS
MNB=140	Minimum block size is 140 characters; product of 10 records, each containing minimum of 14 characters; specified by BLOCK CONTAINS and RECORD CONTAINS
MBL=800	Maximum block length of 800 characters, product of 10 records, each containing 80 characters maximum; specified by BLOCK CONTAINS and RECORD CONTAINS
MNR=14	Minimum record length is 14 in RECORD CONTAINS
RT=D	D record type, specified by format of RECORD CONTAINS
LP=10	Beginning character position of length field, LENGTH-COUNT of FIXED-PART, is 10
LL=4	Length of record length field, LENGTH-COUNT, is 4
CM=YES	Recording mode is decimal

Figure 9-4. K Type Blocks, D Type Records (Second Example)

COBOL clauses:

```
    SELECT EE-R ASSIGN FILEER
    RESERVE 4 ALTERNATE AREAS.
      .
      .
      .
FD  EE-R
    LABEL RECORDS OMITTED
    RECORD CONTAINS 10 TO 20 CHARACTERS DEPENDING ON RECORD-MARK
    BLOCK CONTAINS 5 TO 10 RECORDS.
01  ER-REC.
    02  R-M-O PIC X OCCURS 20 TIMES.
```

Corresponding FILE control statement:

```
FILE(FILEER,BT=E,RT=R,RMK=62B,MNR=10,MRL=20,MBL=200,MNB=50,CM=YES)
```

FILEER	Name of file as given in ASSIGN clause
BT=E	E block type, given by BLOCK CONTAINS clause format
RT=R	R record type, given by RECORD CONTAINS format
RMK=62B	Record-mark character display code is 62 octal; B specifies octal
MNR=10	Minimum record length is 10 characters, in RECORD CONTAINS
MRL=20	Maximum record length is 20 characters, in RECORD CONTAINS
MBL=200	Maximum block length is product of 20 characters per record and 10 records per block, defined in RECORD CONTAINS and BLOCK CONTAINS
MNB=50	Minimum block length is 50, product of 10 blocks and 5 records per block, given by BLOCK CONTAINS and RECORD CONTAINS
CM=YES	Corresponds to RECORDING MODE IS DECIMAL

Figure 9-5. E Type Blocks, R Type Records (First Example)

COBOL clauses:

```
     SELECT EE-R ASSIGN FILEER
     RESERVE 4 ALTERNATE AREAS.
       .
       .
       .
FD   EE-R
     LABEL RECORDS OMITTED
     BLOCK CONTAINS 1 TO 30 RECORDS
     RECORD CONTAINS 14 TO 80 CHARACTERS DEPENDING ON RECORD-MARK
     DATA RECORD R-REC.
01   R-REC.
     02  HEADER-PORTION PIC X(13).
     02  STRING.
         03  STRING-CHARACTER PIC X OCCURS 66 TIMES.
       .
       .
       .
     MOVE RECORD-MARK TO STRING-CHARACTER (SUB).
     WRITE R-REC.
       .
       .
       .
```

Corresponding FILE control statement:

```
FILE(FILEER,BT=E,MBL=2400,MNB=14,RT=R,MNR=14,MRL=80,RMK=62B,CM=YES)
```

FILEER	Name of file created, as given in ASSIGN clause
BT=E	E block type, given by BLOCK CONTAINS clause
MBL=2400	Maximum block length is 2400 characters, product of 30 records per block and 80 characters per record, as given in BLOCK CONTAINS and RECORD CONTAINS
MNB=14	Minimum block length of 1 record per block and 14 characters per record included in BLOCK CONTAINS and RECORD CONTAINS
RT=R	R record type, given by RECORD CONTAINS
MNR=14	Minimum record length of 14, as determined by RECORD CONTAINS
MRL=80	Maximum record length, given by RECORD CONTAINS
RMK=62B	Record-mark character display code is 62 octal; octal is indicated by B
CM=YES	Corresponds to RECORDING MODE IS DECIMAL

Figure 9-6. E Type Blocks, R Type Records (Second Example)

COBOL clauses:

```
    SELECT EYE-W. ASSIGN TO IWFILE-FIW
    RESERVE 4 ALTERNATE AREAS.
      .
      .
      .
FD  EYE-W
    LABEL RECORDS OMITTED
    RECORDING MODE BINARY.
01  IW-REC-1 PIC X(20).
01  IW-REC-2 PIC X(30).
```

Corresponding FILE control statement:

```
FILE(IWFILE,BT=I,RT=W,MRL=30,MNR=20,CM=NO)
```

IWFILE	Name of file created, as given in ASSIGN clause
BT=I	I block type, specified by -I suffixed to implementor-name
RT=W	W record type, specified by -W suffixed to implementor-name
MRL=30	Maximum record length is 30, derived from larger of two records, IW-REC-1 and IW-REC-2
MNR=20	Minimum record length is 20, smaller of two records
CM=NO	Corresponds to RECORDING MODE IS BINARY

Figure 9-7. I Type Blocks, W Type Records (First Example)

COBOL clauses:

```
     SELECT EYE-W ASSIGN TO IWFILE-FIW
     RESERVE 4 ALTERNATE AREAS.
FD   EYE-W
     LABEL RECORDS ARE OMITTED
     RECORDING MODE BINARY
     DATA RECORDS ARE SHORT MEDIUM LONG.
01   SHORT.
     02  FILLER PIC X(50).
01   MEDIUM.
     02  FILLER PIC X(80).
01   LONG.
     02  FILLER PIC X(120).
     .
     .
     .
     WRITE LONG.
     .
     .
     .
     WRITE SHORT.
     .
     .
     .
     WRITE MEDIUM.
     .
     .
     .
```

Corresponding FILE control statement:

```
FILE(IWFILE,BT=I,RT=W,MNR=50,MRL=120,CM=NO)
```

IWFILE	Name of file created, as given in ASSIGN clause
BT=I	I block type, specified by -I suffixed to implementor-name IW-FILE
RT=W	W record type, specified by -W suffixed to implementor-name
MNR=50	Minimum record length is 50 characters, in record description SHORT
MRL=120	Maximum record length is 120 characters, in record description LONG
CM=NO	Corresponds to RECORDING MODE IS BINARY

Figure 9-8. I Type Blocks, W Type Records (Second Example)

COBOL clauses:

```
     SELECT KAY-EFF ASSIGN FILEKF RESERVE 4 ALTERNATE AREAS.
     .
     .
     .
FD   KAY-EFF
     LABEL RECORDS OMITTED
     BLOCK CONTAINS 10 RECORDS
     DATA RECORD F-REC.
01   F-REC.
     02  FILLER PIC X(80).
     .
     .
     .
     MOVE DATA TO F-REC.
     WRITE F-REC.
     .
     .
     .
```

Corresponding FILE control statement:

```
FILE(FILEKF,BT=K,RT=F,RB=10,FL=80,MBL=800,MNB=800,CM=YES)
```

FILEKF	Name of file as given in ASSIGN clause
BT=K	K block type, denoted by format of BLOCK CONTAINS
RT=F	F record type, determined by format of RECORD CONTAINS
RB=10	Number of records per block, in BLOCK CONTAINS
FL=80	Fixed-length record is 80 characters, specified by data record description F-REC
MBL=800	Maximum block length is 800 characters, product of 80 characters per record (PICTURE clause) and 10 records per block (BLOCK CONTAINS clause)
MNB=800	Minimum block length, given by same parameters as MBL in this example
CM=YES	Implied by absence of RECORDING MODE statement

Figure 9-9. K Type Blocks, F Type Records

COBOL clauses:

```
     SELECT CEE-EFF ASSIGN FILECF
     .
     .
     .
FD   CEE-EFF
     LABEL RECORDS OMITTED.
01   CF-REC SIZE 50.
```

Corresponding FILE control statement:

```
FILE(FILECF,BT=C,RT=F,FL=50,CM=YES,MBL=640)
```

FILECF	File name, as given in ASSIGN clause
BT=C	C block type, assigned when no BLOCK CONTAINS clause is specified
RT=F	F record type, automatically assigned when RECORD CONTAINS clause is absent
FL=50	Record length is 50 characters
CM=YES	Corresponds to RECORDING MODE IS DECIMAL (DECIMAL is default)
MBL=640	Maximum block length is 640 characters; mass storage device is assumed

Figure 9-10. C Type Blocks, F Type Records (First Example)

COBOL clauses:

```
        SELECT CEE-EFF ASSIGN FILECF
        RESERVE 4 ALTERNATE AREAS.
        .
        .
        .
  FD    CEE-EFF
        LABEL RECORDS OMITTED
        BLOCK CONTAINS 640 CHARACTERS
        DATA RECORD F-REC.
  01    F-REC.
        02  FILLER PIC X(80).
        .
        .
        .
        MOVE DATA TO F-REC.
        WRITE F-REC.
```

Corresponding FILE control statements:

```
FILE(FILECF,BT=C,RT=F,FL=80,CM=YES)
```

FILECF	Name of file created, given in ASSIGN clause
BT=C	C block type, given by format of BLOCK CONTAINS clause
RT=F	F record type, assigned automatically by default
FL=80	Fixed-length of record is 80 characters, given by description of F-REC
CM=YES	Conversion mode corresponds to RECORDING MODE IS DECIMAL

Figure 9-11. C Type Blocks, F Type Records (Second Example)

COBOL clauses:

```
    SELECT CZ-FILE ASSIGN FILECZ-FZ
    .
    .
    .
FD  CZ-FILE
    LABEL RECORDS OMITTED.
01  CZ-REC SIZE 50.
```

Corresponding FILE control statement:

```
FILE(FILECZ,BT=C,RT=Z,FL=50,CM=YES,MBL=640)
```

FILECZ	Name given to file created
BT=C	C block type, automatically assigned if no BLOCK CONTAINS clause is used
RT=Z	Record type Z, specified by appending -FZ to implementor-name in ASSIGN clause
FL=50	Record length is 50 characters
CM=YES	Corresponds to RECORDING MODE IS DECIMAL
MBL=640	Maximum block length is 640 characters; disk device assumed

Figure 9-12. C Type Blocks, Z Type Records (First Example)

COBOL clauses:

```
     SELECT CZ-FILE ASSIGN FILECZ-FZ
     RESERVE 4 ALTERNATE AREAS.
     .
     .
     .
FD   CZ-FILE
     LABEL RECORDS OMITTED
     BLOCK CONTAINS 640 CHARACTERS
     DATA-RECORD F-REC.
01   F-REC.
     02  FILLER PIC X(80).
     .
     .
     .
     MOVE DATA TO F-REC.
     WRITE F-REC.
```

Corresponding FILE control statement:

```
FILE(FILECZ,BT=C,RT=Z,FL=80,CM=YES)
```

FILECZ	Name of created file
BT=C	C block type, given by format of BLOCK CONTAINS clause
RT=Z	Z record type, given by -FZ appended to implementor-name in SELECT/ASSIGN clause
FL=80	Fixed-length record is 80 characters, given by F-REC size description
CM=^YES	Corresponds to RECORDING MODE IS DECIMAL

Figure 9-13. C Type Blocks, Z Type Records (Second Example)

STANDARD CHARACTER SET

A

Control Data operating systems offer the following variations of a basic character set:

CDC 64-character set

CDC 63-character set

ASCII 64-character set

ASCII 63-character set

These character sets are listed in table A-1. The set in use at a particular installation was specified when the operating system was installed.

Depending on another installation option, the system assumes an input deck has been punched either in 026 or in 029 mode (regardless of the character set in use). The user, however, may specify the alternate mode by a 26 or 29 punched in columns 79 and 80 of the job card or any 7/8/9 card. The specified mode remains in effect through the end of the job unless it is reset by specification of the alternate mode on a subsequent 7/8/9 card.

Under NOS 1, the alternate mode can be specified also by a 26 or 29 punched in columns 79 and 80 of any 6/7/9 card, as described above for a 7/8/9 card. In addition, 026 mode can be specified by a card with 5/7/9 multipunched in column 1, and 029 mode can be specified by a card with 5/7/9 multipunched in column 1 and a 9 punched in column 2.

When the 63-character set is used, the display code character 00 is converted to a space (display code 55_8).

No conversions occur when the 64-character set is used.

Tables A-2 and A3 show the CDC and ASCII character set collating sequences respectively.

STANDARD CHARACTER SETS

CDC Graphic	ASCII Graphic Subset	Display Code	Hollerith Punch (026)	External BCD Code	ASCII Punch (029)	ASCII Code	CDC Graphic	ASCII Graphic Subset	Display Code	Hollerith Punch (026)	External BCD Code	ASCII Punch (029)	ASCII Code
:†	:	00††	8-2	00	8-2	072	6	6	41	6	06	6	066
A	A	01	12-1	61	12-1	101	7	7	42	7	07	7	067
B	B	02	12-2	62	12-2	102	8	8	43	8	10	8	070
C	C	03	12-3	63	12-3	103	9	9	44	9	11	9	071
D	D	04	12-4	64	12-4	104	+	+	45	12	60	12-8-6	053
E	E	05	12-5	65	12-5	105	–	–	46	11	40	11	055
F	F	06	12-6	66	12-6	106	*	*	47	11-8-4	54	11-8-4	052
G	G	07	12-7	67	12-7	107	/	/	50	0-1	21	0-1	057
H	H	10	12-8	70	12-8	110	(	(	51	0-8-4	34	12-8-5	050
I	I	11	12-9	71	12-9	111	)	)	52	12-8-4	74	11-8-5	051
J	J	12	11-1	41	11-1	112	$	$	53	11-8-3	53	11-8-3	044
K	K	13	11-2	42	11-2	113	=	=	54	8-3	13	8-6	075
L	L	14	11-3	43	11-3	114	blank	blank	55	no punch	20	no punch	040
M	M	15	11-4	44	11-4	115	, (comma)	, (comma)	56	0-8-3	33	0-8-3	054
N	N	16	11-5	45	11-5	116	. (period)	. (period)	57	12-8-3	73	12-8-3	056
O	O	17	11-6	46	11-6	117	≡	#	60	0-8-6	36	8-3	043
P	P	20	11-7	47	11-7	120	[	[	61	8-7	17	12-8-2	133
Q	Q	21	11-8	50	11-8	121	]	]	62	0-8-2	32	11-8-2	135
R	R	22	11-9	51	11-9	122	%	%	63††	8-6	16	0-8-4	045
S	S	23	0-2	22	0-2	123	≠	" (quote)	64	8-4	14	8-7	042
T	T	24	0-3	23	0-3	124	→	_ (underline)	65	0-8-5	35	0-8-5	137
U	U	25	0-4	24	0-4	125	∨	!	66	11-0 or 11-8-2†††	52	12-8-7 or 11-0†††	041
V	V	26	0-5	25	0-5	126							
W	W	27	0-6	26	0-6	127	∧	&	67	0-8-7	37	12	046
X	X	30	0-7	27	0-7	130	↑	' (apostrophe)	70	11-8-5	55	8-5	047
Y	Y	31	0-8	30	0-8	131	↓	?	71	11-8-6	56	0-8-7	077
Z	Z	32	0-9	31	0-9	132	<	<	72	12-0 or 12-8-2†††	72	12-8-4 or 12-0†††	074
0	0	33	0	12	0	060							
1	1	34	1	01	1	061	>	>	73	11-8-7	57	0-8-6	076
2	2	35	2	02	2	062	≤	@	74	8-5	15	8-4	100
3	3	36	3	03	3	063	≥	\	75	12-8-5	75	0-8-2	134
4	4	37	4	04	4	064	¬	^ (circumflex)	76	12-8-6	76	11-8-7	136
5	5	40	5	05	5	065	; (semicolon)	; (semicolon)	77	12-8-7	77	11-8-6	073

†Twelve or more zero bits at the end of a 60-bit word are interpreted as end-of-line mark rather than two colons. End-of-line mark is converted to external BCD 1632.

††In installations using a 63-graphic set, display code 00 has no associated graphic or card code; display code 63 is the colon (8-2 punch). The % graphic and related card codes do not exist and translations from ASCII/EBCDIC % yield a blank (55_8).

†††The alternate Hollerith (026) and ASCII (029) punches are accepted for input only.

TABLE A-1. STANDARD CHARACTER SETS

TABLE A-2. CDC CHARACTER SET COLLATING SEQUENCE

CDC CHARACTER SET COLLATING SEQUENCE									
Collating Sequence Decimal	Collating Sequence Octal	CDC Graphic	Display Code	External BCD	Collating Sequence Decimal	Collating Sequence Octal	CDC Graphic	Display Code	External BCD
00	00	blank	55	20	32	40	H	10	70
01	01	≤	74	15	33	41	I	11	71
02	02	%	63 †	16 †	34	42	∨	66	52
03	03	[	61	17	35	43	J	12	41
04	04	→	65	35	36	44	K	13	42
05	05	≡	60	36	37	45	L	14	43
06	06	∧	67	37	38	46	M	15	44
07	07	↑	70	55	39	47	N	16	45
08	10	↓	71	56	40	50	O	17	46
09	11	>	73	57	41	51	P	20	47
10	12	≥	75	75	42	52	Q	21	50
11	13	¬	76	76	43	53	R	22	51
12	14	.	57	73	44	54	]	62	32
13	15	)	52	74	45	55	S	23	22
14	16	;	77	77	46	56	T	24	23
15	17	+	45	60	47	57	U	25	24
16	20	$	53	53	48	60	V	26	25
17	21	*	47	54	49	61	W	27	26
18	22	-	46	40	50	62	X	30	27
19	23	/	50	21	51	63	Y	31	30
20	24	,	56	33	52	64	Z	32	31
21	25	(	51	34	53	65	:	00 †	none†
22	26	=	54	13	54	66	0	33	12
23	27	≠	64	14	55	67	1	34	01
24	30	<	72	72	56	70	2	35	02
25	31	A	01	61	57	71	3	36	03
26	32	B	02	62	58	72	4	37	04
27	33	C	03	63	59	73	5	40	05
28	34	D	04	64	60	74	6	41	06
29	35	E	05	65	61	75	7	42	07
30	36	F	06	66	62	76	8	43	10
31	37	G	07	67	63	77	9	44	11

†In installations using the 63-graphic set, the % graphic does not exist. The : graphic is display code 63, External BCD code 16.

TABLE A-3. ASCII CHARACTER SET COLLATING SEQUENCE

ASCII CHARACTER SET COLLATING SEQUENCE								
Collating Sequence Decimal/Octal		ASCII Graphic Subset	Display Code	ASCII Code	Collating Sequence Decimal/Octal		ASCII Graphic Subset	Display Code	ASCII Code
00	00	blank	55	20	32	40	@	74	40
01	01	!	66	21	33	41	A	01	41
02	02	"	64	22	34	42	B	02	42
03	03	#	60	23	35	43	C	03	43
04	04	$	53	24	36	44	D	04	44
05	05	%	63†	25	37	45	E	05	45
06	06	&	67	26	38	46	F	06	46
07	07	'	70	27	39	47	G	07	47
08	10	(	51	28	40	50	H	10	48
09	11	)	52	29	41	51	I	11	49
10	12	*	47	2A	42	52	J	12	4A
11	13	+	45	2B	43	53	K	13	4B
12	14	,	56	2C	44	54	L	14	4C
13	15	-	46	2D	45	55	M	15	4D
14	16	.	57	2E	46	56	N	16	4E
15	17	/	50	2F	47	57	O	17	4F
16	20	0	33	30	48	60	P	20	50
17	21	1	34	31	49	61	Q	21	51
18	22	2	35	32	50	62	R	22	52
19	23	3	36	33	51	63	S	23	53
20	24	4	37	34	52	64	T	24	54
21	25	5	40	35	53	65	U	25	55
22	26	6	41	36	54	66	V	26	56
23	27	7	42	37	55	67	W	27	57
24	30	8	43	38	56	70	X	30	58
25	31	9	44	39	57	71	Y	31	59
26	32	:	00†	3A	58	72	Z	32	5A
27	33	;	77	3B	59	73	[	61	5B
28	34	<	72	3C	60	74	\	75	5C
29	35	=	54	3D	61	75	]	62	5D
30	36	>	73	3E	62	76	^	76	5E
31	37	?	71	3F	63	77	_	65	5F

†In installations using a 63-graphic set, the % graphic does not exist. The : graphic is display code 63.

GLOSSARY B

Actual Key – Integers used in a random access file. When a READ or WRITE is executed, the key value is used to locate or place a logical record. Also, a file organization.

ANSI – The American National Standards Institute. In this manual, ANSI is used to denote the 1968 version of the ANSI COBOL standard (X3.23-1968).

BOI (Beginning-of-Information) – The start of the first user record in a file. System information such as tape labels of sequential files or indexes does not affect beginning-of-information.

Block – A physical grouping of records.

Blocking – The method by which a specified number of records is organized within a block and handled as a unit to increase input-output efficiency.

Buffer – A temporary storage area that allows input/output devices to operate independently of the central processing unit.

Checksum – System verification of data accuracy. Block checksums can be requested for direct, indexed sequential, and actual key files (see CYBER Record Manager User's Guide).

Data Block – A block in which user records and keys are stored in an indexed sequential file. The user specifies block size or a blocking factor. CYBER Record Manager manipulates data blocks to keep records in sequential order by key.

Declaratives – COBOL statements that invoke procedures defined in the Procedure Division to be executed when a condition occurs that cannot be tested with standard COBOL procedures.

Default – The system-supplied value employed when the user does not define a parameter.

Direct File – A file that contains records stored randomly in home blocks according to the hashed value of the key in each record. The file must be mass storage resident. All allocation for home blocks occurs when the file is opened on its creation run.

Directives – Instructions that supplement processing defined by a control statement or program call for execution of a utility function or member of a product set.

EOI (End-of-Information) – The point beyond which no records exist on a given file. Trailer labels are past end-of-information.

File – A logically related set of information; the largest collection of information that can be addressed by a file name. Starts at beginning-of-information and ends at end-of-information.

FILE Control Statement – A control statement that contains parameters used to build the file information table for processing.

FIT (File Information Table) – A table through which a user program communicates with CYBER Record Manager. All file processing executes on the basis of fields in the table. CYBER Record Manager manipulates the FIT for the COBOL user; FIT fields are initially set by COBOL source statements.

FSTT (File Statistics Table) – A table generated and maintained by CYBER Record Manager to collect statistics about direct, indexed sequential, and actual key files. The FSTT is a permanent part of a file; it contains information such as file organization, block size, and number of current accesses. The COBOL user cannot access the FSTT.

Hashing – The process of mapping keys to produce a relative home block address for records in a file with direct organization.

Home Block – A block in a direct file; the relative address of the block is computed by hashing keys. A home block contains synonym records whose keys hash to that relative address. If all the synonym records do not fit in the home block, an overflow block might be created by the system. When the direct file is created, the user must define the number and size of the home blocks with COBOL statements.

Index – A series of keys and pointers to records associated with the keys.

Index Block – A block with ordered keys and pointers to data blocks and other index blocks. Used for indexed sequential files to form a directory of the data blocks within the file.

Key – Group of contiguous characters or numbers the user defines to identify a record in an indexed sequential or direct access file.

L Tape – Long record stranger magnetic tape. PRU size is not restricted.

LDSET – Loader control statement; when a FILE statement is used, an LDSET statement containing the FILES parameter is necessary.

Local File – File maintained only for duration of the run unit.

Major Key – The leading characters of a record or symbolic key in an indexed sequential file.

Mass Storage – A disk or disk pack that can be accessed randomly as well as sequentially.

Multiple-Index File – A file containing an index that associates primary and alternate keys in cross-reference form. The file can be accessed on any of the alternate keys.

Overflow Block – A block created by CYBER Record Manager for use when home blocks in a direct file are full.

Padding – Free space reserved in a file at creation to accommodate additional records; specified as a percentage figure.

Permanent File – A file on a mass storage permanent file device that is protected against accidental destruction by the system and can be protected against unauthorized access or destruction. A CATALOG (SCOPE 3.4) or SAVE or DEFINE (KRONOS 2.1) is required to make a file permanent.

Physical Record – On magnetic tape, information between inter-record gaps. It need not contain a fixed amount of data.

PRU (Physical Record Unit) – The smallest unit of information that can be transferred between a peripheral storage device and central memory. The PRU size is permanently fixed for all operating system devices; the concept does not apply to S/L devices. PRU sizes are given in table 3-1.

Random Access File – A file from which records can be retrieved in a nonsequential manner.

Record – A group of 6-bit characters; the smallest collection of information passed between CYBER Record Manager and the user. The user defines the structure and characteristics of records within a file by declaring a record format. The beginning and ending points of a record are implicit in each format.

Record Ordinal – A number that specifies a record's position in a series of records.

Reel –Synonymous with volume.

Run – File accesses encompassing file open through close. Reopening a file in a single program constitutes a second run.

S Tape – A magnetic tape with recording format of physical records containing the contents of 512 central memory words of binary information or 128 words of coded information. A stranger tape.

Sequential Access – Records read or written record-by-record from the beginning to the end of a file; access by specifying successive positions.

Source Code – A COBOL user program that must be compiled or assembled before execution.

Symbolic Key – A character string used to specify a record in a random access file. Used only for standard, direct, and indexed sequential files.

Utility Routines – Operating system routines that provide functions used universally with most programs. Such routines include input-output, diagnostics, CREATE, sorting routines, and the multiple-index capability (IXGEN).

Volume – A reel of magnetic tape or one sequential disk pack. A given file can encompass more than one volume.

CYBER Record Manager provides utilities which help the programmer define efficient file structures and create files.

ESTMATE UTILITY

The efficiency with which an indexed sequential file can be processed is influenced by the size of its data blocks and index blocks, the number of index levels, and the size of the buffer in the user field length in central memory. COBOL-calculated default values might not be the best for a particular file.

The utility ESTMATE can be used to calculate index block length, data block length, and buffer length. ESTMATE uses a description of the file records and keys to produce suggested sizes for blocks and the central memory buffer, based on various possible blocking factors, padding factors, and index levels. Buffer length can then be specified directly through the BFS parameter on the FILE control statement, or indirectly through the RESERVE ALTERNATE AREAS clause.

Usually, the ESTMATE utility is run as a separate job. Program results are printed on the standard OUTPUT file with values for each directive. When only one set of parameters for a file structure is given to ESTMATE, parameters can appear on the ESTMATE control statement. Otherwise, the deck structure is:

Job control statement

ACCOUNT statement if required by the operating system

ESTMATE control statement

7/8/9

Directives describing possible file structure parameters

6/7/8/9

On the ESTMATE control statement parameters may appear in any order:

.ESTMATE(NR=x,KS=x,MR=x,MI=x)

NR	Approximate number of records in file
KS	Key length in characters; number of characters in alphanumeric key; must be specified as 10 for COMPUTATIONAL-1 and COMPUTATIONAL-2 key items
MR	Maximum number of characters in record
MI	Minimum number of characters in record

All subsequent calculations are based on the above file definition.

When only one set of block parameters is to be used, the following parameters also can appear on the ESTMATE control statement:

NL	Number of desired index levels, 1 through 63
BF	Blocking factor (number of records per block) for average size records in data blocks
PI	Index block padding percentage, 0 through 99
PD	Data block padding percentage, 0 through 99

An empty directive section should be provided if any processing occurs after ESTMATE. ESTMATE always reads a section from the INPUT file; therefore, the INPUT file must be positioned correctly for any subsequent processing.

Directives have the format below. The * in column 1 is required. Parameters on directives must appear in the order listed and may be separated by spaces, commas, or other special characters. Installation default values are substituted for any parameters not specified by the user.

*nl,bf,pi,pd

nl	Number of index levels anticipated, 1 through 63
bf	Blocking factor for average size records in data blocks
pi	Index block padding percentage, 0 through 99
pd	Data block padding percentage, 0 through 99

Output from ESTMATE shows the number of words for the buffer and blocks. Figure C-1 illustrates two versions of a job deck under NOS/BE 1 and the resulting output (the same in both cases).

The buffer referenced in the output is the buffer within the user field length that holds index and data blocks manipulated by CYBER Record Manager. By default, a size greater than minimum is provided. For an indexed sequential file, a larger buffer often increases processing efficiency. The suggested buffer size should be used if possible.

CREATE UTILITY

The CREATE utility can be called through a COBOL program to create a direct file. The default hashing routine or a user-supplied hashing routine may be used.

CREATE should be used for large files (more than 1000 records). It significantly reduces creation time, since all records that hash to a given home block can be written in one mass storage access. Otherwise, a home block must be transferred from the central memory buffer to mass storage for each record to be written.

CREATE hashes the key of an input record and prefixes the hashed key to the record. After all records have been read, CREATE calls Sort/Merge to sort records according to the hashed key results. When the sort is complete, CREATE calls CYBER Record Manager to create a direct file. The hashed keys are removed; they do not become part of the direct file records.

Sample Job Deck 1

```
ESTJOB,CM40000,T10.
ESTMATE,NR=8000,KS=10,MR=500,MI=450,
NL=3,BF=30,PI=5,PD=4.
6/7/8/9
```

Sample Job Deck 2

```
ESTIMAT,CM40000.
ESTMATE(MI=450,MR=500,KS=10,NR=8000)
7/8/9
*3,30,5,4
6/7/8/9
```

Output:

```
ESTMATE,NR=8000,KS=10,MR=500,MI=450,NL=3,BF=30,PI=5,PD=4.

                        INDEXED SEQUENTIAL FILE ESTMATE

 NUMBER OF RECORDS=       8000                  KEY    SIZE=       10 CHARACTERS

 MINIMUM RECORD SIZE=        45 WORDS        MAXIMUM RECORD SIZE=        50 WORDS

   NUMBER                          INDEX          DATA          MINIMUM      SUGGESTED
  OF INDEX        ACCESS           BLOCK          BLOCK          BUFFER        BUFFER
   LEVELS          MODE            SIZE           SIZE            SIZE          SIZE
                                  (WORDS)        (WORDS)         (WORDS)       (WORDS)

      3           RANDOM             63           1535            1768          3442

      3        SEQUENTIAL                         1535            1627          3164
```

Figure C-1. ESTMATE Example

The key for each input record must be in the record. CREATE cannot combine a separate key with a record.

A job using CREATE must contain:

FILE control statement to describe the direct file structure, including the key position in the record

Execution of source program that reads a record and calls CREATE

CREATE directive in a separate section on the job INPUT file

A FILE control statement must be used to describe file structure. The direct file cannot be defined within a user program calling CREATE.

The FILE control statement must include the following parameters or accept the default noted:

LFN	Logical file name
FO	File organization FO=DA required
HMB	Number of home blocks
MBL	Number of characters in home block. May be specified or calculated by the methods discussed in section 6.
MNR	Minimum record length in characters
MRL	Maximum record length in characters

KL — Key length in characters

RKW — Word in record in which key begins, counting ten 6-bit fields to a word. Words numbered from 0. Default 0.

RKP — Position in word described by RKW in which key begins, counting 6-bit fields 0-9 left to right. Default 0.

RT — Record type, plus any other parameters needed to complete record description. Default is Z type records with MRL defining FL.

Values in the SDACRTU call in a program cannot override key position existing at file open.

CREATE DIRECTIVE

The CREATE directive has the format:

CREATE(dafile,hash,hashlfn)

dafile — Name of direct file; required

hash — Entry point to user hashing routine. If omitted, default hashing routine is used.

hashlfn — Logical file name of file containing user hashing routine; required only if a user hashing routine is used

Only one input directive is possible. Parameters may be separated by any legal separator. Embedded blanks are allowed. The directive must be terminated by a period or right parenthesis.

User hashing routines must be in relocatable binary format. The file identified in the CREATE directive must be made local to the job prior to the utility call.

CREATE CALLS IN SOURCE PROGRAMS

When CREATE is used in a source program, two calls to library routines are needed:

SDACRTU — Must be called for each record, so the key will be hashed and affixed to that record

SDAENDC — Called once after all records are read, so records will be sorted and inserted into direct file

The user must read each record and call SDACRTU with parameters identifying the key in the record, the record location, and the number of characters in the record. After all records are read, SDAENDC must be called to complete direct file creation.

SDACRTU call parameters:

ENTER SDACRTU ka recarea length.

ka — Key item in record identified by recarea. If key is not left-justified in the word ka, the RKP parameter of the FILE control statement must be used to specify alignment.

recarea — Record name

length — Integer record length in characters

SDAENDC is called by:

ENTER SDAENDC

The source program must not reference the direct file being created.

KEY ANALYSIS UTILITY

The key analysis utility helps the user select a hashing routine or the number of home blocks that best suits a particular file. Ideally, a good hashing routine results in a uniform distribution of records in all home blocks, with no overflow blocks.

When the records hashed to a home block exceed the block size, an overflow block must be used. Consequently, at least two mass storage accesses are required to read each record that could not be accommodated in the original home block, with a resulting increase in execution time. Any record in a home block that is not filled, on the other hand, can always be read with a single mass storage access. Faster execution time is gained, but mass storage requirements for the file might be greater if no overflow blocks exist.

The key analysis utility provides information about hypothetical record distribution for the file. By changing the number of home blocks or the routine that distributes records among those blocks, the user can balance mass storage requirements and access time considerations before actual file creation. The utility reads the key of each record in the file and determines the home block where the record would reside. After all keys are examined, statistics are output.

Results from the utility are written to a file named KEYLIST. The user must rewind KEYLIST and copy it to OUTPUT if the results are to be printed. A successful execution produces the message:

END KEY ANALYZER

USER HASHING ROUTINES

A user hashing routine to be run with the key analyzer utility must have the following characteristics:

Be in relocatable binary format on the file identified in the LFN parameter of the KYAN directive. (Relocatable binary format results from normal COBOL compilation.)

Have an entry point identified in the H parameter of the KYAN directive.

Execution must result in an integer value in the lower bits of a word. CYBER Record Manager divides this value by the number of home blocks to obtain the hashed key. The division uses only the lower 48 bits of the user provided integer.

If the routine with the entry point named on the KYAN directive is not available to the analyzer the following message appears:

KEY ANALYZER CANNOT LOCATE name

During key analysis, the utility might output several messages indicating a faulty hashing routine:

ENTRYn – SYNONYM LIMIT EXCEEDED

More than 4095 keys hash to the same home block using the routine identified by Hn on the KYAN directive. The key analyzer discontinues use of this routine for the remainder of the run.

ENTRYn – BAD KEY ENCOUNTERED

The routine identified by Hn produces a value outside the range of the number of home blocks. This key is not counted in the output. If 26 of these messages appear for any one hashing routine, the job is terminated with the message MORE THAN 25 BAD KEYS ENCOUNTERED.

KEY ANALYZER EXECUTION

The user can call the key analysis utility through a COBOL program. The user must supply a KYAN directive in a separate section in the job deck to describe the proposed file parameters.

KYAN Directive

Format of the KYAN directive is given below. All parameters must be specified in the order shown; no default values exist. The parameter list, which must be enclosed in parentheses, may contain extra blanks after commas, but blanks cannot be embedded within a parameter. If only one hashing routine is to be used, the parameter list must be terminated by a right parenthesis after the output format option parameter of that routine. Up to five routines can be specified.

When all parameters cannot be contained on a single line, as many as six continuation lines can be added for one KYAN directive. Column 80 must contain a slash to indicate continuation in column 1 of the next line.

Directive format is:

```
KYAN(LFN=lfn,MRL=mrl,KL=kl,RKP=rkp,
   RKW=rkw,H1=entry,hmb,outopt,...,
   H5=entry,hmb,outopt)
```

lfn — Name of file containing any user supplied hashing routines in relocatable binary format

mrl — Maximum record length in characters

kl — Key length in characters

rkp — Character position of start of key in word designated by the RKW parameter

rkw — Relative word in record in which the key begins, counting the first word as 0 with 10 characters in each word

n — Hashing routine identifier 1-5 for user documentation purposes only. It may be omitted. The three parameters immediately following apply to each entry name.

entry — Entry point name of hashing routine. The entry point for the default routine is SDAHASH.

hmb — Number of home blocks

outopt — Output option:

- S — For each home block list the number of keys that hash to that block (synonyms)
- D — Output only standard deviation of records in all blocks
- B — Output both synonyms and standard deviation

As many as five hashing routine entry points can be listed on the KYAN directive. The user has the option of specifying five different routines or specifying the same entry point more than once and varying the number of home blocks associated with the entry point. Any combination of entry points and home blocks can be specified, including both the default and user supplied routines, but no more than five routines can be called for key analysis through one KYAN directive.

The default hashing routine resides on a system library and is always available to the calling job. (Nevertheless, when only the default hashing routine is specified, the LFN parameter is still required; any file name can be used in this case.) When several user supplied hashing routines exist, they must all reside on the file identified by the LFN parameter.

Calling Key Analysis Utility Through COBOL

In a COBOL program, the key analysis utility is identified by SDAKEYH and SDAENDH, the two entry points to the COMPASS coded utility program. SDAKEYH must be entered repeatedly after each record is read so that each key can be hashed; when all keys are read and processed, SDAENDH must be entered to write output to the file KEYLIST. To examine the utility output, the user should rewind KEYLIST and copy it to the file OUTPUT or otherwise cause it to be printed.

When the utility is entered the first time, it expects the next unexecuted section in the job INPUT file to contain the KYAN directive. The utility, itself, calls for loading the file containing the hashing routines. The user should ensure that the file is available to the job, but should not load the hashing routines.

If more than one hashing routine entry point is given on the KYAN directive, each key is examined by each hashing routine before control returns to the user program.

The general structure of a job deck for a COBOL program call to the key analyzer is:

Job control statement

COBOL.

LGO.

Output file KEYLIST to printer

7/8/9

COBOL program that executes SDAKEYH for each data record and SDAENDH after all records have been processed as outlined in the language sections

7/8/9

KYAN directive

6/7/8/9

Field length requirements for execution of a source program using the key analysis utility are increased by 1600 words plus the largest number of home blocks specified.

KEY ANALYZER OUTPUT

Information output from the B option is shown in figure C-2.

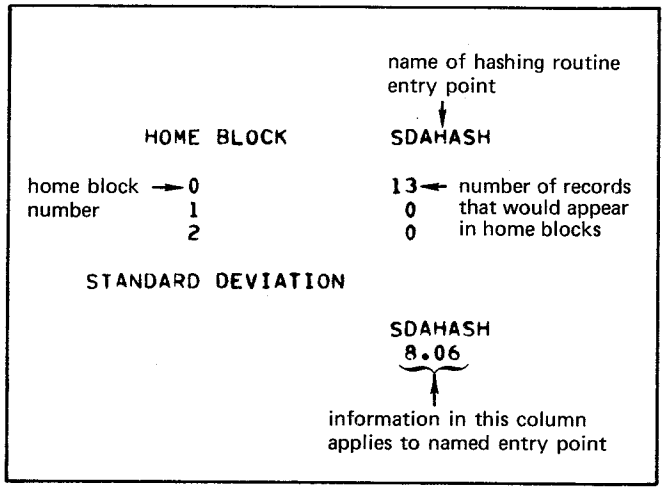

Figure C-2. Key Analyzer Output

In an ideal situation, the hashing routine would result in the same number of records in each home block, and that number would be a value equal to the blocking factor implicitly defined by the user when maximum record size and number of home blocks are specified on the KYAN directive. The key analyzer does not attempt a distribution based on any implied blocking; it simply keeps count of the results of the hashing.

IXGEN UTILITY

The IXGEN utility can be used either to define alternate keys and an index file for an existing file that does not have them, or to define or modify alternate keys for an existing multiple index file. The data file must be indexed sequential, actual key, or direct. The IXGEN utility can only be called by control statements. It reads directives from a file and creates or modifies an index file according to those directives and the contents of the data file.

CONTROL STATEMENTS

The IXGEN control statement has the following format:

IXGEN(lfndata,lfndir)

lfndata — Logical file name of the data file

lfndir — Logical file name of file containing directives. Optional; default is INPUT.

The SYSIO and COBOL libraries must be made available before the IXGEN utility is called. Either a LIBRARY control statement or an LDSET control statement can be used.

IXGEN requires a field length of at least 70000 . A larger field length will improve the efficiency of IXGEN.

A FILE control statement is also necessary to specify the file organization of the data file and to identify the index file. The FO parameter must be set to IS, AK, or DA; the XN parameter must be set to the index file name.

If the file named by the XN parameter does not exist, IXGEN will create a file; if it exists, IXGEN will modify it. In either case, the alternate key definitions are read from the directive file.

RMKDEF DIRECTIVES

Input to IXGEN is in the form of RMKDEF directives, found on the directive file. Each directive defines one alternate key.

RMKDEF(lfndata,rkw,rkp,kl,0,kf,ks,kg,kc)

Required parameters:

lfndata — Logical file name of the data file

rkw — Relative word within the data record where the key begins, counting the first word (ten 6-bit characters in each word) as 0

rkp — Relative position within the word defined by rkw where the key begins, counting 6-bit fields 0-9, left to right

kl — Length of key in characters, must be 1-255

0 — Required to mark position for reserved field

kf — Key format, may be any one of the following:

- 0 — Character string, similar to KT=S for indexed sequential files
- 1 — Signed binary, similar to KT=F and KT=I for indexed sequential files. The length of the key must be a multiple of 6 bits
- 2 — Unsigned binary, described in COBOL as PICTURE 9. No existing file primary key will have this designation.

When a key is being deleted, kf is used to indicate the action, rather than the format of the key:

- 3 — Purge alternate key from index

Optional parameters:

ks — Structure for primary keylists for this alternate key may be either of the following:

- U — Unique; default
- I — Indexed sequential; recommended for efficiency in processing

If kg and kc are used, ks must be specified as U:

kg Length in characters of repeating group where key resides

kc Occurrences of group; must be used if kg is selected. kc is 0 if group defined by COBOL OCCURS DEPENDING ON clause.

When the index file is created, the first RMKDEF directive must define the primary key for the file. Each alternate key must also be defined with a directive. Normally, one RMKDEF directive defines one key; however, when the alternate key exists within a repeating group such as the trailer item of a T type record, as many keys are defined as there are occurrences of the group.

INDEX

COMMENT SHEET

CONTROL DATA CORPORATION

TITLE: CYBER Record Manager Guide for COBOL

PUBLICATION NO. 60496000 **REVISION** A

This form is not intended to be used as an order blank. Control Data Corporation solicits your comments about this manual with a view to improving its usefulness in later editions.

Applications for which you use this manual.

Do you find it adequate for your purpose?

What improvements to this manual do you recommend to better serve your purpose?

Note specific errors discovered (please include page number reference).

CUT ON THIS LINE

General comments:

FROM NAME: _____ POSITION: _____

COMPANY NAME: _____

ADDRESS: _____

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

FOLD ON DOTTED LINES AND TAPE

TAPE

TAPE

FOLD

FOLD

FIRST CLASS
PERMIT NO. 8241

MINNEAPOLIS, MINN.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION
Publications and Graphics Division
215 Moffett Park Drive
Sunnyvale, California 94086

CUT ON THIS LINE

FOLD

FOLD

TAPE

TAPE