PHILIPS

PTS 6800 TERMINAL SYSTEM

User Library

PTS 6800 DATA MANAGEMENT

Module M07

Data
Systems

Date : May 1978

Code : 5122 993 45131

MANUAL STATUS SURVEY

Module M07 "PTS 6800 DATA MANAGEMENT"

This issue comprises following updates :

– U1.45131.0479 (April 1979, Release 9.1)

PREFACE

This Manual is intended for programmers who have knowledge of the Assembler or CREDIT programming languages.

It describes the system of Data Management used by the PTS 6800 Terminal System in terms that apply to both Assembler and CREDIT. For the detailed information relating to the use of Data Management in either of these languages the reader is referred to the language reference manuals after reading this manual. These manuals are respectively M06, The Assembler Programmer's Reference Manual, and M04, The CREDIT Programmer's Reference Manual. In addition, the reader will need to refer to manual M08, the TOSS Utilities Reference Manual for detailed instructions on creating files and setting up file structures. This data management manual shows how the various utilities are used as steps in the process of creating file structures while M08 defines the precise operating instructions for each utility program.

1. INTRODUCTION

The PTS 6800 System provides a means of handling transactions at the time they are performed, and processing the required information immediately. It is an 'on-line system' and as such demands high standards of handling and processing data. On a batch system time can be taken to correct data errors, but for on-line working the source of the data, the transaction at the workstation, cannot always be recalled to repeat the transaction.

Data files are mostly required for immediate on-line access and the program must be able to find the required information in as short a time as possible. The same file may be accessed by more than one application at the same time, and each application may be making its own changes to the file.

To ease the programmer's task of manipulating data for different purposes, the PTS 6800 system has been supplied with a system of data management. This manual describes this data management system, how to set up the file structures for various purposes, and how to access the data in those files.

This manual has been written for both the Assembler programmer and the CREDIT programmer and does not contain information specific to those languages – it describes *data management as seen by both languages.*

Chapters 2 and 3 describe the general aspects and requirements of any system of managing data and may be familiar to the programmer experienced in the field of on-line processing. Nevertheless readers are advised to read these chapters first in order to understand the PTS viewpoint on familiar subjects like files, volume organisation, record handling, and security aspects.

Chapter 4 et seq. give a detailed explanation of the different kinds of file, how to create them and what instructions can be used for handling records in those files.

The detailed format of instructions in Assembler or CREDIT are explained in manuals M06, Assembler Programmer's Reference Manual and M04, CREDIT Programmer's Reference Manual. For the detailed instructions on operating each of the TOSS utility programs used in this Manual, the reader is referred to M08, the TOSS Utilities Reference Manual, or, in the case of certain utilities, to M11, the DOS6800 Reference Manual (TOSSUT utility).

2. PRINCIPAL ASPECTS OF DATA MANAGEMENT

This chapter describes the characteristics of data and files as seen by the system. Although much of this will be familiar to the experienced programmer he should nevertheless read this chapter before using the rest of the manual in order to understand how the terminology and concepts of data and files are used on the PTS system.

2.1 Files

When using the word *file* it is assumed that the data collected in a file is recorded in such a way that it can be read by a machine. This restriction places a file within the field of automatic data processing, within the limits of a computer program.
Thus, a file is a machine-accessible collection of data which should be organized logically according to the accessibility requirements of the separate data elements and the overall collection.

A file consists of a number of *records*, each record giving information on a specific subject. In an account file each record describes one account. A record consists of a number of data items. The current balance, for example, is one of the data items in an account record. In a program the data items of a record can be given names so that they can be referred to and processed.

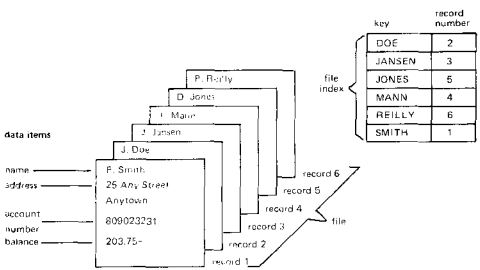


Figure 1.1 Composition of a file (with an optional index)

2.1.1 *Use of data files*

Data processing, at least in business applications, is very often a matter of routine. Files give a permanent description of some aspect of the business environment. As long as the information system manages to keep this description up-to-date, efficient processing may be done on the basis of this description.

Some processing only uses data stored in permanent files. If a user wants to know the current balance of an account holder, it is sufficient to select the amount from the relevant file, assuming that the file has been kept up-to-date. Keeping files up-to-date implies that changes must be made as a result of some calculation (an amount paid in is added to the account balance), or to some permanent item (the account holder's address).

Thus two types of data may be distinguished:
- input data
- permanent data (note: the existence of such a data item is permanent; the value of the item may be changed).

The new data could be input to the system in one of two ways:
- it could be gathered into input files from one or more sources and presented to the system as a batch of changes to be processed at one time (batch processing).
- it could be input via a terminal during the actual transaction that generates the change (on-line transaction processing).

2.1.2 *Retrieving data*

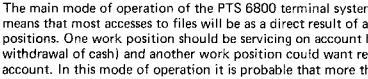
When processsing a file, data will be retrieved from the file, i.e. specific units of data are separated from the file and considered by the processing system. If necessary, the data is modified and rewritten to the file. For instance, when updating an account record it will be retrieved from the file and, after modification, be rewritten. Sometimes several data items of the same record must be considered and if necessary, updated at the same time, e.g. a woman account holder gets married and changes her surname and address.

When serving a customer, his account record will be required. Since the correct account record must be located, the record must have some identification in the form of a unique name or number which can be reffered to during the transaction.
For this purpose "keys" are used. When the key is known, the corresponding record can be retrieved from the file. The account number could be the key to the customer's account (this item is also a data item of the record required). Such an item is called the record key. Another example of a record key is the account holder's name.
The kind of key used is important when considering the methods of accessing the record in the file and possibly the organisation of the file.

In the case of a numeric data item such as the account number, the file can be constructed in such an order as to ensure that the last few digits of the number represent the record's actual position in the file. Account number 80803237 could be the 3237th record relative to the beginning of the file. This record key is called the 'logical record number'.
However, if the file is ordered according to the alphabetic order of account holders' names some other kind of key must be used because there can be no numerical relationship between alphabetic data items. The record key must be cross-referenced by an index to give the logical position of the record on the file.
Thus it can be seen that two kinds of record key exist:
- the logical record number that permits direct access to the record
- an index that provides a cross-reference to the record on the disk.

2.1.3 *Accessibility of Files*

The main mode of operation of the PTS 6800 terminal system is 'on-line processing'. This means that most accesses to files will be as a direct result of a request from one of the work positions. One work position should be servicing on account holder's transaction (deposit or withdrawal of cash) and another work position could want reports on the current status of account. In this mode of operation it is probable that more than one task wants access to the same file at the same time. Each of these tasks can treat the data in an entirely different way.
Consider the following file composed of 10 accounts.



Task A0 wants only the data relevant to the present customer. Customers come into the office at random, so the data accesses would be random, for example the following sequence might occur:

7, 5, 8, 3, 4, 6, 2 . . . . . .

Task B0 wants a report on the present balance of each account, so would access the file sequentially in account number order:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Task C0 wants the present status of certain accounts accessed with the account holder's name. In this example, the system must have a cross-reference index between the name and the account record. Accesses for task C would appear thus:

F. Smith	J. Doe	J. Jansen	A. Mann	– index
↓	↓	↓	↓	
8	7	9	3	– record

The access is random (the order of requirements cannot be predetermined) and via an index.

Physically the data file is the same for all tasks. The system itself only sees one kind of file. However, each task sees a different logical relationship between records and as a file is organised according to the logical relationship between records, each task 'sees' a different file organisation.

It is important to note the distinction between access methods and types of file. There are only two methods of accessing a record:
– direct access (either the 'next' record from the current position or via a logical record number)
  indirect access (via an index).

File organisation is a logical concept that describes the relationship between record accesses as seen by the task. There are three kinds of logical file:

– sequential	(as seen by task B0)
– random	(as seen by task A0)
– indexed random	(as seen by task C0)

Each kind is described in more detail in section 2.1.7 and chapters 4, 5 and 6.

2.1.4 *File turnover and growth*

During the normal course of business, some accounts are closed and some are opened and the corresponding data in the accounts will be changed. Data is deleted and new data is inserted. These activities are called 'file turnover'. There may be no more data in the file than there was before. The file did not grow but part of it was deleted and somewhere else a comparable set of data was inserted.
File turnover places certain requirements on the construction of a file. For example, new records cannot be inserted in the free space left by the old records, although items within the record can be replaced by new values. Different tasks can use different rules to determine the validity of new data. A file of accounts organized by account holder's name could have problems if there are two account holders by the same name (like father and son, or coincidences with common names).

Associated with the problems of file turnover is that of file growth. The number of records must be specified at the time the file is created. Allowance must be made for an increase in the user's business by specifying enough empty records for the file to grow without needing re-organizing. The factors to be balanced are:
– availability of disk space for 'empty records'
– rate of growth of the file (especially in the initial set up of the installation when data is being transferred onto the system)

- time available for re-organising files (busy installations cannot afford to waste time on frequent 'housekeeping' jobs).
- nuisance factor to the operator (if files are constructed without enough room for growth, they may cause overflows requiring frequent re-organization).

2.1.5 *File Organization*

The PTS system recognizes three kinds of file organization:
- standard files (type S)
- library files (type L)
- non-standard files (type X)

A standard file is one that has been formatted under the TOSS system for use by an application that operates under TOSS. All the files described in this manual, are 'standard files' and all these files are understood to contain data, or index records.

A library file is one that contains program coding in one form or another, i.e. as source, intermediate object or loadable form. This kind of file is outside of the scope of this manual and the reader should refer to manuel M11, the DOS System Reference Manual, or M08, the TOSS Utilities Reference Manual.

A non-standard file is one that is either unformatted, or from a computer system that uses different labelling and formatting standards. This kind of file is outside the scope of this manual and the reader should refer to manual M11, the DOS System Reference Manual, or M08, the TOSS Utilities Reference Manual, in order to process this kind of file.

2.1.6 *File categories*

All files used by the Data Management instructions can be divided into the following 'file categories':

a. *Data files* – these contain information that is processed by the tasks and could contain records about accounts, account holder's etc. A data file on its own can be used for sequential or random requests.

b. *Index files* – these contain a list of symbolic keys that are used to reference records in data files. The use of an index file considerably reduces search time for a record, especially if the record can be referenced by more than one key. Index files can be treated as a data file for updating purposes (sequential requests) or used as the index to a data file (indexed random requests).

c. *Master index files* – this is a 'summary' of the index file that is produced after the index file has been created. It reduces the time required to search an index file and is used by the system in conjunction with indexed random requests. The master index file is held in memory after its relevant index file has been assigned to a task.

With indexed requests, these different files are related by pointers. A data file can be associated with more than one set of index/master index files but these latter cannot exist without a data file. A set of data, index and master files constitute a 'file structure'. A file structure can only contain *one* data file.

2.1.7 *Data File Organization*

Data files can be organized in one of three ways:

a. *sequential* – the file is created and accessed in such a way that records are processed serially.

b. *random* – there is no relationship between records and they are required randomly. Each record is accessed by its position relative to the beginning of the file via its logical record number.

c. *indexed random* – records are accessed via a key that is contained in an 'index file'.

The kind of organization used depends basically on the use of the file. A sequential file is used where actions are always carried out in a sequential order, for example list processing, reporting on the state of accounts, logging various activities on-line as they occur. Sequential files are more often seen in batch processing, but could be used for on-line processing in some circumstances (for example log files, see chapter 3.2.

A random file is used when the accesses are happening at random times and to randomly required records. This kind of file organization is more often seen in on-line processing where the accesses are coming from any number of terminals dealing with randomly occurring events, for example, customers walking into a bank to deposit or withdraw cash from their accounts. It would be nonsense to expect the customers to visit the back in alphabetical order so the files must be organised to allow records to be accessed directly – one of the data items in the record must be used to indicate the record's position in the file relative to the start of the file.

The usefulness of a random file is limited if records are to be accessed by a choice of items, for example, an account file could be accessed by either the account number or the account-holder's name. If this is the case then all the items used for access (the keys) must be set up in a cross-reference file (the index) that gives the logical record number related to all the keys. This method of organization, a data file with an index, is called indexed random.

Each kind of data file organization is described in the following chapters.

2.2 Volume Organization

A volume is a single physical unit capable of holding information.
For the purpose of this manual this is understood to be:
– a removable disk cartridge
– a fixed disk
– a flexible disk

2.2.1 *Disk Structure*

Each disk volume is divided into cylinders, each cylinder into tracks, and each track into sectors. The user program does not use this structure as it only addresses records within a file. The programmer must be aware of this structure when constructing files as it could affect the blocking factor of records within blocks, number of file extents in the volume, or number of volumes required for one large file. The structure of the PTS 6875/76 disk is shown in the figure below.

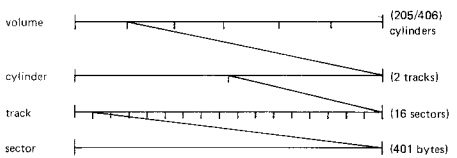


Figure 1.3 Disk Structure, PTS 6875/76

Each sector can be subdivided into records according to the program's requirements. The number of records stored in each sector is called the blocking factor and (record size X the blocking factor) should never exceed 401 bytes. The largest record allowed is 400 bytes + 1 status byte used by the system. Every record in the file has one byte reserved for system purposes so a record of 80 data bytes must be created as a record with the length 81. When accessing the record, the program uses the length '80'. Thus only four records could be blocked onto one sector (5 x 81 = 405, is too large). If the record length could be reduced to 79 data bytes + 1 status byte then five records could be blocked onto one sector, making more efficient use of disk space.

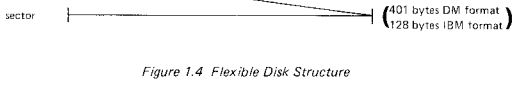


Figure 1.4 Flexible Disk Structure

disk model	*Cylinders per volume*	*Tracks per cylinder*	*Sectors per track*	*bytes per sector available to the user*
PT8 2 x 2¼ M	203	2	16	401
PTS 6876 2 x 5 M	406	2	16	401
		tracks per volume		
PTS 6879 (flexible disk)	—	77	6½ 26	401 DM format 128 IBM format

Table 1.1 Disk Capacity Available to User Programs

Some of the space on both disks and flexible disks is reserved for system use and so can never be accessed by user programs, see the table below.

disk model	*system-reserved areas*	*Purpose*
PTS 6875 }	cylinder 00, track 00, Sector 00	Volume Label
PTS 6876 }	cylinder 00, track 00, Sector 01	Initial Program Loader (IPL)
PTS 6875	cylinder 200 203	system use
PTS 6876	cylinder 406 407	system use
flexible disk (PTS 6879)	track 00, Sectors 01-04 (128 byte sectors	Volume Label
	track 00, Sectors 05-08 (128 byte sectors	IPL

Table 2 Reserved Areas

Note that both disks and flexible disks will have variable amounts of space allocated to the Volume Table of Contents (VTOC) depending upon the number of file extents.

The VTOC contains one record of 41 bytes for every file extent on the volume. VTOC records are blocked 9 per sector so simply divide the number of file extents by 9 to find the number of reserved sectors.

The VTOC is accessible only through Assembly routines, so the reader is referred to the Assembler Programmer's Reference Manual.

2.2.2 *Creating a volume*

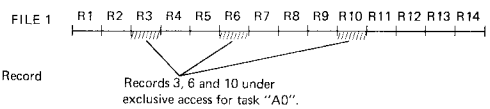
A disk volume cannot be used on the PTS6800 system until it has been initialized and formatted by the Create Volume utility (CRV). A full description is available in the Utilities Reference Manual, M08 an in M11 DOS6800 Reference Manual (TOSSUT Utility), but is mentioned briefly here. CRV writes a volume label and an empty VTOC, then writes cylinder identifiers in all sectors. This identifier is outside of the area of the sector that is available to the user. CRV also performs a quality test on each sector by writing then reading back. If any defective sectors are found CRV creates a dummy file called 'BADSPOT' and assigns all unusable sectors to that file. IPL is written to the disk and CRV terminates. The volume is then available for use, unless any badspots are located in the area reserved for the volume label on IPL.

2.3 Record handling

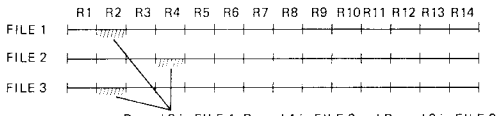
2.3.1 *Exclusive access*

Data files may be shared by a number of tasks, so simultanuous updating of records must be prevented. Exclusive access is a function which is used to prevent simultaneous updating or records. The exclusive access function for use by the user program is included at system generation time, but is superfluous when only one task exists using data management. However, the user still has the possibility to allow exclusive access setting for a record as an option in the instruction (assuming that exclusive access was included at system generation time).
Exclusive access is controlled on record level, which means that individual records can be held under exclusive access (no other task can get these records) but not the whole file. In this way a task may have in one file more than one record under exclusive access and the task can have records under exclusive access in different files.
In one file different records can be under exclusive access for different tasks.

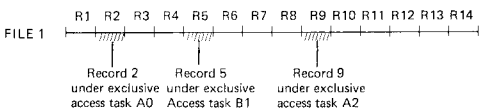
Example 1



Example 2



Example 3



Exclusive access is not used for index files, only for data files.

A record can be set under exclusive access, after an unsuccessful read operation. Exclusive access is released after a (re)write, delete or release exclusive access operation of the record has been performed.
When a record is accessed which is already under exclusive access by another task, a status is returned indicating "record protected".

2.3.2 *Current record number*

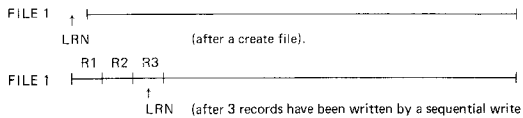
Data mangement has an internal handling of Current Record Number (CRN). For each file structure (files consisting of one data file and possibly index and master index files) an area is reserved per task in the system by data management, in which the CRN from the last request on this file is stored for each task.
Some instructions use this CRN value before execution to obtain the next record. After execution of the instructions the CRN may be updated by data management, depending on the type of instruction (see table 2.3 below). This current record number can be obtained by the user with the command 'get currency index' or 'get currency data'. In this case the last accessed record of the index and data files, respectively, is returned to the user for this task.

Instruction	*CRN used for execution of the instruction. Affected on file type:*	*CRN updated after execution of the instruction. Affected on file type:*
Sequential Read	Data file	Data file
Sequential Write	–	–
Random Read	–	Data file
Random Write	–	Data file
Random Delete	–	–
Indexed Read	–	Data file + Index file
Indexed Rewrite		Data file
Indexed Delete	–	
Indexed Insert	–	Data file + Index file
Indexed Read Next	Index file	Data file + Index file

Table 2.3 Current record number handling

2.3.3 *Last record number*

Data management holds per file (not per task) a last record number pointer (LRN) indicating until which record the file is filled. The user cannot access this pointer, but can be informed by means of an error return code, or via the control word after a sequential write.
When a file is created by the utility Create File, the last record number pointer is always set to the beginning of that file. After a sequential write the pointer is updated every time a record is written. The sequential write and Indexed Insert instructions will influence the updating of the last record number pointer. However, the LRN is not put onto the disk until the file has been closed.



By the indexed and random instructions the user is able to read or write records located after the last record number pointer. However, an error code is returned indicating an "End of File" condition, but the I/O operation is *not* aborted. It is *up to the user* to decide whether an action has to be taken or not. When a sequential read results in an error code with the "End of file" bit set, the I/O operation *will be* aborted. Note that, when a file is being processed a difference in the value of the last record number pointer may exist between the one on disk and the one updated in memory. After the file is closed this updated last record number pointer is saved on disk.

3. FILE INTEGRITY AND SECURITY

All the subjects and techniques described in this manual assume that processing is proceeding without any interruption or errors. This of course is an ideal situation and one that will hopefully continue. It is also an unrealistic situation because at some time, no matter how infrequently, an error may occur that could corrupt one or more items of data, possibly the whole file. The programmer must therefore write his application to ensure that the *integrity* of a file is maintained under all circumstances, regardless of the source of the error. File integrity means that the file contains valid, meaningful, and usable data. If the application is batch processing the file then the effects are only damaging in the sense that time can be taken to reconstruct the file before it is required again. To achieve this reconstruction can be relatively simple. If the application is using the file for on-line processing then the problems of an error become far more serious. A file is required to be available immediately and one or more work stations are thereby prevented by the error from performing any transactions.

From this it can be seen that there are two areas of file use that must be supported in the event of corruption:

- files used for batch processing applications
- files used for on-line processing.

The latter includes files used for batch *and* on-line processing. To maintain the integrity of a file, it must be made *secure*. As previously stated in this manual, a file is a logical concept in that it is a body of information used during the processing of an application. The physical representation of that information is not important from the point of view of the application that uses it. If the medium on which the file is held is irrecoverably damaged it does not matter to the application *as long as an exact copy of that file is available*. A file can be made physically secure by ensuring that a copy exists, either an the same storage medium or on an alternative medium if the file can be easily copied to the original medium. An on-line file requires actions to be taken within the application.

The different aspects of file security, and file integrity, are described in this chapter for both batch and on-line files. Note that the subject of *data security* is considered to be completely application dependant – the program must check that all data handled is valid and must secure the confidentiality of data according to the requirements of that application.

3.1 Securing batch files

Security of files used for batch processing is simple to achieve if the following system is adopted. The major requirement for any security system is to ensure that a complete copy of a file is available if the working copy is corrupted or destroyed. It would be expensive to duplicate all disk files on other disks so cassette, magnetic tape, or flexible disk can be used as a back-up volume.

Suppose we have a file called DKFILE on disk that we wish to secure (see figure 3.1). After the file has been set up (stage 1) we make a copy of the file (stage 2a – this process is usually called 'dumping' the file.). DKFILE is now available for the batch application.

At stage 3 DKFILE is required by the application. There are two main sources of damage to DKFILE that we can consider:

- internal, i.e. damage that occurs as a result of the application, system error, hardware faults, operator errors.
- external, i.e. damage to the medium away from the computer such as mishandling by the operator or environmental damage (fire, left in direct sunlight, or magnetic fields).

When the damage has been discovered, copy the file from the security volume (stage 4) onto the same disk or onto another disk of the original was damaged physically. If the damage was environmental, it is possible that the security copy has been damaged as well. To avoid this, the security copy should be stored apart from the working copy, or make another copy (stage 2b) and store this in another room or building.

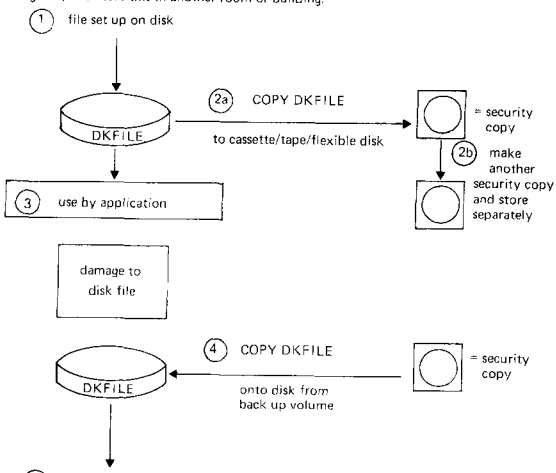


Figure 3.1 Simple security system for batch files

This is of course a very simple situation because (apart from listing, reporting or statistical applications) the application will probably change some of the data in the file. If this is the case, then the security copy will no longer represent a true copy of the disk file. Every time there is a change to DKFILE (stage 5), a copy must be made, see figure 3.2.

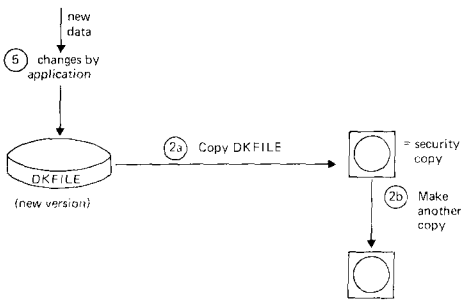


Figure 3.2 Updating the file and its security copy

It would appear that the disk file is now completely secure against corruption or damage because a copy of the latest version is always available. There is still a source of corruption that has not been allowed for, and that is the new data itself. If one or more of the updating records were wrong it might not be discovered until the updated records on DKFILE were used in later processing. This could be prevented by keeping generations of the file. For all practical purposes it is only necessary to keep three generations of the file (including the latest) plus the changes that updated each generation into the next. This method of file security is known conventionally as 'grandfather, father, and son', or gfs, and is maintained as follows, (refer also to figure 3.3):

1. DKFILE has been set up (version 1) and a security copy exists (one or two copies as required, A1 and B1 respectively).
2. During the next run of the application, changes are made to existing records, new records are added, or old records deleted (C1). The security copies no longer represent the latest version (version2) of DKFILE so new security copies must be made, A2 and B1. It is debatable whether a new B level volume is required, so three A level volumes plus one B level will give adequate levels of security. If a large number of changes are made frequently there is a case for using three B level volumes. For the purpose of this discussion only one B level is shown.
3. DKFILE can be reconstructed on disk by copying from A2 (or B1). If the changes were proven to be corrupt (wrong records deleted or changed) then DKFILE could be reconstructed by copying version 1 to disk from volume A1 and running the application with the correct changes. There is still the remote chance that the latest version on disk was copied incorrectly to A2. If DKFILE were then to be corrupted it could be reconstructed by copying the previous version to disk from A1 then updating the disk file from C1 to make the latest version of the file. When the previous generation and its associated

changes must be saved. This will ensure against practically all sources of corruption.

4. To achieve three generations of security, the process of copying is continued at each stage until there are three A level (and B level if required) volumes.
5. When the fourth version of DKFILE is generated (with changes C3) the new version (4) is copied to A1, or to a new volume A4 (and A1 then released). Thus we have 3 generations of security, or two in the remote event of corruption occurring during the process of generating A4.

Once set up, this system becomes completely automatic and can be performed either through the application or as part of the operator's 'housekeeping' jobs on the system. The decision to 'secure' any particular file must be made at the system design stage of the application. If DKFILE were only a transient file to be passed to another part of the application, or between applications, it would be unnecessary to maintain this kind of security.

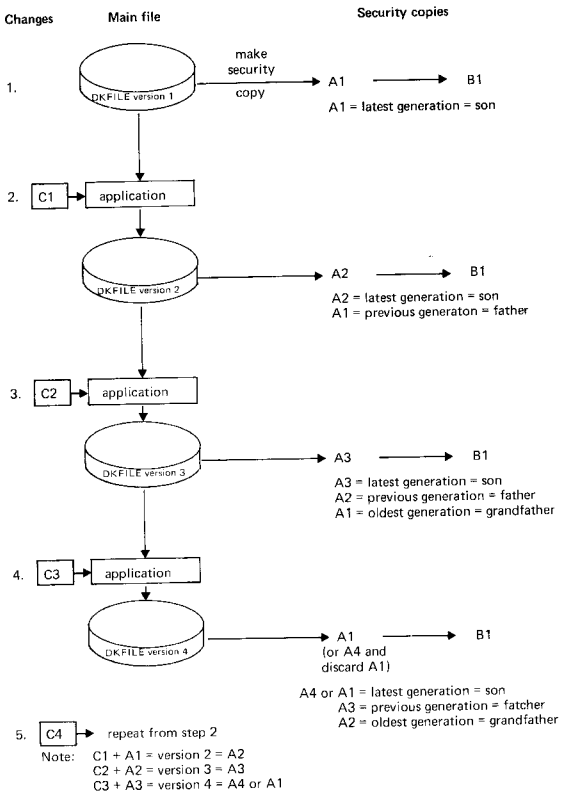


Figure 3.3 Full security system for batch files

3.2 Securing on-line files

Files that are being used by an on-line application present a more difficult situation from the viewpoint of security. The files are required to be available all the time the application is running. If the work stations and the application are providing a service to the public, like a bank, tax office, or local authority office, we can hardly expect people to come back later when a corrupted file has been reconstituted. The application must be able to provide the same service in as short a time as possible after an interruption.
Consider the situation where an application is providing a service at a number of work stations, see figure 3.4. All work stations have access to the file DKFILE through the application-records can be accessed for information, for changing, for addition to the file or for deletion.

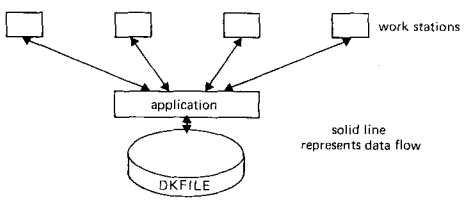


Figure 3.4 Multiple access to an on-line file

The first consideration when an on-line file is corrupted is to ensure a continuation of the service. A copy of the file must be available in as short a time as possible. Note that the original on-line file may have been updated only seconds before the corruption occurred or was discovered. How do we ensure that those changes exist on the back-up copy of the file?

3.2.1 *Duplicated on-line files*

If the installation had a limitless budget allocation one could afford the luxury of two disk units with identical files mounted – a change to one would be duplicated on the other. For most situations this is not possible although it does feature in systems where the immediate availibility of information is absolutely critical, see Figure 3.5. If the application detects corruption of some kind in one copy of the file it still has access to the other and can continue providing a service at the work stations. Where an alternative volume is mounted, the application could activate a subordinate task to create a file with the name of the corrupted file (in this example DKFLE1) then copy from the uncorrupted version. This would provide an uninterrupted service and only the operator at the computer would know that anything untoward had happened.

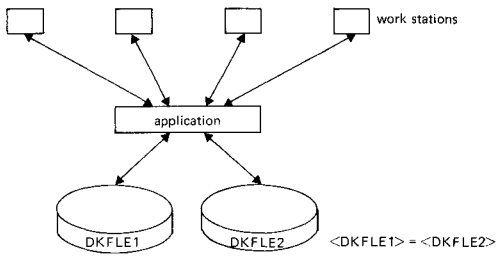
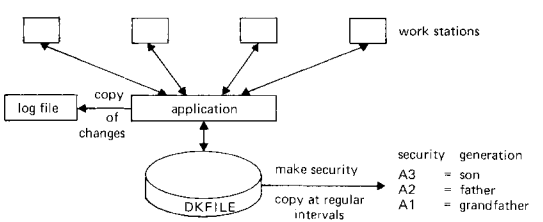


Figure 3.5 Duplicated on-line files

3.2.2 *Logging the changes*

Duplicating on-line files is an expensive solution to the problem of maintaining the availibility of information. There is a cheaper way of doing this that is similar to the method of securing batch files. It features both a back-up of the disk file and a file contaning the changes to the disk file since it was 'dumped'. The difference is that the changes to the file are recorded *as they are made*, that is, all changes are sent to a log file on cassette, magnetic tape, or flexible disk. It is not necessary to have one log file for every disk file as each record copied to the log could have an indicator or identifier attached to it showing which disk file it belonged to.

At certain intervals, decided at the system design stage, the on-line file is dumped to a security volume. The decision to dump a file could be taken at every 1 000th update, every hour or the end of the day depending upon the requirements of the application. The security volumes can be kept in the same way as those for batch files, that is, using three generations in a rotating gfs system. Figure 3.6 illustrates this situation.



In the event of corruption to DKFILE, it could be reconstituted by a subordinate task which copies from the latest generation security volume and merges this with the records belonging to DKFILE that have been logged to the log file. The result is a new copy of DKFILE complete with all the latest changes. The application could then be back on-line within minutes of the corruption being discovered.

3.2.3 *'Graceful degration'*

From the point of view of the work stations, is there any reason why the application should go 'off-the-air' at all? If the installation is giving a service to the public, is it right to make people wait because some part of the system has sustained damage? If the corruption is due to physical damage to a disk unit it could be sometime before full service can be restored. The fact that we are recording changes to the file on the log file implies that we can indeed continue to give a restricted version of the service. The application could continue to accept changes as a result of the individual transactions and send these changes to the log file. If any transaction wanted an inquiry only it would not be possible, because the disk file is unavailable due to corruption. This is all a matter for system design of course, but there is no reason why the application should not provide an alternative to the log file if that device should fail too. The important factor is the integrity of the file DKFILE and changes to the information that that file represents could be accepted as long as there is still a device operating at the central computer that can accept those changes. This method of offering a restricted service in the event of device malfunction or file corruption is known as 'letting the user down gently' or 'graceful degradation'.

4. SEQUENTIAL FILES

4.1 Description

A sequential file is one that is processed serially from the first record in file through to the last. All files can be processed sequentially regardless of any other use. A file that is used as an indexed random file could be used as a sequential file in another task for listing, reporting or statistical purposes. A log file containing changes to an on-line file is both generated and accessed later as a purely sequential file. This chapter discusses the characteristics and handling of 'sequential files' without regard to any other way of using them.

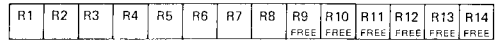


Figure 4.1 A sequential file

Consider the file shown in figure 4.1. For sequential processing all accesses to this file should be serial. When the file is opened the CRN is zero, and the LRN is R8, the last used record in the file. The sequential read instruction is issued and the first record in the file is read into the user's buffer. The CRN is 0 before the instruction and set to 1 after the instruction, thus pointing to the current record being processed. The next time a sequential read is issued, record R2 is read into the user's buffer and the CRN is updated to 2. The instruction makes no reference to a record number or to any particular record, it can only access the record after the one pointed to by the CRN.

The sequential write instruction acts in a slightly different manner. If a new record was being added to the file, it would be placed after R8 and the LRN would be updated in memory to R9, and so on for as many new records were being added to the file. The CRN is not used or updated by the sequential write instruction.

Suppose the file was to be updated and that the change was to record R6. The sequence of instructions for this are shown below:

1. Assign the file (CRN = 0, LRN = 8).
2. Read sequential until the relevant record is formed (CRN now = 6).
3. Process the data in the required record.
4. Sequential write the changed record. This will go to position 9 and the LRN is updated in memory to 9.
5. The 'old' version of the record still exists and will have to be deleted – 'get currency data file' is the instruction to access the CRN for the user program.
6. Random delete using the CRN as the logical record number. R6 is now set to FREE.
7. Close the file. The LRN will be updated on disk to 9.

If the file being processed is normally used as a random file or indexed random file, then *great care must be taken to ensure that the order of records is not changed*. Problems could arise in later processing when the record is being accessed by its logical record number.

4.2. Creating a sequential file

The rest of this chapter assumes that the file is being created specifically for the purpose of sequential processing. If it is to be used for random or indexed random processing refer to the following chapters.

The file can only be created on a TOSS formatted disk, i.e. one that has been created by the utility Create Volume. The creation of the file must be performed in two stages. Firstly, the file space is set up by the utility program Create File (CRF); see the Utilities Reference Manual, M08, or the TOSSUT utility in M11 DOS6800 Reference Manual, for a detailed description. Secondly, the actual records must be written to the file. A sequential file could contain records that are present in a predetermined sequence and any operations on the file should preferably be performed in a sequential manner.

1st Stage

The sequence of operations for CRF is as follows:

1. Call CRF utility under the TOSS utilities Monitor, or as a subroutine by the application, or via the DOS utility TOSSUT.
2. CRF requests a number of parameters. Most parameters can be given as required but for a sequential file, two parameters are obligatory.
   To 'File organization' reply 'S', and to 'Number of index files' reply '0'.
3. CRF searches the volume(s) for free extents large enough to hold the stated file size.
4. The file is set up with the required number of records, all of which contain space characters. Each record is set to 'FREE' status. The LRN is set to zero for this file.

2nd Stage

The actual records must be written to the file. For a purely sequential file, the records may be required in an order determined by the value of one of the data items. If the records have been prepared off-line on punched cards, or cassette, it is possible they may not be in the 'correct' order, so the input file must be sorted according to the required key before the file is released for use, for details of the SORT utility see the Utilities Reference Manual, M08.

When the file is available for use, it can be processed with the sequential instructions or with the random instructions presuming that the record key used for random access can be directly related to the record's position in the file. The sequence of records in the file could be determined alphabetically like a name and address file but this key could not be reduced to a numerical value to give a logical record number. If the file is required to be accessed by a task using the indexed random file, it must be set up according to the instructions described in Chapter 4. After set up, the file can then be used for sequential processes as well.

4.3 Instructions

A definitive description of these instructions is contained in the relevant language reference manuals, either Assembler Programmer's Reference Manual, M06, or CREDIT Programmer's Reference Manual, M04.
These instructions are briefly:

ASSIGNING THE FILE

When the file is 'assigned', the file name is linked to a file code declared in the same task. If the file is to be used only by that task it must be assigned with the TC parameter = 1, or if the file is to be accessible by more than one task it must be assigned as a common file, with TC = 0.
After successful assignment, the file is available to the assigning task.

SEQUENTIAL READ

After the file has been assigned, the CRN for this file in this task is set to zero. Whenever this instruction is used the CRN is updated by one and that record number is read. It is not possible to specify a particular record or a previous record. The current record can be put under exclusive access if required. If the record is already under exclusive access to another task it will not be read.

SEQUENTIAL WRITE

The record will be written to the file immediately after the record pointed to by LRN. The LRN will be incremented by one to the last record written by sequential write.
If a system crash occurs during the run of this task, the new LRN will be lost but the new records still exist. These records could be read with a random read, deleted, then rewritten with a sequential write. The LRN will then be correct.
If the record status is 'free' it will be set to 'used'. If it is 'used' sequential write is not allowed. Random delete, see chapter 3 will have to be used to delete a 'used' record.
The CRN is not changed by sequential write.

CLOSING THE FILE

A 'close file' instruction is used to indicate that the file is no longer required by that task, the LRN is updated and saved on the volume. The close action applies only to the task that issued it, and the file is still available to other tasks that are using it at that time, unless the file was assigned as common, in which case the file is no longer available to any tasks.

5. RANDOM FILES

5.1 Description

It has already been stated that a 'random' file comprises records that can be required at random – there is no way of predetermining the next record to be accessed from the last record accessed. The system needs some kind of reference to each record so that the record can be found and read when the user requires it. For this purpose, a 'logical record number' is used that identifies the record's position relative to the beginning of the file. This implies that the file is normally created as a *sequential file with sequence determined* by the value of one item in the record. The last few digits of an account number, for example, could be used as a logical record number as long as there is a direct relationship between the value of that part of the field and the position of the record relative to the beginning of the file (account number 0132, must be the 132nd record in the file).
An example sequence of events could be as follows:

1. A file is created using the CRF utility (see manual M08, The Utilities Reference Manual or the TOSSUT utility in M11 DOS6800 Reference Manual) to contain records about accounts.
2. Account records are written to the file *sequentially* in account-number order.
3. When the system is on-line the keyboard operator inputs an account number.
4. The program decodes the account number to produce a 'logical record number'.
5. The 'random read' instruction is executed and the Data Management routine takes the logical record number and calculates the physical position of the record on the disk. (It knows the address of the first sector in the file and the blocking factor so it can find the relevant sector by dividing the logical record number by the blocking factor).
6. The record is read directly (and placed under exclusive access) and made available to the task.

Note that when the file is assigned, the CRN is always set to zero. After subsequent accesses, the CRN is set to the last record that was accessed.

It is possible that records do not contain an item that is increasing by one for each record in the file. This does not matter as long as there is a direct numerical relationship between the item and the record's position in the file.
If a file is required that contains no numerical relationship between records then the programmer has the choice of inventing a new field in the record that can contain a logical record number, or better, to use the *indexed* random method described in chapter 6.

5.2 Creating a Random File

The file can only be created on a TOSS formatted disk, i.e. one that has been created by the utility Create Volume (CRV). The creation of the file must be performed in two stages, firstly, the file space is set up by the utility program Create File (CRF); see the Utilities Reference Manual, M08, or the TOSSUT utility in M11 DOS6800 Reference Manual for a detailed description, then the actual records must be written to the file.

1st Stage

The sequence of operations for CRF is as follows:

1. Call CRF utility under the TOSS utilities Monitor, or as a subroutine if required by the application, or via the TOSSUT utility under DOS
2. CRF requests a number of parameters. Most parameters can be given as required, but for a random file, two parameters are obligatory. To 'File Organisation' give 'S', and to 'Number of index files' give '0'.
3. CRF searches the volume(s) for free extents large enough to hold the stated file size.
4. The file is created with the required number of records, all of which contain space characters.
   Each record is set to 'FREE' status. The LRN is set to zero for this file.

2nd Stage

It is possible to write records with random write instructions at this point but on a subsequent read instruction the LRN will still be zero and an error will be returned stating End-of-File. The actual records must be written to the file using the sequential write instruction so that the LRN will then be set to the last used record in the file. The operations for this second stage depend upon the source of the records. If the records already exist, for example a bank putting its account records onto the computer, then the records could be copied into the 'empty' file on the disk. If it is a new system, then the account numbers could be written to predetermined positions on each record. Each account record could then have the rest of the information set up at the time account numbers are allocated to customers.
Remember when setting up the records that the logical record number will be related to the record's position on the file. The records must be written, therefore, in the order determined by the relationship between the record key to be used and the logical record number. (Account Numbers allocated as 1, 2, 3, 4, etc. should be set up as the 1st, 2nd, 3rd, 4th, etc. records in the ifle. If account number are allocated by tens, i.e. 10, 20, 30, 40 etc. these will still be logical record numbers 1, 2, 3, 4 etc. so provision must be made in the using task to reduce the input key to the relevant logical record number.)

The file can now be used as a random file, see the instructions in the following paragraphs, or as a sequential file, see Chapter 2. If another task is going to use this file as an indexed random file, then this file should be created according to the instructions in Chapter 4. After set up, the file can be used for random, as well as indexed random, processes.

5.3 Instructions

A definitive description of these instructions is contained in the relavant language reference manuals, either *Assembler Programmer's Reference Manual*, M06, or CREDIT Programmer's Reference Manual, M04.
These instructions are, briefly:

ASSIGNING THE FILE

When the file is 'assigned', the filename is linked to a file code declared in the same task. If the file is to be used only by that task it must be assigned with TC = 1. If the file is to be accessible by more than one task, it must be assigned as a common file, with TC = 0. After successful assignment, the file is available to the assigning task for random read, random write and random delete, the record being accessed by its logical record number, see section 3.2.

RANDOM READ

This instruction will allow the task to access the record by its logical record number, and to put the record under exclusive access. The CRN will be updated to the last record that was accessed.

RANDOM WRITE

The record to be written is specified by its logical record number and the CRN is updated to this record after a successful write. If the record on the file is 'free' it will be set to 'used' but if it is already 'used' it can only be written if the record is under exclusive access. This implies that if an item in a record is being changed, the record must be read by random read first. Exclusive access is released after a random write.

RANDOM DELETE

This instruction can only be used on a record, specified by its logical record number, that is under exclusive access. It sets the status character to 'free' then releases exclusive access. Note that in CREDIT, this instruction is effected through the Data Set Control 1 statement (DSC1).

CLOSING THE FILE

A 'close file' instruction is used to indicate that the file is no longer required by that task. The LRN is updated and saved on the volume. The close action applies only to the task that issues it, and the file is still available to other tasks that are using it at that time.

6. INDEXED RANDOM FILES

6.1 Description

Random files as described in the previous section are simple to set up and use. The limitations become apparent when records are referenced by non-serial items such as names, or the user wants to access the record via a number of different items such as name as well as account number.

In this case, an index must be created that provides a cross-reference between the item provided as the reference (the symbolic key) and the relevant record. These keys can be alphabetic such as names, alphanumeric such as encoded charge numbers, or numeric such as account numbers. Up to four keys can be used to reference the record. When a record is required, the instruction is supplied with the key and the system looks up the logical record number associated with the key. The symbolic keys are all held in a sequential file called an 'index file'. A summary of this file is created in order to reduce the search time for a record. This summary of the index file is called the 'master index', which is also a sequential file. The data file, index file and master index file constitute a 'file structure'.

6.1.1 *File structure*

The basic component of a file structure is a data file. Each file structure can contain *only one* data file. For every record in the data file, at least one item has been nominated as a key. The keys are gathered into an index file that has been sorted according to the pure binary form of the key and into ascending order. The index file is thus a *sequential file.* Each key in the index file is given the logical record number of the record it belongs to in the data file. The record key in an index file is called the 'symbolic key'.
The number of symbolic keys in the index file is obviously the same as the number of records in the data file. To search through the index could take a lot of time for a large file so the system divides the index into partitions. The highest value key in each partition is copied into a 'master index' with the lowest record number of that partition. This master index is also a sequential file. When a symbolic key is input at runtime, the master index is searched sequentially until there is a match or until the next highest value key is found. In either case the master index points to the partition that contains the symbolic key and hence the logical record number of the required record. Note that the master index is stored in memory while the index file is assigned.
The relationship between the data file, the index file and master index is shown in the following diagram.

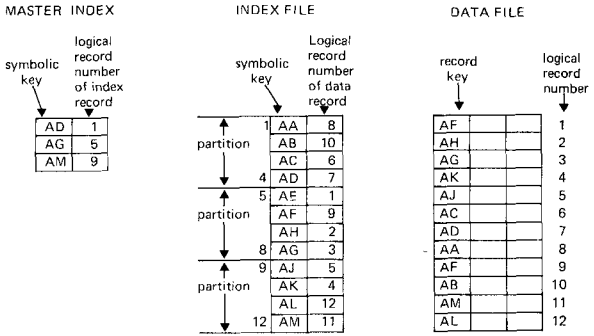


Figure 6.1 File Structure

Consider two examples of record access from a key supplied via the keyboard. First, key AG is supplied, then AL.

Example 1:

1. The operator inputs AG.
2. A sequential search is made of the master index until a match occurs, or the next highest value is found.
3. A match occurs stating that symbolic key AG is in the partition that starts with record 5.
4. A sequential search of partition 2 of the index file until a match occurs – this gives the logical record number 3 in the data file.
5. The record is accessed directly.

Example 2:

1. The operator inputs AL.
2. A sequential search is made of the master index until a match occurs or the next highest value is found.
3. AL does not exist in the master index but a higher value is encountered, i.e. AM, which gives the partition that starts with record 9.
4. A sequential search is made of partition 3 of the index file until a match occurs – this gives the logical record number 12 in the data file.
5. The record is accessed directly.

Thus the use of a master index as a summary of the index file, and the division of the index into partitions, gives considerable savings of time when searching for a record.

Any data item in the record can be used as the key and it is possible to use more than one data item for accessing the record. If a second (or third or fourth) item is required as a key then more index files must be created (and master index files), see figure 6.2 below.

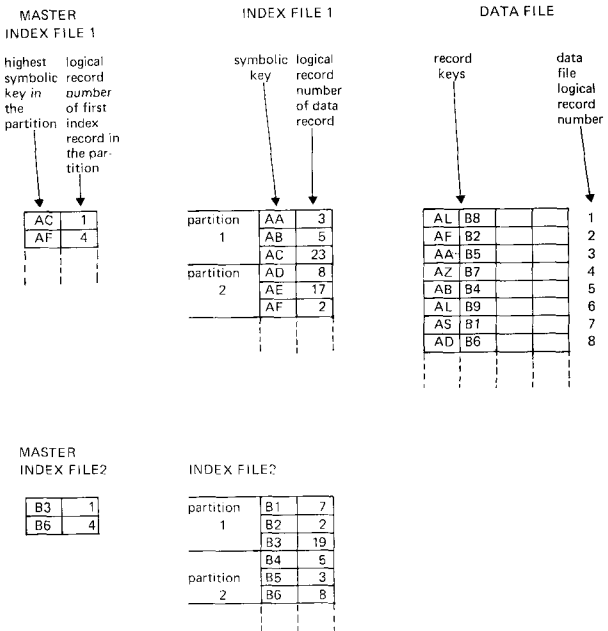


Figure 6.2 Data file with Two Sets or Index Files

6.1.2 *The Index file*

The index file is a sequential file and contains one symbolic key for each record in the data file and the logical record number of the data file record. It also contains a status indication for the record and an indication of duplicate characters in a set of symbolic keys.

Allowance must be made for the file to grow without the need for frequent reorganization. Just as a number of empty records are allowed for when creating the data file, so should empty records be present in the index file. The ratio of used to empty records is called the 'load factor' and is a parameter required for the Re-organize Index utility (RIX) which is fully described in the Utilities Reference Manual, M08. The number of empty records in the index file should be the same as that in the data file, but the empty records are put at the end of each block (i.e. sector) in the stated proportion. Consider an example data file of 100 records spaces. It is expected to grow to this size quickly but when the records are set up, only 50 data records are available. The index file is created with symbolic keys pointing to the existing records and not the empty ones. However, provision must be made in the index file for the data file to grow, so the index is created with a load factor 50%. In the data file, the empty records are positioned logically at the end of the file. The index is a sequential file and new symbolic keys must be inserted in the correct sequence so the spare index records are placed in each sector. The new key is inserted in the correct position and the following records in that sector are shifted along. Figure 6.3 summarizes this situation.

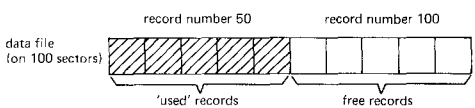


Figure 6.3a Illustration of Load factor-Data File

There is 50% utilization of record spaces after initialisation of the file, so the index is created with a load factor of 50%. A new record, logical record number 51 is placed at the end of the used area and the LRN updated.

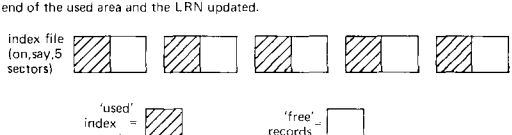


Figure 6.3b Illustration of Load factor-Index File

The index record for record 51 in the data file is placed in the correct position according to the symbolic key and the keys following in the same sector are shifted along. If the records overflow into the next sector, the records already contained in that sector are shifted along.

The format of the index record is as follows

field	length
1. Symbolic key	1-n characters
2. Dummy	2 characters
3. Duplicate key	1 character
3. Logical record number	3 characters

where:
'symbolic key' is the data item contained in the data file record that is used for identification of the record. It is left adjusted and padded with blanks if n is greater than the key used.
'Dummy' is not used.
'Duplicate key' is the binary value of the minimum number of leading characters in the key that is identical with the next symbolic key in the index file.
'Logical record number' is that of the record in the data file that is referred to.
Remember when calculating the blocking factor that one byte must be added to the record length for the status byte.

6.1.3 *The Master Index file*

The master index is used to reduce the time required to search the index file. This is created by the system during the utility Re organize Index (RIX). The size of the master index file is decided by the user and this will determine the number of partitions, and hence the number of master index file entries.
The master index file must reside on the same volume as the index and there can be a maximum of 16 master indexes in the system at the same time. When the index file is assigned to the data file by the application, the master index is completely read into memory. The size of the master index memory area has to be specified during system generation, and must be able to contain all master index files required simultaneously plus three words.

The optimum size for partitions should be related to the physical storage of the index file on the disk. That is, the master index file should be constructed to minimize disk head movements when searching for index records. The system reads a whole sector at a time even through only one record is being accessed. If the index file occupies, for example, 5 sectors then the master index file could be created with 5 records. Each master index entry would describe one sector (= 1 partition). The key value is the last record in the sector/partition (highest key value) and the lowest record number is the first record in the sector/partition. Thus only one disk head movement is required to access all the index records in the partition. Figure 6.2 above illustrates this relationship.

For very large data files this could result in a large master index. A data file with 4 000 records would have an index file with 4 000 entries. If these latter were blocked 20 per sector then the index file would occupy 200 sectors with 200 entries in the master index file. This could result in relatively long search times for records with high key values. Time could be saved by creating the master index files with enough records for 1 entry per track (number of sectors = 16). Thus the disk head only needs to move once to the relevant track and have access to any index record in that partition/track within one revolution/track of starting the search.

It is difficult to give exact values for the time required to find a particular record in a file structure because of the number of variables involved. Nevertheless, the programmer should keep these points in mind when creating a file structure and balance these to give a reasonable partition size for each circumstance. The variables to be considered are:

a. number of index file records
b. blocking factor of index file
c. load factor of index file
d. amount of memory available to hold the master index.

If 'a' is high and 'b' and 'c' low, then 1 track to a partition could be better than 1 sector to a partition. If 'a' is low then 1 sector to a partition could be better. However, with limited amounts of memory available the master index would have to be small and the partitions arranged to represent a large number of index records.

6.2 Creating the files

The files can only be created on a TOSS formatted disk, i.e. one that has been created by the utility Create Volume (CRV).
The creation of the files must be performed on 8 stages:

1. The data file is set up by using the utility Create File (CRF)
2. The actual records must be written to the file.
3. The index file and master index file must be created using the utility CRF.
4. Before the index files can be built, the utilities used in this process must have two intermediate files. These are created at this stage and for the purpose of this explanation are called IFILE1 and IFILE2.
5. The utility Build Index File (BIX) is next. It takes the data file as input file and builds a file of records that contain the key and the logical record numbers of the data records. This output file is the intermediate file IFILE1.
6. The records on IFILE1 must be sorted according to the pure binary form of the key and into ascending order. Output is to intermediate file IFILE2.
7. The sorted file IFILE2 is used as input to the utility Re-organise Index file (RIX). It takes the sorted records from IFILE2 and writes them in index record format onto the index file created at stage 3. The master index file is generated by RIX at the same time using the file created at stage 3.
8. The data file, index file and master index file are now available for use by the application. IFILE1 and IFILE2 can be deleted.

This process is summarized in figure 4.4.

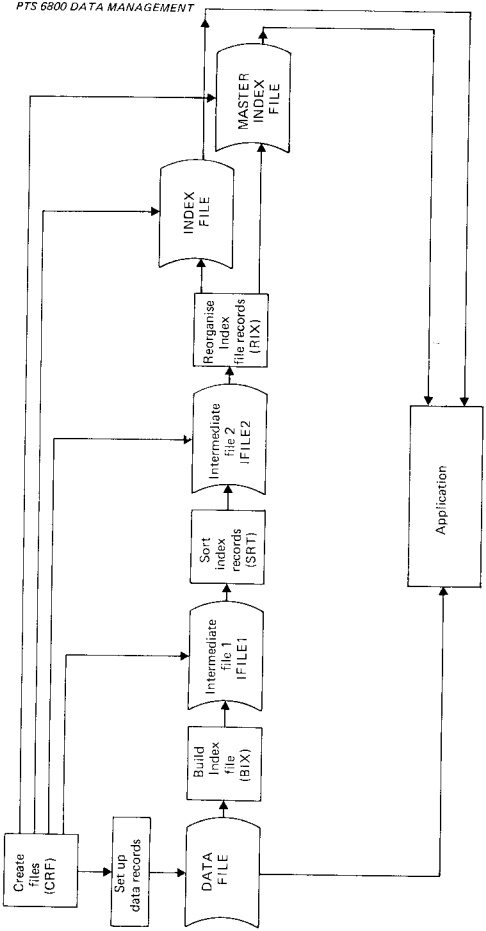


Figure 6.4 Setting up a file structure

Stage 1

The data file must be created using the utility CRF.

1. Call CRF utility under the TOSS utilities Monitor, or as a subroutine from the application, or via the DOS utility TOSSUT.
2. CRF requests a number of parameters. Most parameters can be given as required, but for an indexed random file, two parameters are obligatory. To 'File organization' give 'S', and to 'Number of index files' give 'n' where n is the number of index files required (1–4).
3. CRF searches the volume(s) for free extents large enough to hold the stated file size.
4. The file is created with the required number of records, all of which contain space characters.

Each record is set to 'FREE' status. The LRN is set to zero for this file.

Stage 2

The actual records must be written to the data file using sequential operations otherwise the LRN will not be updated and will still be set to zero the first time that the file is used. The operations for this stage depend upon the source of the records. If the records already exist, for example a bank branch putting its account records onto the PTS system then the records could be copied into the 'empty' file on the disk. If it is a new system it would save time during the next stage if the records could be written into the file in the ascending sequence of the key data item.

Stage 3

The index file and master index files must be created using the utility CRF. The number of index files must be given as '0' in both cases.
The record lengths must be:
for index file = key length + 6
for master index file = key length + 3.

Stage 4

Create the intermediate files using the utility CRF. Any file names can be used, but the example shown in figure 4.4 uses IFILE1 and IFILE2. The number of index files must be given as '0' in both cases.
The record lengths must be:
for intermediate file 1 = key length + 6
for intermediate file 2 = key length + 6.

Stage 5

Next, the utility BIX must be performed as follows:

1. Call BIX utility under the TOSS utilities Monitor.
2. BIX requests a number of parameters via the console typewriter. Most parameters are input as required. The data item that is to be used as the index is specified by two parameters. These are 'Key Address in Record' – give the address of the first character of the key relative the start of the record, in decimal; 'Key Length' – give the length of the required key, in decimal.
3. BIX then scans the data file and copies the required keys to IFILE1. The index records are written sequentially to the file without any regard to the value of the key. For this reason the next stage (sorting) must be performed.

Stage 6

IFILE1 at this point contains the required number of symbolic keys but they are unsorted. The following sequence of operations must be performed.

1. Call utility SORT under the TOSS utilities Monitor, or as a subroutine from the application.
2. SORT requires a number of parameters but this process is a standard sort and requires no special parameters.
3. The sorted records are output to IFILE2.

Stage 7

IFILE2 consists of a set of symbolic keys (with logical record number referring to the data file) that are sorted into ascending order. The index records must now be formatted onto the index file and a master index file built. The master index is structured and formatted by the same utility, Reorganize Index File, and requires no parameters from the user. This master index is stored on the same volume as the index and is assigned at the same time. The sequence of operations for this last stage is as follows:

1. Call RIX utility under the TOSS utilities Monitor, or as a subroutine.
2. RIX requests a number of parameters via the console typewriter. Most parameters are input as required but the reader should note that a value is required for 'Load Factor' which was discussed in section 6.1.2, The Index File. This parameter is the percentage of 'used' records to be written to each sector and should reflect the percentage of 'used' records in the data ifle. The RIX utility will use this factor to construct the partitions and to build the master index file.
3. Index records will be read out from the input file and written in the required format for an index record with the required number of free records at the end of each sector. Records are written to the master index file sequentially during the run. RIX performs a check on the record sequence and an error is set if a key sequence error is detected.
4. At completion, the index is properly structured and formatted and RIX has constructed the relavant master index file.

Stage 8

The file structure is now available for use as an indexed sequential file, see the instructions in the following paragraphs. It is possible to use this data file as a sequential file (for copying or generating reports) or as an ordinary random file (but only if a record key can be input which is related to the record's logical record number). However, no process should be allowed to take place on the data file that is going to disturb the order, or position, of records without also updating the index file.

Note
It is also possible to start with an 'empty' data file, LRN = '0', and hence empty index and master index files. Indexed-insert can be performed until performance considerations require index reorganization with RIX. This would probably be the case for a new application where data is generated and collected as the application goes live (a new branch office, for example, taking on new accounts).

6.3 Instructions

A definitive description of these instructions is contained in the relevant language reference manuals, either. Assembler Programmer's Reference Manual, M06, or CREDIT Programmer's Reference Manual, M04. Before these instructions can be used, both the data file and the index file must be assigned otherwise an error will be returned.
These instructions are, briefly:

ASSIGN THE FILES

An indexed random file must be assigned in two steps.
- the data file is assigned to a file code as accessible by all tasks or only accessible by one task, that is with TC = 0 for common files, TC = 1 for this task only.
- the index file is assigned using the index file assign instruction. The associated master index file is assigned implicitly and read into memory. If more than one index/master index is to be used, they must all be assigned separately.

On this data file records may be retrieved, deleted, stored, inserted, or retrieved sequential from a certain point by using the commands indexed random read, indexed delete, indexed rewrite, indexed insert, or indexed read next. The record to be fetched is indicated by means of a symbolic key.
It is allowed to assign an index-file as a data file, in this case the user is able to process this file as an ordinary data file and can create his own index records. It is not allowed to assign an index file as both a data file and index file at the same time.
An indexed rewrite, indexed delete or indexed insert may be performed when all index-files corresponding to the data file are assigned.

Assigning a file code to a data file or an index file on flexible disk results in the door of the corresponding flexible disk drive being locked.

INDEXED RANDOM READ

The task must supply a symbolic key with this instruction. Data Management searches the master index first until a matching or first highest key is found. This will point to the relevant partition in the index file and cause a search in that partition until a match occurs. This will give a logical record number and the record will then be accessed directly. The record can then be put under exclusive access if required. The CRNs for both the index file and data file will be updated.

INDEXED REWRITE

The data record must first be read and exclusive access set (if it has been specified during system generation before this instruction is used). The record is replaced by the new record, except the symbolic key which remains unchanged, and exclusive acces released. The CRN for the data file will be updated to this record number and set to zero for the index file. Note that in CREDIT this instruction is effected through the DSC1 statement.

INDEXED DELETE

This instruction will delete the data record and the entries in the index file. The task must supply the logical record number of the data record which must be under exclusive access (if this has been specified during system generation). The data record is read and the index entry is deleted only after a successful read. The data record is set to 'free' and exclusive access is released. The CRN is not affected.

INDEXED INSERT

This instruction allows the task to insert an new data record into the data file and create a new index record. The data record is added to the end of the file and the LRN is updated in memory. The symbolic key is inserted in the correct place in the index file according to its pure binary value and the records following the new index record in same disk sector shifted along into the free area. If an index record already exists with the same symbolic key, the new is inserted in front of the old. Exclusive access is not set and the CRN in the data file and index file will be set to the values of the new record and its index.

INDEXED READ NEXT

This performs the same function as indexed random read except that no symbolic key is supplied. The data record that is read is the one that is referred to by the *index record* following the *current index record number*. This implies that the 'read next' is the next highest symbolic key and the data record it points to. If the CRN is zero then the first key in the index file is used. The CRNs in the data file and index file will be set to the new values.

This instruction can only be used after :-

a. indexed random read
b. indexed insert
c. or another indexed read next.

CLOSING THE FILES

A close file is issued for first the data file, then each of the index files to indicate that the file is no longer required by the task. By closing a file the LRN in the volume table of the corresponding file is updated and saved on the volume. The previously assigned file code is no longer valid for this file.

When an index file is closed the master index file will be closed implicitly.

Note

If a large number of changes has occured during the time the file structure was in use, it is advisable to use utility 'A' to reorganize the index and master index files. This will ensure that the master index file is a true picture of the structure of the index file.

A data file with 'secondary record' is reorganized with 'Copy File with pack' utility program (Cf F) after which the indexed must be rebuilt.

CONTENTS