A general purpose inverting indexing system

B.H. Pereira

UNIVERSITY OF DURHAM - FACULTY OF SCIENCE

M.Sc. Thesis - Summary

A GENERAL PURPOSE INVERTED INDEXING SYSTEM

The object of the thesis is to study the use of Inverted File organization in a research environment. The main reason for the selection of this subject is the proliferation of information processing systems, leading to duplication of programming and maintenance effort.
Some other considerations to be taken into account include provision of Data Independence by use of Data Description Directories, together with User-Friendliness using tabular processing concepts based on the Relational Model of Data.
A computer system has therefore been developed which embodies the above concepts in a practical system which is general purpose - i.e. it could be used to process any type of data provided it is presented in a tabular form.
This system - called the General Purpose Inverted Indexing System - has been written in PL/1 and operates on the Northumbrian Universities Multiple Access Computer (NUMAC) IBM System/360 Model 67 under control of the Michigan Terminal System (MTS). It comprises 24 modules totalling about 2000 PL/1 statements and includes the major functions required in a general purpose system, including File Description, Set-Up of Inverted Indexes and Processing of Queries.
The thesis first describes inverted file organization concepts, and then goes on to show how they are applied in a general purpose system. The components of the General Purpose Inverted Indexing System are then described, together with their interaction and operation. The logical and performance capabilities are defined and analysed; followed by recommendations as to how the system may be improved by future enhancements.

B.H.Pereira, April 1975.

UNIVERSITY OF DURHAM

FACULTY OF SCIENCE

M.Sc. THESIS

A GENERAL PURPOSE INVERTED INDEXING SYSTEM

B.H.PEREIRA - APRIL 1975

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION TO THE THESIS

1.1. Background

The aims of this thesis are defined to be a study of the use of Inverted File organization in a research environment. The reasons for the selection of this subject are as follows:

1. Proliferation of Information Retrieval Systems.
   Varying disciplines within the university research environment use computer facilities to store and access data. This leads to a situation where a number of researchers could be writing virtually identical programs to accomplish the same data processing functions. The consequences of this approach are a considerable duplication of programming and maintenance effort.
2. A Common Information Retrieval System.
   An alternative approach might be to design a single general purpose information retrieval system which can handle differing user requirements. To do this, it would be necessary to examine research requirements for data processing; and from this to design a system which embodies the major characteristics of research data processing.
3. Selection of Inverted File Organization.
   A characteristic of information retrieval systems is that the emphasis is placed on fast retrieval of information rather than on speed of update of the Data Files. The Inverted List method of file organization is a prime candidate for consideration; as it allows fast index searching, but tends to be inefficient in updating situations.

The Inverted File organization method is therefore defined as the foundation stone on which a system might be constructed. However, from studies of current thinking on File Organization and Data Base concepts, two other building blocks might be defined:

1. Data Independence.
   The trend in File Organization is towards files that can be maintained independently of programs. The advantages of this approach are that files can be maintained independently of the programs, without necessitating re-compilation of the programs. The provision of Data Independence has been studied in a number of Data Management systems and included in the CODASYL Data Base Task Group Report (CODASYL(1971) - 2 references) as a possible industry standard.
   The requirement for Data Independence is provided by an interface between the user and the physical Data File by

1.1. Background contd

means of a Data Description Table (or similar names). All user access to the Data File is through the Data Description Table using the required File and Field names.

2. The Relational Model of Data.

A prime requirement of future information systems is that they employ some entirely general method of storage and processing of data. The relational calculus, based on the Relational Model of Data, is a method very suited to this requirement, as described by CODD(1970) and NOTLEY(1972). The major consequences of use of this approach is that it provides:

(a) An entirely general method of manipulating data.
(b) A maximum degree of machine and data independence.
(c) The simplest possible data structure (i.e. a relation or table) consistent with the semantics of the stored information.

Therefore, in order that the system may make use of the Relational Model of Data, it would be necessary to store the data in a Relational or tabular form. It would also be necessary to provide outputs from the system in a form best suited for processing by Relation operations e.g. Select, Join etc. The method for this would be to provide output lists in sequence for further processing.

A system is therefore to be designed and implemented which embodies the above concepts in a practical system which is general purpose - i.e. it could be used to process any type of data provided it is presented in a tabular form. A general method of defining queries and displaying the results should also be provided.

1.2. Development and Implementation

In line with the objectives defined above, a General Purpose Inverted Indexing System has been designed and implemented. It is written in PL/1 and runs on the Northumbrian Universities Multiple Access Computer (NUMAC) IBM System/360 Model 67 under control of the Michigan Terminal System (MTS). It contains 24 modules - 10 Main Programs and 14 Subroutines or Macros totalling some 2000 PL/1 statements. There are three main levels of processing with associated functions:

1. Data Management.

Facilities are provided to set up an Index File where inverted indexes abstracted from various Data Files may be stored.

1.2. Development and Implementation contd

The direct access space within the Index File is controlled within the system. All I/O coding is contained within a number of routines which may be accessed from higher levels of the system. Processing functions provided are:

(a) Set Up a Blank Index File. Initialize the direct access space within the Index File before any indexes are loaded.

(b) Extend an Existing Index File. Increase the amount of direct access space available within the Index File.

(c) Status Report on Index File. Display current file status such as amount of space in use or available.

2. File Description Directories.

Within the Index File, Directories may be set up to describe the physical storage characteristics of Data Files. Each entry describes the File Name, together with associated Field Names and Characteristics for one Data File. Functions provided are:

(a) Add a File Description to the Directories.

(b) Delete a File Description from the Directories together with any associated inverted indexes.

(c) Directory Display Report. Display contents of Directories.

3. Inverted Index Processing.

The number of fields in a Data File which are to be indexed are dependent on the user's decision. He is provided with the following processing functions:

(a) Create an Inverted Index for a specified Field.

(b) Delete an Inverted Index for a specified Field.

(c) Define and Process Queries using the Inverted Indexes.

(d) Display the Contents of a specified Inverted Index. This report is primarily intended for system debugging.

1.3. Testing, Results and Discussion

Initial system testing was carried out using a small file of data on characteristics of Minicomputers.

This was then followed by examination of two aspects of system performance essential to the viability of an inverted file organization method:

1. Creation of Inverted Indexes.
2. Processing Realistic Queries.

To provide data for testing, two sets of research data were used:

1. An abstract from the Durham High Energy Physics databank

1.3. Testing, Results and Discussion contd

to provide various Data Files for processing of between 2000 and 10000 records (approximately).

2. Geographic Data from the Northern Archaeological Survey in a file containing approximately 3300 records.

A number of runs were carried out and analysed, from which the following sample results and discussion are abstracted:

1. The CPU time taken on the IBM System/360 Model 67 to create sufficient inverted indexes to meet the query requirements for the two Data Files is shown in the following specimen times:
   (a) High Energy Physics - Create inverted indexes on 7 fields from 9635 records at an average of 0.0132 seconds CPU time per index entry:
   Total CPU time required = 890 seconds ≐ 15 minutes.
   (b) Geographic Data - Create inverted indexes on 3 fields from 3280 records at an average of 0.0109 seconds CPU time per index entry:
   Total CPU time required = 107 seconds ≐ 1.75 minutes.
2. A number of queries of varying complexity were defined and solved. The solution time is dependent on a number of factors, including the complexity of the query, the number of comparisons and the number of records found. The maximum time taken to answer the largest, most complex query was 40 seconds of CPU time.

Following analysis and discussion of the results, a number of recommendations are presented as to how the performance and function of the system may be improved. The means of performance improvement would be rationalization of PL/1 coding and provision of larger block sizes for Disk I/O. Functional enhancements recommended include Interactive Processing, Language Processing and Display Facilities, and Update Processing.

1.4. Content of the Thesis

The thesis is composed of seven Chapters and three Appendices. A brief description of each of these components follows:

1. Chapter 1: Introduction to the Thesis.
   The background of the thesis is introduced, followed by comments on the development, implementation and testing of a General Purpose Inverted Indexing System. This section shows where various items of information may be found; and is followed by acknowledgements.

1.4. Content of the Thesis contd

2. Chapter 2: Inverted File Organization.
   The basic principles of Inverted File organization are described with particular reference to its applicability in a research environment. Two commercial implementations - TDMS and System 2000 are described and analysed.
3. Chapter 3: Design Philosophy.
   The main requirements for research data processing are identified, and a design philosophy developed making use of Inverted File organization, tabular Data Files and File Description Directories.
4. Chapter 4: System Description.
   The description of the system logic is presented here. Topics covered are the conditions for Data Storage and Types, a description of the System Modules, their interrelation, and dependencies of the system on the Michigan Terminal System (MTS) under which it runs.
5. Chapter 5: System Operation.
   The use of the system is explained with reference to a small Data File of Minicomputer data. The system operation under MTS is explained, together with associated Job Control, Input requirements, and Display of results.
6. Chapter 6: Performance Testing.
   Two research data files - High Energy Physics and Geographic Data on Archaeological Sites - are used to provide Performance Testing data for Index Creation and Query Processing.
7. Chapter 7: Analysis, Discussion and Recommendations.
   The Performance Testing results are analysed and discussed. Recommendations are made as to how both the performance and function of the system might be improved by future enhancements.
8. Appendix A: Glossary of Terms.
   Words and phrases defined for use in description of the General Purpose Inverted Indexing System are listed and their meaning explained.
9. Appendix B: Bibliography.
   The literature references mentioned in the text of the thesis, e.g. CODD (1971), are listed.
10. Appendix C: Source Listings.
    PL/1 Source Code listings for all the modules in the system are displayed.

1.5. Acknowledgements

The author would like to acknowledge in particular the assistance of his Supervisor - Mr.J.S.Roper, Senior Lecturer in Computing. He has by his constructive comments and advice contributed greatly to the specification and implementation of the concepts that have been researched in this thesis.

In addition, thanks are offered to Dr.C.Cooper of the Physics Department and Mr.P.Clack of the Archaeological Department for allowing the use of their research data with which to test the system, as well as for discussions on their research data processing requirements.

2.1. Selection of File Organization Method

File searching may be regarded as a two-step process:

1. An input key or field value is decoded or translated to a list of record addresses in a Data File of all records containing that key or field value. The most common method is by the use of an Index or Key Directory.
2. A random access search of the Data File is then made to retrieve the required records.

The number of access paths that are provided to the Data File will depend on the retrieval requirements. Two main types of retrieval requirement may be identified:

1. Commercial Systems.
   Commercial data processing systems emphasise the currency and accuracy of their Data Files. The major requirement is that these files may be quickly updated. The enquiries tend to be on one key only and fairly standard in structure.
2. Library Systems.
   Information systems have evolved from filing systems that could no longer contain the increasing volume of reference material. Until recently, Information Retrieval has referred to a context in which bibliographic information (which is relatively static) is assembled for unpredictable reference. Computer-based information retrieval systems are particularly used in areas of high technical content, e.g. medicine and chemistry.
   The major requirement here is for fast retrieval in response to unpredictable queries. Multiple keys may have to be accessed to quickly answer the queries.

The requirement for information retrieval in research work might fairly be classified under the second heading of Library Systems. There are varying types of search file organization which offer multiple access paths to the data such as Inverted List, Multilist etc. DODD(1969), LEFKOVITZ(1969), and ROBERTS(1972) all describe and compare these varying techniques. Each offers certain advantages and disadvantages dependent on the processing requirements. CARDENAS(1973) goes one step further in developing a methodology, a model and a programmed system to select an appropriate file structure for a specific situation. Results of theoretical and simulation comparisons made in the references quoted above indicate that the Inverted File Organization method is most suited to a file processing environment in which speed of retrieval is essential and frequent update minimal; which describes to a major

2.1. Selection of File Organization Method contd

extent the processing of research data.

This chapter examines the basic principles of Inverted File Organization, takes as examples its application in two practical implementations - TDMS and System 2000, and summarizes the basic characteristics.

2.2. Basic Principles

In an Inverted File Organization method, all record addresses are contained within the Index. This normally results in a much larger Index than for other search file organizations; although the total storage requirement is often no greater because pointer linkages are not stored in the Data File records. Figure 2.1. shows a specimen Inverted File for a Skills Index. The format shown here could be physically stored with the Index and Data as separate files.

Storage efficiency could in fact be further improved if the keys or field values were not individually cited within the record itself by removing the keys into the Index. This may however complicate updating of the Data File.

The lists in the Index are variable length records that must be maintained in a given sequence for efficient manipulation. Both the maintenance of the lists as variable length records (which can be quite diverse in size), and their maintenance in sequence, can contribute to certain programming complexities that are absent in other search file organizations (such as Multilist), although they buy much in performance.

2.3. Retrieval using an Inverted File

The Inverted File Organization was originally developed to minimize the time needed to retrieve data from the required Data File at the expense of update time. The ideal file structure for information retrieval would read only the desired data records from direct access storage and no others; performing all preliminary searching in main memory. If the complete Inverted Index could be held in main memory, it would achieve this goal. However, even if the inverted lists must be held on direct access or secondary storage, one access can read a list pointing to many more records than could be read in one access; so that even in this situation, the number of accesses is less than would be required to search the records themselves. It is this ability to perform the search on the required conditions before the Data File is accessed that makes the Inverted File so fast on retrieval compared to other search file organization methods.

FIGURE 2.1: INVERTED FILE - SKILLS INDEX

INDEX INVERTED ON SKILL,LANGUAGE,LOCATION

FIELD VALUE	COUNT	LIST
PROGRAMMER	3	2,4,5
SYSTEMS ANALYST	4	1,3,6,7
FRENCH	3	3,5,6
GERMAN	1	1
RUSSIAN	2	2,7
AMSTERDAM	2	1,2
LONDON	3	3,4,7
PARIS	2	5,6

DATA FILE SEQUENCED BY NAME

NO.	NAME	SKILL	LANGUAGE	LOCATION
1	ANDERSON,P.	SYSTEMS ANALYST	GERMAN	AMSTERDAM
2	ATLEY,T.	PROGRAMMER	RUSSIAN	AMSTERDAM
3	BAKER,A.	SYSTEMS ANALYST	FRENCH	LONDON
4	CONWAY,J.	PROGRAMMER	——	LONDON
5	CURRY,W.	PROGRAMMER	FRENCH	PARIS
6	CURTIS,B.	SYSTEMS ANALYST	FRENCH	PARIS
7	WILSON,N.	SYSTEMS ANALYST	RUSSIAN	LONDON

2.3. Retrieval using an Inverted File contd

The contents of each Inverted Index entry are as follows:

1. Key or Field Value being indexed.
2. Count giving the number of records in the Data File containing this Key or Field Value.
3. List of Record Addresses, the number given in 2. above.

The Count value may be used to facilitate AND and OR operations by selecting the shortest lists for scanning, if the search permits such a choice. The potential advantage of this approach can be very great if the two lists are very different in length. For example, if one wished to find a Graduate who could speak both Spanish and Portugese in a Skills Index, it might be easier to first intersect the Spanish and Portugese lists, and then intersect the resultant list with the Graduate list; rather than start with the Graduate list.

Union and Intersection operations on Inverted Lists are facilitated if each list is maintained in collating order of record addresses. In this case, two lists can be either intersected or their union found in one pass through both lists. For this reason, inverted files are almost always kept in collating sequence.

2.4. Updating an Inverted File

An Inverted File is difficult to update. To add a record, the inverted list corresponding to the value for each field in the record that is indexed must have a pointer to the new record added. The necessity of keeping each inverted list in order (to speed up searches) tends to increase the complexity of this operation. One method of simplifying this process is to allow extra space at the end of each inverted list to allow for a small number of additions.

Conversely, the deletion of a record from a Data File accessed by an Inverted Index similarly requires the modification of inverted lists corresponding to the value of every field in the record indexed, with the same problems.

The complexity of the update problem is reflected in various approaches to attempt to improve the update efficiency of Inverted Files such as work by VOSE & RICHARDSON(1972) and INGLIS(1974).

2.5. Practical Implementations

It is of interest at this point to investigate two practical implementations of Inverted File Organization with reference to their capabilities and limitations.

2.5. Practical Implementations contd

The two systems are:

1. Time Shared Data Management System (TDMS).

   This system, developed by SDC in the U.S. is described by BLEIER & VORHAUS (1968). It is designed to operate under the control of a time-sharing executive on IBM System/360 Models 50 and larger.

   In this system, the inverted structure has been developed to support the user who needs to find answers to unpredictable questions. The inverted structure is totally organized and sorted by component value and occurrences of each value (examples of "components" are Name, Height, Weight, Sex etc). Any value of any component in the file may be used as a key for retrieval. The organization of the inverted structure groups all potential keys into blocked tables in a manner that minimizes storage requirements and access requests. The user defines his Data File within a dictionary which contains the names and synonyms of all the components in the Data File, the hierarchy to which they belong, the maximum number of characters for each component in the load process, and all the legality statements associated with each component.

   Bleier & Vorhaus find that with this type of file structure, the size of the data base has very little effect on the speed of retrieval. The important limiting factors in retrieval time are the number of Boolean expressions, the number of unique values per component in each Boolean expression, and the number of occurrences of the desired value in each Boolean expression.

   To obtain fast retrieval, some penalties must be paid. First, the inverted structure requires as much space (if not more) than the data itself. Secondly, the structure is difficult to maintain. Two types of updating are allowed: On-Line and Batch. To accomodate On-Line updating, a small number of spare words of storage are allocated at load time in every block of tables. Where Batch updating is required, the file is unloaded and reloaded.

2. System 2000.

   This system was developed by MRI Systems Corporation in the U.S. and is marketed in the U.K. by CAP. It is described in a General Information Manual - MRI (1972). System 2000 operates

2.5. Practical Implementations contd

on IBM System/360 and 370, CDC 6000 and CYBER 70 and UNIVAC 1100 series computer systems.

Basic System 2000 provides the user with a comprehensive set of data base management capabilities. These include the ability to define new data bases, modify the definition of existing data bases, and to retrieve and update values in these data bases.

System 2000 data bases consist of three separate but integrated parts: the definition, the logical entries, and two sets of pointers. One set describes the interrelationships of the data values and the other, the Inverted Index, indicates locations of data values. At the time the data base is defined, the user determines which values are to be inverted. The major part of data base search takes place within these sets of pointers before any data values are accessed.

Various features supplied by System 2000 include:

(a) Logical page size for the data base may be specified, thus providing for efficient access to various sizes of data base.

(b) Logical entries may be retrieved using a WHERE clause which allows operators such as Equal, Greater Than etc. as well as Boolean processing.

(c) Varying data types are supported together with varying field lengths.

(d) The system may be processed either in Batch Mode or On-Line.

It should be noted that both TDMS and System 2000 provide:

1. A directory to describe Data Files.
2. All logical file processing requirements.
3. The ability to formulate queries with Boolean processing.
4. The systems may be run either in Batch or On-Line modes.

No mention is made in the System 2000 Manual of how On-Line Update is accomodated.

2.6. Inverted File Characteristics

LEFKOVITZ (1969) presents a summary of the properties of various search file organization methods. Those relating to the characteristics of Inverted File Organization might be presented as follows:

1. Total retrieval time in answer to a query is very fast in comparison to the other methods. This is due to the

2.6. Inverted File Characteristics contd

search being carried out wholly in the index where lists of record addresses are processed.

2. Pre-search retrieval statistics may be easily obtained by determining the counts for the field values being accessed.
3. Programming an inverted list organization is more difficult than other search file organizations due to problems in creating and maintaining variable-length index lists.
4. Update time is long compared to other methods.
5. The direct access storage requirement can vary depending on whether the keys are abstracted from the Data File into the Index or not (if not a double update problem emerges).
5. "Complex" queries involving Boolean processing may be quickly answered, depending on the number of fields within the data record indexed (degree of inversion).

Thus, Inverted File Organization might be effectively used in information retrieval requirements where the data records are relatively stable (small amounts of update) but fast retrieval is required.

CHAPTER 3: DESIGN PHILOSOPHY

3.1. Introduction

In a university research environment there is a continuing requirement for data from many varying disciplines to be stored and accessed by computer processing methods. This proliferation of many different types of data files, each with their own processing programs, leads to considerable duplication of programming effort and increased maintenance requirements.

It was therefore decided to investigate an alternative approach: to design a single file processing system that could handle differing user requirements. In this M.Sc. project, the Inverted File organization method has been studied in order to design and implement a General Purpose Inverted Indexing System.

3.1.1. Characteristics of Research Data Processing

In order to be able to implement any general purpose system, it is first necessary to be able to define the characteristics of current systems. Common factors should be identified as well as any variations. The design process sets the most acceptable common factors as standard but also defines what must be done to bring variations into line with the standard.

The major characteristics of how research data is stored and accessed may be defined as follows:

1. Research data is stored in files.
2. The files are made up of logical records.
3. Some form of blocking of logical records may take place.
4. File access may be serial or direct.
5. There may be varying record types - e.g. some form of hierarchical organization.
6. The logical records are made up of fields.
7. The fields may be stored as many different data types.
8. The fields may be variable in length.
9. There may not be the same number of fields per record - e.g. repeating groups.
10. The general processing mode for the files after they have been created is the answering of queries. Any update seems to take the form of the addition of a large number of new records to the file - e.g. addition of a new batch of experimental results.

3.1.2. The Tabular Method of Data Storage

To be able to define a common approach, it is necessary to consider what form of data storage might be acceptable. Probably the most generalized acceptable form of data storage from the point of view of the user is a table. Implications of a tabular view of storage and consequent processing have been thoroughly researched by CODD (1970) in his work on the Relational Model of Data. In a further paper on Normalized Data Base Structures, CODD (1971) has this to say about the simplicity and applicability of a tabular approach:

> "The complexity of the physical representations which these (file management) systems support is perhaps understandable, because these representations are selected in order to obtain high performance in access and update. What is less understandable is the trend towards more and more complexity in the data structures with which application programmers and terminal users directly interact. Surely, in the choice of logical data structures that a system is to support, there is one consideration of absolutely paramount importance - and that is the convenience of the majority of users."
>
> "To make formatted data bases readily accessible to users (especially casual users) who have little or no training in programming we must provide the simplest possible data structures and almost natural language. ..... The casual user at a terminal often has occasion to require tables to be displayed or printed. What could be a simpler. more universally needed, and more universally understood data structure than a table? Why not permit such users to view all the data in a data base in a tabular way?"

3.1.3. Implications of the Tabular Method for Research Files

By presenting the data in a tabular form, a number of common factors may be identified for the storage of research data:

1. Research data is stored in files.
2. The files are made up of logical records.
3. There is only one record type per file.
4. Each logical record is made up of fields.
5. There are the same number of fields in each record.
6. All fields are of fixed length.
7. The acceptable data types are defined and standardized.

Note that no mention is made in the above list of blocking or file

3.1.3. Implications of the Tabular Method for Research Files contd

access, as data stored in a tabular form may be blocked or unblocked, serial or direct, as required.

3.1.4. Description of the Files

Having therefore defined a method of data storage as a tabular file, that file may now easily be described by an entry in a File Description Directory. A directory of this type would contain a number of records, each of which describe the record structure of one data file. Setting up of a File Description Directory would provide a measure of data independence, as all access to the data files would be made through the Directory.

3.1.5. Use of the Inverted File Organization Method

The next consideration in the design philosophy must be the provision of indexing capabilities. In a commercial data processing environment, this is fairly straightforward as each record is normally indexed only on one pre-defined key field. However, in a scientific and/or research environment, the problem is more complex. To be able to answer queries, it may be necessary to index on more than one field. The philosophy of inverted indexing is quite amenable to this, providing as it does the concept of indexing on one field through a number of fields (partial inversion) to all the fields in the record (total inversion). In addition, as discussed by LEFKOVITZ (1969), the inverted file organization method is most suited to a file processing environment in which the speed of query processing is essential and update minimal; which describes to a major extent the processing of research data.

3.1.6 Practical Implementation of Inverted Indexing Concepts

From the theoretical consideration of inverted indexing concepts, what is now needed is the practical ability to implement these concepts. A facility should therefore be provided to enable the user to create indexes on specified fields at will. The index creation would take place through and be linked to the File Description Directory. As the number of indexes is the prerogative of the user, there would be no way of knowing prior to creation of the indexes how many indexes there would be. In addition, there is the probability that during the course of the user's research, that either new indexes would need to be created or existing indexes destroyed.

3.1.6. Practical Implementation of Inverted Indexing Concepts contd

It would therefore seem reasonable to contain the indexes within some direct access space which is controlled either directly or indirectly by the user; because to leave this to the operating system would entail a considerable overhead in creation/deletion of files with associated job control requirements.

3.1.7. Updating of Inverted Indexes

As discussed in Section 3.1.1., any update in a research environment generally seems to take the form of the addition of a large number of new records to the file. LEKFOVITZ (1969) states that inverted file organizations are extremely inefficient in update. As the provision of update facilities would necessitate extremely complex space management facilities with corresponding decreases in query access speed; the update facility required as mentioned above will take the form of deletion and re-creation of the appropriate inverted indexes.

3.1.8. Processing of Queries

An Index File is being considered which is controlled by a File Description Directory, created from Data Files, and providing access to Data Files in answer to processing queries. It would be desirable to be able to answer all queries within a given basic framework of logical selection conditions.

3.1.9. Definition of System Components

Three main areas may be defined for the design philosophy of a General Purpose Inverted Indexing System. These are:

1. Data Management.
   The control of direct access space within an Index File.
2. File Description Directories.
   The ability to describe the structure of records within Data Files.
3. Inverted Index Processing.
   To be able to specify creation of inverted indexes, perform logical processing, and deletion of inverted indexes as required by query requirements.

Each of these areas is discussed in more detail in the following sections.

3.2. Data Management

If separate indexes are to be set up and maintained for each field indexed, it will be necessary to contain these indexes within some direct access space which is controlled by the user; because to leave this to the operating system would entail a large overhead in creation of files with associated job control requirements. It is therefore recommended that a data management method for creating and maintaining control of the direct access space required for the various inverted indexes be implemented.

3.2.1. Control of Direct Access Space

To explain the processing involved, it is first necessary to define the terms used. The Index File contains a number of physical records called Blocks, each of which is the same pre-determined length and is directly addressable. The first block in the Index File contains control information. On creation of the initial Index File, a chain of direct access pointers is set up anchored to the first block (Control) through all the blocks in the system to provide an availablility chain (See Figure 3.1). All pointers on the system are forward. To store inverted indexes and other information on the Index File, it may be necessary to allocate a number of blocks to a particular usage. One or more blocks chained together for this purpose are called a Sub-File. When sub-files are allocated, the overall Index File availability chain is modified to branch round the blocks allocated (See Figure 3.2). Conversely, when a sub-file is deleted, the blocks contained within are returned to the availability chain.
Thus on-line space allocation and recovery within the Index File is available to the user through access from higher levels of the inverted indexing system.

3.2.2. Storage of Data within Sub-Files

Processing within sub-files is forward sequential. Read and Write routines within the Data Management routines but accessed by higher levels of the system automatically keep track of positions within chains of blocks which make up a sub-file.
For example, a higher level routine would request the writing of a number of bytes to a sub-file from a given position. The appropriate data management routines would simply write this data to the sub-file making all allowances for overflow of physical blocks if required.

FIGURE 3.1: INDEX FILE CREATION

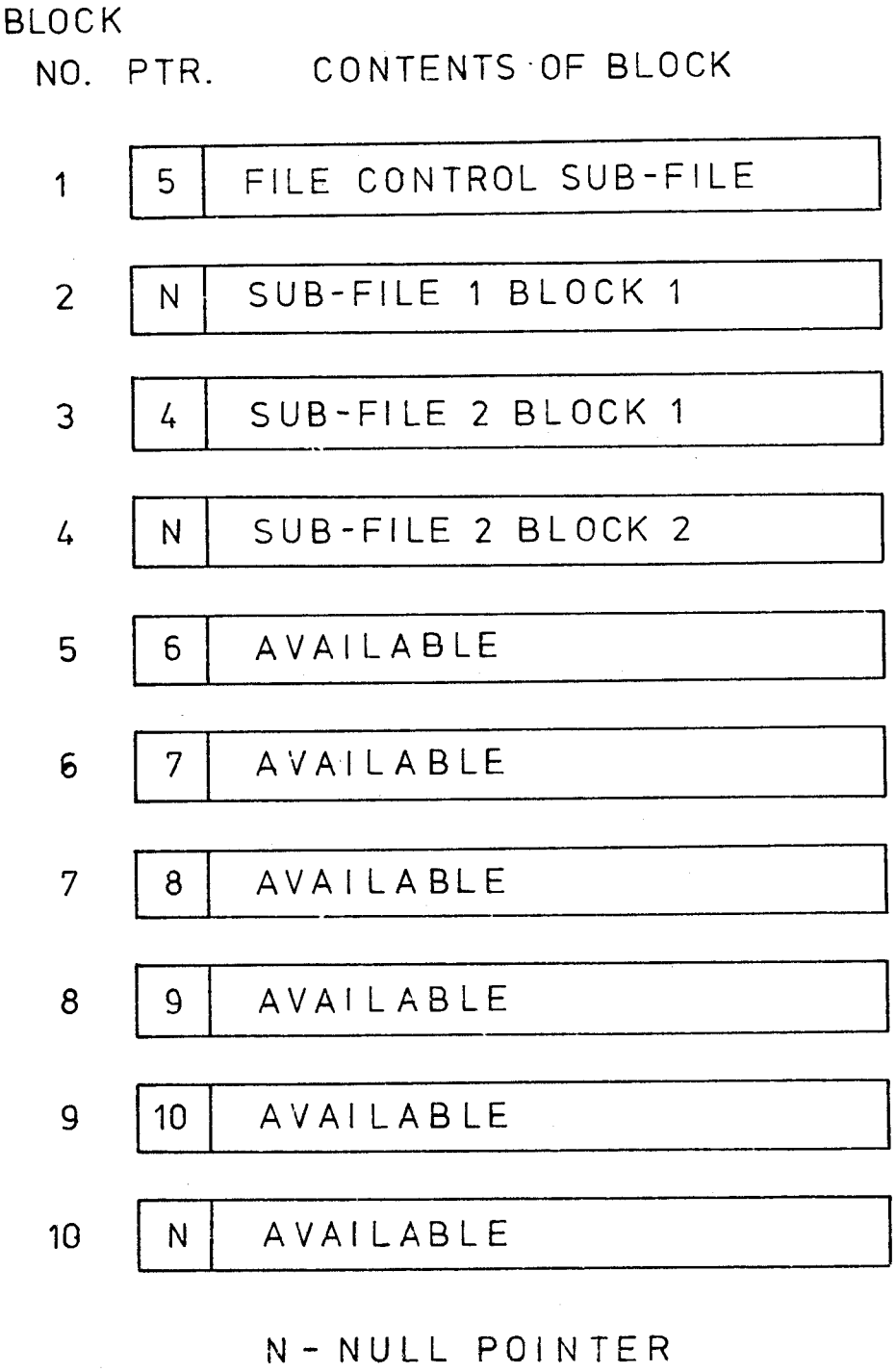
10 BLOCKS - 9 AVAILABLE

BLOCK NO.	PTR.	CONTENTS OF BLOCK
1	2	FILE CONTROL SUB-FILE
2	3	AVAILABLE
3	4	AVAILABLE
4	5	AVAILABLE
5	6	AVAILABLE
6	7	AVAILABLE
7	8	AVAILABLE
8	9	AVAILABLE
9	10	AVAILABLE
10	N	AVAILABLE

N - NULL POINTER

FIGURE 3.2: ALLOCATION OF SUB-FILES

10 BLOCKS - 6 AVAILABLE



N - NULL POINTER

3.2.2. Storage of Data within Sub-Files contd

Figure 3.3 shows an example of a sub-file comprising 2 blocks - 3 & 4. The sub-file position pointer is pointing to the start of a string of numbers in block 3. A Read command is given to get 10 bytes from the sub-file. The data management routines read 4 bytes from block 3, detect an overflow, chain to block 4, and then read the remaining 6 bytes into the data area requested. Following completion of the Read, the sub-file position pointer is set to point to the next byte following the string.

3.2.3. Data Management Utility Programs

A number of utility programs are required to set up, extend, and display the status of the Index File. A brief description of each of these programs follows:

1. Index Creation Utility.
   This program is run before any File Descriptions or Inverted Indexes are loaded. It sets up the Index File Control Sub-File (1 block) containing control information and sets up the initial availability chain through the direct access blocks.
2. Index Extension Utility.
   If the available space in the Index File is becoming filled, this program may be run to extend it with modification of the availability chain. The File Control information is updated.
3. Index Status Utility.
   This program is used by the user/systems programmer to determine the current status of the Index File; paying particular attention to the number of blocks presently available.

3.2.4. Data Management Processing Routines

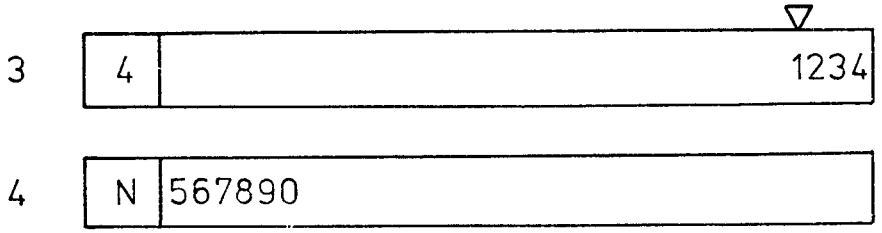
A number of routines are used by the data management system to process the data contained on the Index File. These are generally called from higher levels of the system. The main routines are:

1. OPEN.
   Generally invoked at the start of a run, it brings the Index File Control Sub-File into main memory, where it remains for the duration of the run.

FIGURE 3.3: LOGICAL PROCESSING OF SUB-FILES

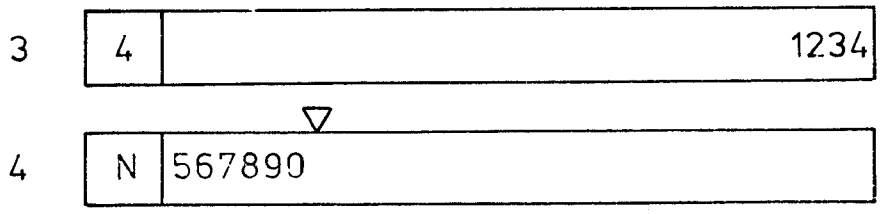
▽ - SUB-FILE POSITION POINTER

BEFORE READ COMMAND



READ COMMAND: GET 10 BYTES FROM SUB-FILE

AFTER READ COMMAND



RESULT: CHARACTERS '1234567890' READ

3.2.4. Data Management Processing Routines contd

2. CLOSE.
   This routine is called either at the end of a run or when an error occurs. It takes the Index File control information in main memory including any modifications made during the run and writes it back to the File Control Sub-File.
3. CREATE.
   The first available block on the Index File is made available to the user as the start of a sub-file, and the availability chain is modified to by-pass it.
4. DESTROY.
   The address of the first block in a sub-file is supplied to the system and the blocks thus released are returned to the Index File availability chain.
5. READ.
   The address of some data within a sub-file is supplied to the system together with the number of bytes to be read. The data is transferred from disk into an area in main memory specified by the calling routine. On completion of the read process, the sub-file position pointer is modified to point to the next address in the sub-file.
6. WRITE.
   The address of some data within a sub-file is supplied to the system together with the number of bytes to be written. The data is transferred from an area in main memory specified by the calling routine to disk. If a request for a write means that the current limits of a sub-file will be extended (as in creation of an inverted index), a further block is transferred from the Index File availability chain to be added onto the end of the sub-file in process. On completion of the write process, the sub-file position pointer is modified to point to the next address in the sub-file.

3.3. File Description Directories

Once the basic Data Management routines for the Index File have been provided, it now becomes necessary to be able to describe to the system the necessary data file and field specifications. To be able to describe a file it is necessary to have available the following information:

1. File Name and No. of Fields in each record.

3.3. File Description Directories contd

2. For each Field: Field Name, Type and Length.

To this basic information, the system may add such information such as file and field numbers, run statistics, pointers to indexes etc.

3.3.1. Storage of Directories on the Index File

Within the General Purpose Inverted Indexing System, the storage of file and field information may be carried out by building a set of sub-files. More than one file description together with associated indexes may be stored on the system at one time. The sub-files used for the File Description Directories (see Figure 3.4) are as follows:

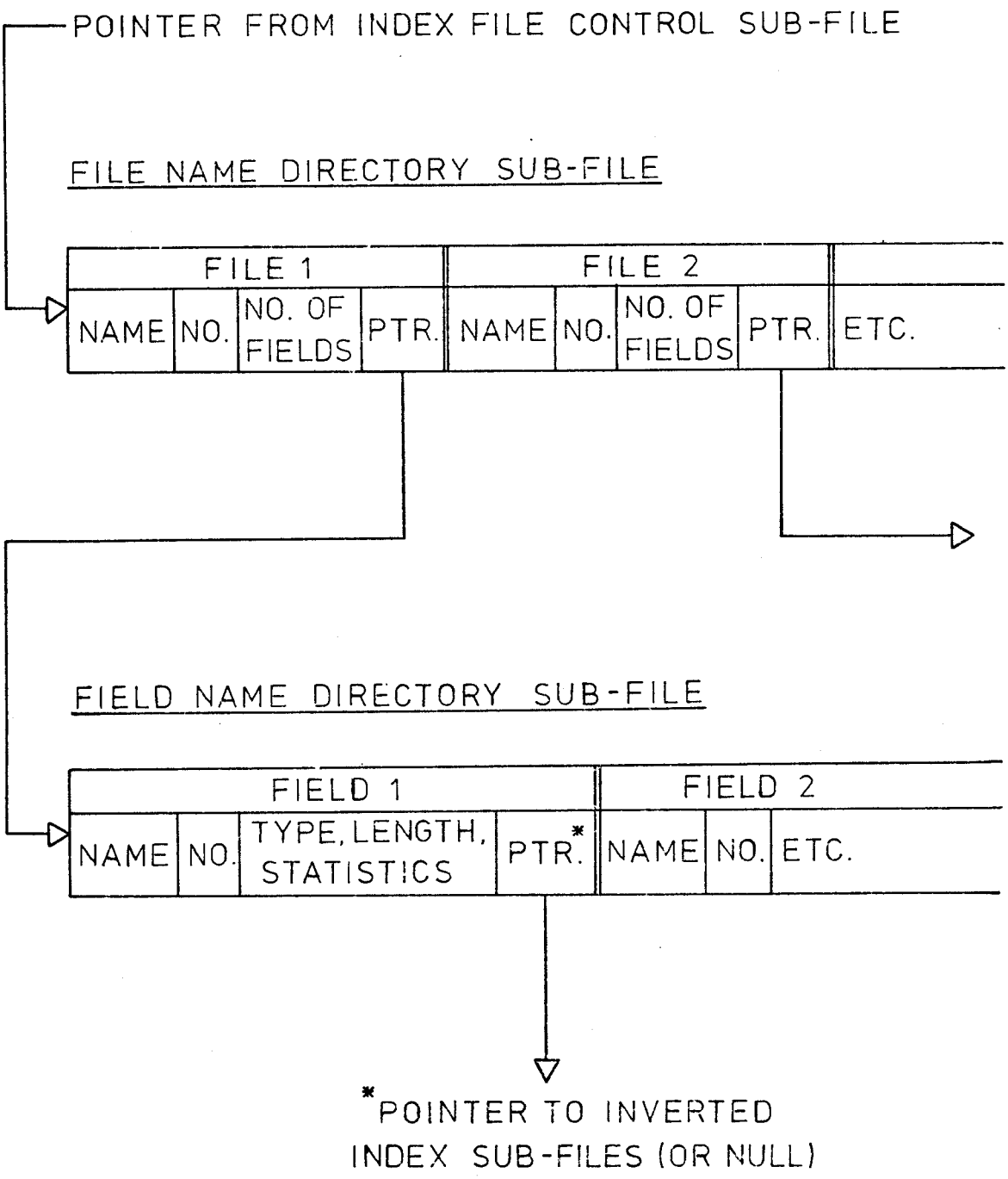
1. File Name Directory Sub-File.
   This sub-file is pointed to from the overall file control information contained on the Index File Control Sub-File. It contains the following information for each file description stored:
   (a) File Name.
   (b) File Number.
   (c) Number of Fields.
   (d) A Pointer to the Field Name Directory Sub-File for this file name.
2. Field Name Directory Sub-File.
   This sub-file is pointed to from the parent file name entry in the File Name Directory Sub-File. It contains the following information for each field within the record:
   (a) Field Name.
   (b) Field Number.
   (c) Field Type.
   (d) Field Length.
   (e) Run Statistics.
   (f) A Pointer to the Inverted Index Sub-Files for this field (if indexed). Otherwise, a null pointer is stored.

3.3.2. Use of the File Description Directories

Accessing to Inverted Indexes always takes place using the following procedures:

1. A file and associated field name is provided.
2. From a pointer in the Index File Control Sub-File, the File Name Directory Sub-File is accessed.
3. A search is made through the File Name Directory Sub-File for the required entry.

FIGURE 3.4: FILE DESCRIPTION DIRECTORIES



3.3.2. Use of the File Description Directories contd

4. If found, the associated Field Name Directory Sub-File is accessed.
5. A search is made through the Field Name Directory Sub-File for the required entry.
6. If found and the field is indexed, a search of the Inverted Index Sub-Files may commence.

3.3.3. File Description Processing Programs

The user has the ability to add and delete file descriptions from the Index File at will. To this purpose the following processing and utility programs should be provided:

1. File Description Addition Program.
   The user may store a file description onto the Index File. This contains the file name and associated field information. At this point, no indexes have been created.
2. File Description Deletion Program.
   The user may decide to delete a file description from the Index File. This deletion process not only deletes the file name and associated field name information, but will also delete any inverted indexes in existence associated with that file name. The space previously occupied by the sub-files is returned to system availability.
3. File Description Display Utility.
   A report may be produced on demand which displays the current File Description Directories status: file descriptions presently stored in the system together with associated field information.

3.4. Inverted Index Processing

Having loaded and stored file descriptions on the Index File, it is now necessary to provide the system user with the ability to determine the degree of inversion of the selected file (i.e. how many fields are to be indexed). Conversely, the user should also be provided with the opportunity to decrease the degree of inversion by deleting indexes for specified fields. Thus it should be possible to vary the fields that are indexed with changing research requirements.

3.4.1. Storage of Inverted Indexes on the Index File

Within the General Purpose Inverted Indexing System, the storage of a particular inverted index may be carried out by the building of a

3.4.1. Storage of Inverted Indexes on the Index File contd

set of sub-files. Searching through the sub-files is carried out by a form of Skip Sequential organization. An inverted index is built on two levels (see Figure 3.5) as follows:

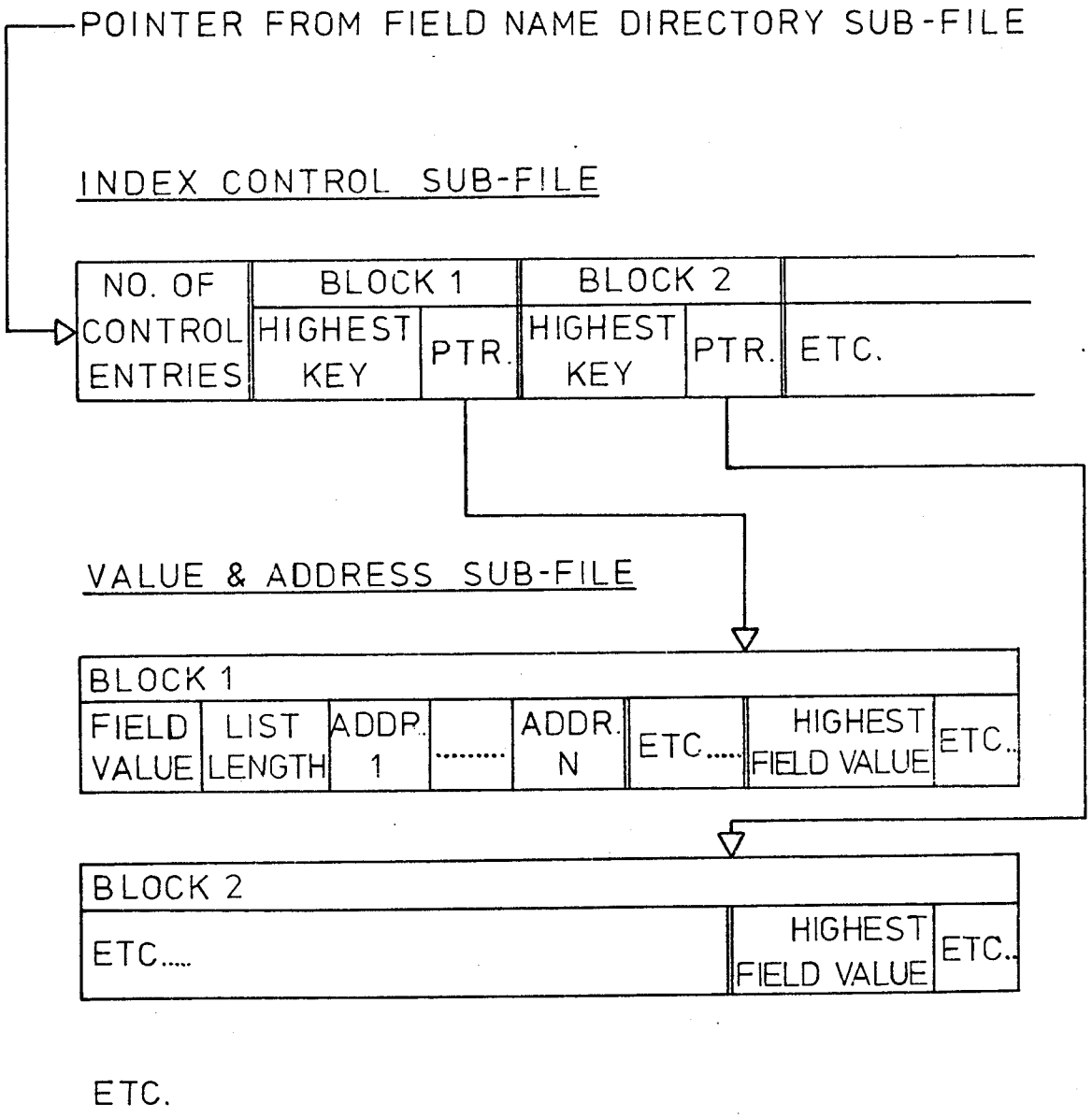
1. Index Control Sub-File.
   This sub-file is accessed by a pointer in the field name entry in the Field Name Directory Sub-File with which it is associated. It provides the top level in a two-level tree structure for index searching. The first field in the Index Control Sub-File contains a count of the number of control entries. This is followed by a number of entries each containing the following information:
   (a) The highest key (or field value) on a physical block at the second level of indexing - the Value and Address Sub-File.
   (b) A pointer to the location within the physical block of the key stated above in (a).
2. Value and Address Sub-File.
   This sub-file is accessed from the Index Control Sub-File via a series of pointers. The first pointer in the Index Control points to the start of the Value and Address Sub-File, while the other pointers indicate the highest field values (or keys) on a number of physical blocks through the Value and Address Sub-File. This sub-file is so named because it contains the base indexing data by which a specified field value may be connected to a Data File record. Each entry in the Value and Address Sub-File contains the following information:
   (a) Field Value.
   (b) No. of records in Data File containing field value - list length.
   (c) A list of addresses of all records on the Data File which contain that field value.

3.4.2. Searching an Inverted Index

Searching of a specified inverted index is carried out by the following processes:

1. Determining whether an inverted index exists for a specified file and field name.
2. Searching the Index Control Sub-File until the appropriate control entry is reached.

FIGURE 3.5: INVERTED INDEX PROCESSING



3.4.2. Searching an Inverted Index contd

3. Using the pointer contained in the control entry, access the appropriate physical block in the Value and Address Sub-File.
4. Search Value and Address Sub-File until the appropriate field value is reached.
5. Return a list of record addresses stored for the appropriate entry.

3.4.3. Specification of Search Criteria

The required inverted indexes having been created, the user should then be able to specify various search criteria and have a list of record addresses satisfying those criteria returned.

The "Basic" query that the system should be able to answer may be defined as a Selection Condition in the following form:

SELECT filename fieldname operator fieldvalue

where:

filename is the name of the Data File to be searched.

fieldname is the name of the field to be searched within the above file name.

operator specifies the type of search that is to be made. Suggested operators for a general purpose system might be Equal, Not Equal, Greater Than, Greater Than or Equal, Less Than, and Less Than or Equal.

fieldvalue is the value against which the inverted index is to be searched.

From the "Basic" query or Selection Condition, a "Complex" query may be defined in which more than one Selection Conditions are linked together. The Selection Conditions would be linked by a logical operator such as "AND" or "OR". For example, one "Complex" query might be defined as follows:

Selection Condition 1 AND Selection Condition 2 AND
Selection Condition 3 OR Selection Condition 4 AND
Selection Condition 5.

The "OR" operator represents a delimiter between a number of "AND" processes which will be considered together. Almost all "Complex" selection queries may be answered using this method.

3.4.4. Inverted Index Processing Programs

The basic requirements for a user to be able to process inverted indexes will be the ability to create and delete inverted indexes on specified

3.4.4. Inverted Index Processing Programs contd

fields, perform selections on "Complex" queries, and display the status of particular inverted indexes. Therefore, the following processing and utility programs should be provided:

1. Create an Inverted Index.
   On provision of a specified file and field name which are contained in the File Description Directories, the basic Data File should be accessed to create an inverted index. Safeguards should be provided to ensure that creation of the inverted index does not overflow the current size of the Index File.
2. Delete an Inverted Index.
   The inverted index for a specified file and field name may be deleted on demand by the user, and the space previously occupied returned to system availability.
3. "Complex" Query Processing.
   The user may specify a "Complex" query which initiates a search of the Index File and returns a list of all Data File record addresses which satisfy the search criteria.
4. Inverted Index Display Utility.
   A report may be produced on demand which may display information from an inverted index for a specified field name.

3.5. Summary and Further Design Considerations

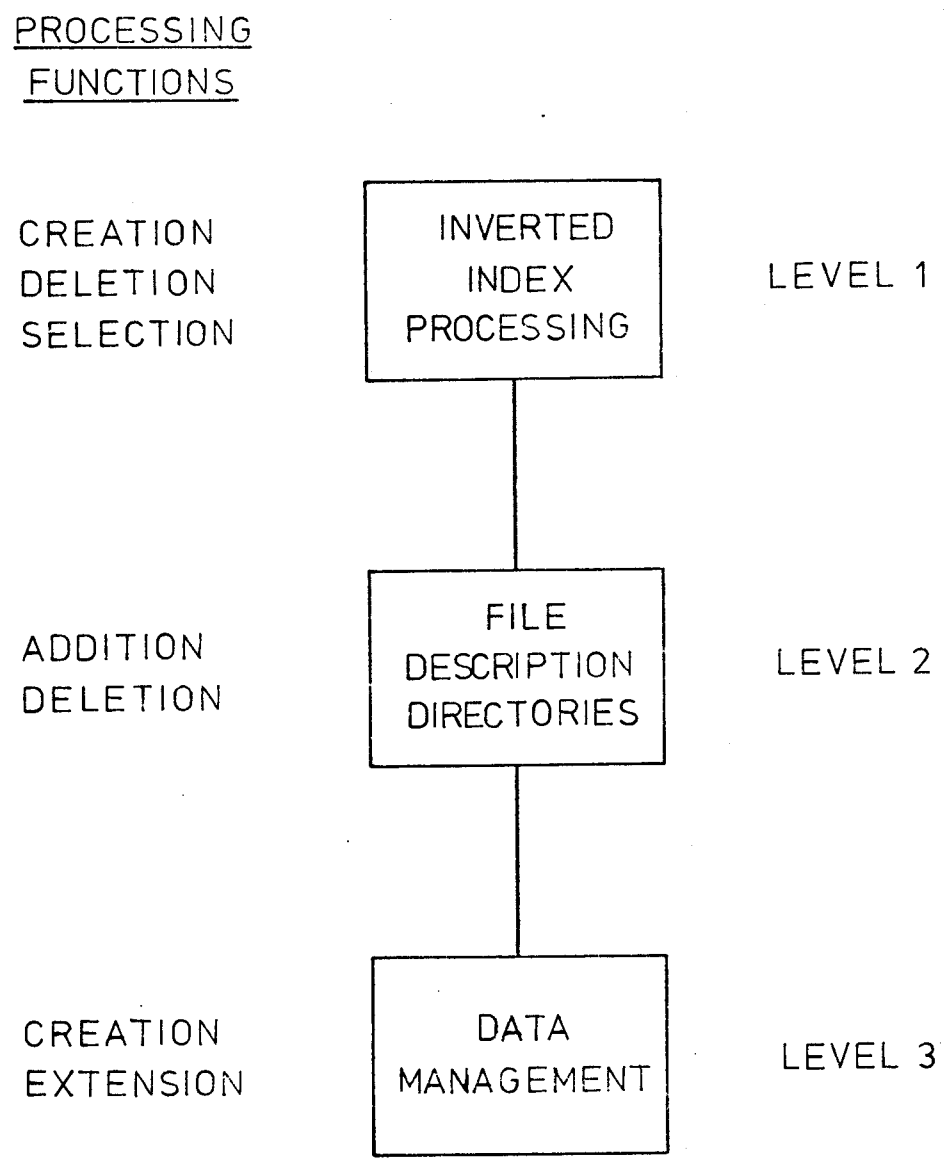
The design philosophy for a General Purpose Inverted Indexing System may be defined as a hierarchy of processing functions (See Figure 3.6). Various processing functions may be defined at the various levels; however the main access by the user will take place at Level 2 where file descriptions are added and deleted, and at Level 1 (the top level) where inverted indexes are created, searched and deleted. Apart from the creation and extension of the Index File at Level 3, the Data Management routines are only accessed through upper levels.

The scope of the design might be stated as the maintenance of a General Purpose Inverted Index. It should be borne in mind that the system might reasonably be incorporated into a larger system with the following interfaces:

1. The passing of a set of commands (together with associated parameters) to Inverted Index processing functions.
2. The return of address lists to the main system for further processing.

Provision should be made for performance monitoring of system functions.

FIGURE 3.6: LEVELS OF PROCESSING



NOTE:
DISPLAY UTILITIES AT ALL LEVELS

CHAPTER 4: SYSTEM DESCRIPTION

4.1. Introduction

The General Purpose Inverted Indexing System is written in PL/1 and operates on the Northumbrian Universities Multiple Access Computer (NUMAC) IBM System/360 Model 67 under control of the Michigan Terminal System (MTS). It contains 24 modules - 10 Main Programs and 14 Subroutines or Macros. A naming convention for the modules has been established:

1. Each module has a 6 character name under which it is stored in Object or other form on the MTS system.
2. The first 2 letters of the module name are always IX.
3. The third letter of the module name gives the type. A number of different module types have been specified:
   - M Macro - A group of PL/1 statements previously stored and invoked at Compile time by the PL/1 Preprocessor.
   - D Data Management Routines - This set of modules controls and processes all disk I/O on the system.
   - P Processing Routines - Modules which use the Data Management Routines for processing of Directories and Indexes.
   - U Utility - A number of Utility Programs which perform various system functions.
   - C Conversion - This type of module would generally be user-written. It would be required if the original Data File was not in a tabular form, and therefore must be converted.
4. The last 3 letters of the module name uniquely identify the module.

Within the modules, a convention has been established in that all PL/1 External names are prefixed by the letter Z. Thus, the Procedure which is stored under the name IXDOPN, would have the Entry Point ZDOPN.
In this chapter, the following information is presented:

1. Data Storage and Types.
2. Description of the System Modules.
3. Interrelation of the System Modules.
4. MTS Dependencies.

A basic assumption is made that the reader is familiar with the PL/1 programming language. Where reference is made to the use of MTS, the facilities provided by the system are fully discussed to provide the non-MTS user with a concept of the functions being used.

4.2. Data Storage and Types

As explained in the Design Philosophy, the Index File is stored on a disk file which is made up of a number of directly addressable fixed length blocks. A number of these blocks from 1 to n may be linked to form a sub-file, in which processing is always seqentially forward. Five types of sub-file may be identified:

1. Index File Control Sub-File.
   This sub-file comprises 1 block (the first on the Index File) which contains control information.
2. File Name Directory Sub-File.
   Contains the names of all file descriptions which are stored on the Index File. It is designed so that extra file names may be readily added; also file names can be deleted. The sub-file is pointed to from the Index File Control Sub-File.
3. Field Name Directory Sub-File.
   Contains the names and characteristics of all fields which are associated with a particular file name. The sub-file is pointed to from the appropriate file name entry in the File Name Directory Sub-File.
4. Index Control Sub-File.
   The first level of inverted index. Contains a number of entries, each of which point to a particular position on the Value and Address Sub-File where the inverted index is actually stored. The sub-file is pointed to from the appropriate field name entry in the Field Name Directory Sub-File.
5. Value and Address Sub-File.
   This sub-file contains the actual inverted index. Each entry contains a field value, the number of data records in which the value is found, and a list of record addresses. It is accessed through a number of pointers contained in the Index Control Sub-File.

The control over access to all sub-files is carried out by using a disk pointer which is declared in PL/1 as follows:

```
DCL 1 ZPOSN EXTERNAL,
      2 BLOCK BINARY FIXED(31,0),
      2 OFFST BINARY FIXED(15,0);
```

Thus, when any Index File processing operation is to take place, BLOCK is set to the block required, while OFFST is set to the position required within the block. After completion of the processing operation,

4.2. Data Storage and Types contd

BLOCK and OFFST are set by the system to point to the next position in the sub-file in process. Thus sequential access to sub-files is provided.

The physical storage of information on each of the sub-files described above is now examined in more detail.

4.2.1. Index File Control Sub-File

The control information stored on this sub-file may be divided into two portions. The first considers the overall control of the Index File including space allocation and recovery, while the second covers the processing of the File and Field Name Directory Sub-Files.

(a) Index File Control Record.

Length 16 bytes. Start position is Block 1, Offset 1.

Field Name	Description	Initial Value
FAVAL	First available block on file	3*
NOBLKS	Total number of blocks on file	Entered
NABLKS	Number of available blocks	NOBLKS - 2
DASIZE	Size of data area	251**
FIXPTR	Pointer to File Name Directory	2*

*The File Name Directory Sub-File is initially defined as starting at the first available block in the blank Index File. Therefore, Block 2 is allocated for this purpose, while the first available block is Block 3.

**The current implementation of the system under MTS uses 255 byte blocks. Thus the first 4 bytes of each block are reserved for a chain pointer.

(b) Directory Control Record.

Length 10 bytes. Start position is Block 1, Offset 17.

Field Name	Description	Initial Value
DINUM	Total file name entries (existing or deleted)	0
DLNUM	No. of deleted file name entries	0
DLPTR	Pointer to 1st deleted entry	0
FILEN	Length (bytes) of a file name entry	28
FLDLN	Length (bytes) of a field name entry	32

4.2.2. File Name Directory Sub-File

The File Name Directory Sub-File contains the names of all the file names stored on the Index File at any one time. It is accessed from

4.2.2. File Name Directory Sub-File contd.

the Index File Control Record and controlled from the Directory Control Record. Within the Directory Control Record, a count of the total number of file name entries (whether existing or deleted) is kept. The addition of a new file name either involves adding a new entry to the end of the existing sub-file or inserting the new entry in a space left by a previously deleted entry. The format of a File Name Directory entry is as follows:

Length 28 bytes. Start position of sub-file is Block 2, Offset 1.

Field Name	Description	Initial Value
FDPTR	Pointer to Field Name Directory	Set by system
FLNUM	File Number (or deletion pointer)	0
NOFLDS	Number of Fields	Entered
FLNAM	File Name	Entered

Searching of the File Name Directory is carried out by means of a sequential scan.

4.2.3. Field Name Directory Sub-File

The Field Name Directory Sub-File for a particular Data File is linked to that file name in the File Name Directory Sub-File. It contains a number of field name entries, the number being stored in the associated file name entry. The format of each field entry is:

Field Name	Description	Initial Value
FDIXPTR	Pointer to Inverted Index (if existing)	-1 (Null)
FDNUM	Field Number	-
FDTYP	Field Type	-
FDLEN	Field Length	-
FDSTAT	Usage Statistics (not in use)	0
FDNAM	Field Name	-

The length of each entry is 32 bytes.

The storage of field values in the Inverted Index is dependent on provision within the field name entry of two items of information - Field Type and Field Length. It is defined that all field values must be fixed length and that the allowable types are:

Type No.	Description	Length(bytes)	PL/1 DCL
1	Character	1 - 255	CHAR(n)
2	Halfword Binary	2	BIN FIXED
3	Fullword Binary	4	BIN FIXED(31,0)
4	Single Precision Floating Point	4	BIN FLOAT(21)
5	Double Precision Floating Point	8	BIN FLOAT(53)

4.2.3. Field Name Directory Sub-File contd

If an inverted index for a particular field does not exist, the pointer to the index is set to null (-1).

4.2.4. Index Control Sub-File

The Index Control Sub-File is a top level index for an Inverted Index created for a specified field. It is constructed as follows:

(a) No. of Entries on Index Control Sub-File.

A field of 2 bytes in length containing the number of entries.

(b) Index Control Entry

Each Index Control Entry contains 3 fields:

1. Field Value. Length and type dependent on Field Type specified. Contains the highest value on a Value and Address Block.
2. Block Address. Block number on which field value is stored - fullword binary.
3. Offset Position. Start byte of field value on block referenced - halfword binary.

A search of the Inverted Index starts with a scan of the Index Control Entries. When the correct entry is reached, a branch is made to the required Block and Offset position in the Value and Address Sub-File.

4.2.5. Value and Address Sub-File

There is one entry in the Value and Address Sub-File for each unique field value for the field being indexed, Each entry contains two values plus a list:

1. Field Value. Length and type dependent on Field Type specified.
2. No. of Addresses. The number of data records which contain the field value specified - fullword binary.
3. List of Addresses. A list of record addresses. In this implementation, these are sequential (or direct) record numbers within the Data File. Fullword binary.

4.3. Description of the System Modules

As stated in 4.1., each module within the General Purpose Inverted Indexing System is uniquely identified. The uses of the modules can be broken down into 4 major system functions:

1. Data Management.
2. Directory Processing.
3. Inverted Index Processing.

4.3. Description of the System Modules contd

4. Data Conversion.

Each of these system functions in relation to the modules used is examined in more detail in the following sub-sections.

4.3.1. Data Management

1. Module Name IXMSPM
   Module Type Macro - PL/1 Declarations.
   Module Description Data Management Areas and Variables.
   PL/1 Declarations are made for disk I/O areas and processing as well as the Index File Control Record. These statements are included in other modules by the use of the PL/1 Preprocessor statement %INCLUDE.

2. Module Name IXUSTP
   Module Type Utility - Main Program.
   Module Description Set up a Blank Index File.
   Before any processing can be carried out on the Index File, it is necessary to set up the blank file on disk. This program sets up the File Control Sub-File on Block 1, as well as a blank File Name Directory Sub-File on Block 2. It processes one input command - the total number of blocks to be provided on the Index File. After setting up control information, an availability chain through all available blocks is initialized.

3. Module Name IXUSTX
   Module Type Utility - Main Program.
   Module Description Extend an Existing Index File.
   In the course of use of the Index File, it may be found that there is insufficient space available to accomodate all the required Inverted Indexes. This program enlarges an existing Index File by chaining on extra available blocks. It receives one input command - the number of blocks to which the file is to be extended. The availability chain is extended from the existing blocks through the added blocks and the control information is updated.

4.3.1. Data Management contd

4. Module Name IXUSTR
Module Type Utility - Main Program.
Module Description Display Index File Status.
This program accesses the Index File Control Record and displays basic file status information such as the block size, data area size, total number of blocks and number of available blocks.

5. Module Name IXDOPN
Module Type Data Management Routine - Subroutine.
Module Description Open the Index File.
During processing of the Index File, the control information contained in the Index File Control Sub-File is held in main memory for fast access. This routine is called at the beginning of a Directory or Index Processing run to open the Index File, load control information. and initialize variables.

6. Module Name IXDCLS
Module Type Data Management Routine - Subroutine.
Module Description Close the Index File.
At the end of a Directory or Index Processing run, the control information contained in main memory is written back to disk together with any modifications made during the run. This routine is called at the end of the run to restore the control information to the Index File Control Sub-File and close the Index File.

7. Module Name IXDCRE
Module Type Data Management Routine - Subroutine.
Module Description Create a New Sub-File.
During Directory or Inverted Index Processing, it may become necessary to create new sub-files; e.g. when a set of field names is entered with a new file name, or when a new inverted index is to be created. This routine creates a sub-file of length 1 block. The block address is returned to the calling program in the system position indicator ZPOSN. The availability chain is adjusted to branch round the block just removed to form the sub-file.

4.3.1. Data Management contd

8. Module Name IXDDST
Module Type Data Management Routine - Subroutine.
Module Description Destroy an Existing Sub-File.
As for sub-file creation, it may also become necessary at various times to destroy a sub-file; e.g. if an inverted index is deleted. This routine is called with the system position indicator ZPOSN containing the address of the first block in the sub-file to be deleted. The blocks thus released are returned to the Index File availability chain.

9. Module Name IXDREA
Module Type Data Management Routine - Subroutine.
Module Description Read N Characters from a Sub-File.
Data is read from any sub-file on the Index File using this routine. The required procedure for calling the routine is as follows:

(a) Set the system position indicator ZPOSN to the specific block and offset position within the sub-file where the read is to start.

(b) Pass to IXDREA as parameters the number of characters to be read, together with an I/O area into which the results of the read operation are to be put.

On completion of the read, the system position indicator is set to the next position following the data read.
The key consideration in the use of this routine is that it provides data as requested by the calling routine irrespective of the physical storage of that data. For example, some data may be requested which is contained partly on one block and partly on the next block in the sub-file. The read routine will assemble and concatenate the portions of the data to be returned intact to the calling routine.

10. Module Name IXDWRT
Module Type Data Management Routine - Subroutine.
Module Description Write N Characters to a Sub-File.
Data is written to any sub-file on the Index File by this routine. The required procedure for calling the routine is as follows:

(a) Set the system position indicator ZPOSN to the specific block and offset position within the sub-file from where the write is to start.

4.3.1. Data Management contd

(b) Pass to IXDWRT as parameters the number of characters to be written, together with an I/O area from which the input to the write operation is to be taken.

On completion of the write, the system position indicator is set to the next offset position following the data moved. If the write request involves going beyond the current boundary of the sub-file, the sub-file is automatically extended.
This routine operates in the same way as the read routine, in that data is written as requested irrespective of the physical storage of the data.

11. Module Name IXDMOV
Module Type Data Management Routine - Subroutine.
Module Description Move the Sub-File Pointer.
This routine is utilized by the routines IXDREA and IXDWRT to move the system position indicator ZPOSN. It processes the condition when data being read or written across a block boundary.

12. Module Name IXDOSF
Module Type Data Management Routine - Subroutine.
Module Description Add N Blocks to a Sub-File.
When a sub-file is being extended, this routine is called to add a specified number of blocks to a sub-file. The blocks are obtained from the system availability chain, which is adjusted to branch round the blocks removed to add to the sub-file.

4.3.2. Directory Processing

1. Module Name IXMDPM
Module Type Macro - PL/1 Declarations.
Module Description File and Field Name Directory Areas.
PL/1 Declarations are made for the Directory Control Record for control of the File and Field Name Directory Sub-Files as well as the definition of storage for current file and field name information.

2. Module Name IXPFND
Module Type Directory and Index Processing - Subroutine.
Module Description Find a File Name on the Directory.
At the start of all the Directory and Inverted Index processing programs, a file name is read in. This routine determines whether

4.3.2. Directory Processing contd

the file name entered is on the File Name Directory Sub-File. If a match is found, the Directory entry number and position is returned. If no match is found, the value zero and the position at the end of the directory is returned.

3. Module Name IXPNEW
Module Type Directory Processing - Main Program.
Module Description Add a File Description to the Directory.
This program loads a new file name together with associated field names and descriptions onto the File and Field Name Directories.
The input to the program comprises:

(a) The file name and number of fields on each record.

(b) A number of cards as stated in the number of fields above, each of which contain a field name, type and length.
Allowable field types are:

Type No.	Storage	Length
1	Character	1 - 255
2	Halfword Binary	2
3	Fullword Binary	4
4	Single Precision Floating Point	4
5	Double Precision Floating Point	8

All fields processed must be fixed length. File and field names may be up to 20 characters in length.
The input file and field information is edited for:

(a) Potential duplication of file names.

(b) Invalid field type.

(c) Field length inconsistent with field type.

If there are no errors in the input data, a Field Name Directory Sub-File is created and loaded with the field name descriptions, followed by the insertion of a File Name Directory entry. This is either at the end of the existing directory or in available space left by a previously deleted file name entry.
The program is written so that multiple file descriptions may be entered in one run.

4. Module Name IXPDEL
Module Type Directory and Index Processing - Main Program.
Module Description Delete a File Description from the Directory.
This program deletes a file name entry from the File Name Directory together with the associated Field Name Directory Sub-File and

4.3.2. Directory Processing contd

attached Inverted Indexes.

The input to the program is the file name to be deleted. After a check to see if that file name exists on the directory, the following procedure is followed:

(a) Any Inverted Indexes referenced from the Field Name Directory Sub-File are deleted.

(b) The Field Name Directory Sub-File is deleted.

(c) The file name entry in the File Name Directory Sub-File is deleted.

In all the above steps, the space recovered is returned to system availability.

The program is written so that multiple file descriptions with associated indexes can be deleted in one run.

5. Module Name IXUDRP
Module Type Utility - Main Program.
Module Description Display Contents of File and Field Name Directories.

This program is run to display in a formatted printout all the current file descriptions stored on the File Name and associated Field Name Directory Sub-Files. Information displayed for each file includes:

File Name, Number of Fields.

For each Field:

Field Name, Type, Length, Whether Indexed and Run Statistics.

4.3.3. Inverted Index Processing

1. Module Name IXPIXC
Module Type Index Processing - Main Program.
Module Description Create an Inverted Index.

The Inverted Index Creation Program operates in two modes - first, it may be used to estimate the space requirements on the Index File for an inverted index before it is created; and second, it is used to actually the required inverted index. The input and processing for both modes is as follows:

(a) Estimation of Index File Space Required.

The input to the Index Creation program to perform an estimate comprises the file name, the processing option 'EST', followed by a series of field names. For each field name a calculation is carried out based on the number of

4.3.3. Inverted Index Processing contd

records in the Data File (RECCTR), the field length (IFLC) and the data area size in the Index File blocks (DASIZE) as follows:

```
RECVA = 1 + ((RECCTR * (IFLC + 8)) / DASIZE);
RECIC = 1 + ((2 + (RECVA * (IFLC + 6))) / DASIZE);
RECTOT = RECVA + RECIC;
```

where RECVA is the no. of Value and Address blocks,
RECIC is the no. of Index Control blocks, and
RECTOT is the total no. of blocks.

The calculation produces in RECTOT the maximum number of blocks that might be used in the index to be created. This assumes the worst possible case where every field value within the specified field is unique, and hence will generate one index entry.

When Estimate processing is carried out for a number of fields, the total number of blocks required is accumulated and printed out at the end of the run. Thus the potential space requirements for a set of indexes based on one Data File can be evaluated in one run.

(b) Creation of an Inverted Index.

The input to the Index Creation Program to create an inverted index consists of a file name, a processing option and one field name. The available processing options 'NO' and 'YES' refer to the action the Index Creation Program must take if it finds an existing index against the field name specified. If the processing option is 'NO', this indicates that any existing index against the field name specified is to be deleted before the new index is created; while, if the processing option is 'YES' and an existing index is found, the run is to be cancelled before index creation takes place.

The Creation Program divides into four stages:

Stage 1. The input file, processing and field information is processed within the File and Field Name Directories to validate and obtain control information such as field type, length and position within the data record.

Stage 2. The Data File is read and the field name and record address abstracted from each data record into a temporary file. At this point, this calculation of the maximum index size is made (as described above in (a)).

4.3.3. Inverted Index Processing contd

If there is not enough space on the Index File to hold the index to be created, the run is terminated.

Stage 3. The temporary file of indexing information is sorted into ascending sequence by the following sort fields - Fiedl Value (Major) and Record Address (Minor). The output from the sort is stored on another temporary file.

Stage 4. Here the new Inverted Index is created. First, if a previous index exists, it is deleted. Then, if there is not enough space on the Index File to hold the new index, the run is terminated.

Two sub-files are created - Index Control as well as Value and Address, into which the indexing information is processed. The Index Control Sub-File holds a number of field values which are highest on Value and Address blocks, together with pointers to the actual entries. Note that there is not necessarily one Index Control entry for each Value and Address Block - for example, a Value and Address Block might only contain a portion of a long Record Address list without any field value on that block.

At the end of the run, the Field Name Directory is updated with a pointer to the Inverted Index and the run terminated.

Measurements of the CPU time used in various portions of the run are taken and printed out at the end of the run.

2. Module Name IXPIXD
Module Type Index Processing - Main Program.
Module Description Delete an Inverted Index.

This program is invoked when it is required to delete a specific inverted index. The file and field names are entered and edited against the File and Field Name Directories. If there is an existing index, the sub-files comprising it (Index Control together with Value and Address) are deleted and the space recovered returned to system availability. Lastly, the Field Name Directory pointer to the Inverted Index is set to null and the run is terminated.

4.3.3. Inverted Index Processing contd

3. Module Name IXPCMP
Module Type Index Processing - Subroutine.
Module Description Compare Two Field Values and Return a Result.
In processing an Inverted Index during a search, it is necessary to compare various field values e.g. the input search value against a field value on the Inverted Index. This routine is called from IXPSEL where the search through a specified index is made. Input to the routine comprises the field type being processed (passed as an EXTERNAL variable), two fields containing values to be compared and a field in which the comparison result is returned. Processing in the routine takes place as follows:

(a) A branch is made to varying portions of the routine according to the field type passed (e.g. Character, Fixed Binary or Floating Point - see allowable Data Types).

(b) A comparison between the two fields (passed as parameters) is made using the appropriate field type.

(c) The compare result field (also passed as a parameter) is then loaded with a value indicating the result of the comparison:

-1	Less Than
0	Equal
1	Greater Than

(d) A return is made to the calling routine.

4. Module Name IXPEDT
Module Type Index Processing - Subroutine.
Module Description Edit Selection Input Data.
The processing of selection queries calls for the ability to answer "Complex" queries, in which a number of selection conditions are combined by AND or OR logical processes. To be able to perform these "Complex" selections, it is first necessary that all the Selection Conditions be gathered together and edited before a search is commenced. This editing process takes place within this module which is invoked by the main "Complex" selection processing program IXPIXS.
Each Selection Condition is entered in the form:

fieldname operator fieldvalue

4.3.3. Inverted Index Processing contd

the file name having been entered beforehand. The file and field names are checked for validity against the File and Field Name Directories. Valid operators are:

Equal (EQ), Not Equal (NEQ), Greater Than (GT), Greater Than or Equal (GTE), Less Than (LT), and Less Than or Equal (LTE).

The field value entered in character form is converted into the appropriate data type and stored.

Each Selection Condition entered is stored as an entry in an array of Selection Conditions (ZSTAB). The contents of each entry in the array are as follows:

(a) Search Condition Number. Each entry is sequentially numbered from 1 to n.

(b) Level Number. The initial level number entered for each entry is 1. However, when an 'OR' logical process is entered following a Selection Condition, the level number is incremented by 1.

(c) Index Block Number. This is the address of the Index Control Sub-File for the Inverted Index to be searched.

(d) Search Type. A number equivalent to the appropriate operator above (e.g. Equal is Search Type 1) is entered.

(e) Field Type. The field type is used in comparison processing by subroutine IXPCMP.

(f) Field Value Length. This is the field length to be used in allocating storage for an index search.

(g) Field Value Pointer. A pointer to the area where the field value is stored.

If any errors are detected in editing the input selection information, an edit flag is set which prevents further processing past the edit stage.

5. Module Name IXPSEL

Module Type Index Processing - Subroutine.

Module Description Select Records according to Input Parameters.

This routine receives as a parameter an entry number in the array ZSTAB of Selection Conditions generated in IXPEDT. From the contents of the array entry, a decision is made as to what type of search is required on the inverted index specified.

4.3.3. Inverted Index Processing contd

The possible search types are:

Type No.	Search Type
1	Equal
2	Not Equal
3	Greater Than
4	Greater Than or Equal
5	Less Than
6	Less Than or Equal

From this point a branch is made to portions of the program which search the index in the required way. There are in fact 4 portions which perform the following types of select processing:

Equal

Not Equal

Greater Than / Greater Than or Equal

Less Than / Less Than or Equal

Within each type, the method of inverted index search differs. The processes involved in each type are as follows:

(a) Equal.

A search is made through the Index Control Sub-File on the search value entered. When a point is reached at which the field value on the Index Control Sub-File is equal to or less than the search value, a branch is made to the Value and Address Sub-File and the search continued until a match is either made or not made. The record addresses found are then returned in a temporary file.

(b) Not Equal.

A branch is immediately to the Value and Address Sub-File where a sequential search is carried out. Every field value not equal to the search value is accessed and the record addresses stored on a temporary file.

(c) Greater Than / Greater Than or Equal.

A search is made through the Index Control Sub-File using the search value entered. When a point is reached at which the field value on the Index Control Sub-File is equal to or less than the search value, a branch is made to the Value and Address Sub-File and the search continued until a match is either made or not made. A sequential search from the match/no match point in the Value and Address Sub-File is then carried out and all field values answering the condition entered are accessed and the record addresses stored on a temporary file. The search continues until the

4.3.3. Inverted Index Processing contd

end of the Value and Address Sub-File is reached.

(d) Less Than / Less Than or Equal.

As for Not Equal processing, a branch is immediately made to the start of the Value and Address Sub-File, and a sequential search carried out. Every field value up to a match or no match (depending on the search condition) on the search value is accessed and the record addresses stored in a temporary file.

6. Module Name IXPIXS

Module Type Index Processing - Main Program.

Module Description Select Records using Inverted Indexes.

This program is invoked for the purpose of answering "Complex" selection queries. A number of input Selection Conditions are entered to make up the content of a "Complex" query. The Selection Conditions entered are stored in an array (ZSTAB) together with any 'AND' or 'OR' logical processes specified. The editing of the input information and storage on the array is carried out by calling the IXPEDT subroutine. If errors are detected during the editing of input information, a flag is set in IXPEDT, which when examined in the main program terminates processing before index searching commences.

After editing of the input is complete, searching the various indexes may commence. To search each index, the array of selection conditions is accessed and an entry number passed to the IXPSEL subroutine which returns a list of record addresses for that search in a temporary file together with a count of how many addresses were found.

A number of temporary files are used in storage and processing of intermediate information. These are used as both input and output in 'AND' and 'OR' logical processing. The logic behind processing of "Complex" queries and its application in this program is as follows:

A "Complex" query may be regarded as a number of Selection Conditions separated by 'AND' and 'OR' logical processes. The 'OR' logical process may be regarded as a delimiter between a set of 'AND' logical processes. Thus the 'AND' processes are evaluated first, followed by the 'OR' processes.

For example, consider the following "Complex" query:

SC1 AND SC2 AND SC3 OR SC4 AND SC5.

where SCn are Selection Conditions. Selection Conditions 1-3 form

4.3.3. Inverted Index Processing contd

one group, while Selction Conditions 4 & 5 form another. The processing carried out by the program will be as follows:

(a) Call IXPSEL with SC1 and obtain a list of record addresses. (L1).

(b) Call IXPSEL with SC2 and obtain another list of record addresses (L2).

(c) Sort L1 and L2 together in ascending sequence to form a resultant list (L3).

(d) Perform 'AND' logical processing on L3 by reading and passing only those duplicated record addresses to an 'AND' list (L1).

(e) Call IXPSEL with SC3 and obtain a list of record addresses (L2).

(f) Repeat processes (c) and (d).

(g) 'OR' processing requested - end of 'AND' processing. Transfer contents of L1 to 'OR' list (L4).

(h) Repeat processes (a) to (d) for SC4 and SC5 to obtain another 'AND' list (L1).

(i) End of query. Add contents of L1 to 'OR' list (L4).

(j) Sort L4 into ascending sequence and store in L1.

(k) Perform 'OR' processing on L1 by reading and passing all unique record addresses (ignoring duplicates) to the final 'OR' list (L4).

(l) Display or pass back to enquirer the list of record addresses obtained.

At the end of the run, the CPU time used in the program is displayed broken down into times for the various sections of the program.

7. Module Name IXUIXL

Module Type Utility - Main Program.

Module Description Display Contents of Inverted Index.

This program is primarily designed for debugging the General Purpose Inverted Indexing system, although it may be also be invoked by a user. The input is a file and field name followed by a processing option. Depending on the processing option (ALL or other), a specified inverted index may be displayed in its entirety (Index Control and Value and Address Sub-Files), or the Index Control Sub-File only. The formatted printout displays a complete specified inverted index.

4.3.4. Data Conversion

The processing of Data Files by the General Purpose Inverted Indexing System is dependent on the data being available in a tabular form. If the basic data is not in this form, it must be converted by a user-written program.

For example, some experimental data used in work on this thesis was stored on card images in this form:

Master Card
Other Card(s)
Data Cards

There was one Master Card which might or might not be followed by one or two Other Cards together with a number of Data Cards. A tabular record would include unique information from one Data Card together with Master Card information which would be duplicated on multiple tabular records. A program was therefore written to convert the original data from its hierarchical form to a tabular form suitable for processing by the General Purpose Inverted Indexing System.

4.4. Interrelation of the System Modules

The method of linkage of the various system modules is shown in Figure 4.1. The module names across the top of the table are the Called Modules; while those down the side are the Calling Modules. Some modules are both calling and called, and therefore appear in both lists.

4.5. MTS Dependencies

The General Purpose Inverted Indexing System is written in PL/1 and operates on the NUMAC IBM System/360 Model 67 under the Michigan Terminal System (MTS). To use the MTS system, it was necessary to introduce various portions of system-dependent coding for the following purposes:

1. Storage and Retrieval of Directory and Index Information from the Index File on a direct access device.
2. Invocation of a Sort program from within the Index Creation and Selection modules.
3. Use of MTS routines to log the CPU time usage in processing Index Creation and Selection requests.
4. Invocation of an MTS System Command within the Index Selection module to scratch temporary work files when no longer required.

Each of these MTS system dependencies is examined in more detail in the following sections.

FIGURE 4.1: INTERRELATION OF SYSTEM MODULES

CALLING MODULES	CALLED MODULES													
	*IXMSPM	*IXMDPM	IXDOPN	IXDCLS	IXDCRE	IXDDST	IXDREA	IXDWRT	IXDMOV	IXDOSF	IXPFND	IXPCMP	IXPEDT	IXPSEL
IXUSTP	X	X												
IXUSTX	X	X												
IXUSTR	X		X	X										
IXUDRP	X	X	X	X			X							
IXUIXL	X	X	X	X			X				X			
IXDOPN	X	X												
IXDCLS	X	X												
IXDCRE	X			X										
IXDDST	X													
IXDREA	X			X					X	X				
IXDWRT	X								X	X				
IXDMOV	X									X				
IXDOSF	X			X										
IXPFND	X	X					X							
IXPNEW	X	X	X	X	X	X	X	X			X			
IXPDEL	X	X	X	X		X	X	X			X			
IXPIXC	X	X	X	X	X	X	X	X			X			
IXPIXD	X	X	X	X		X	X	X			X			
IXPEDT	X	X		X			X				X			
IXPSEL				X			X					X		
IXPIXS			X	X									X	X

*USING PL/1 PREPROCESSOR

4.5.1. Storage and Retrieval on the Index File

In the original design for the General Purpose Inverted Indexing System, it was intended that the possible block size within the Index File should be adjustable based on parameters input at Index File setup time. Thus, a particular implementor could select the maximum block size for his memory space available, thus decreasing Index Control Sub-File entries and the number of comparisons and disk accesses required for a search. However, on examination of facilities available to a PL/1 programmer for direct access on the MTS system, it was found that the only facilities provided performing the same functions as PL/1 Regional(1) were two subroutines IHEREAD and IHERITE which provide direct access to MTS Line Files.

The MTS Line File is primarily used by the on-line user for file storage with the facility for direct access to a specified line (or record). It was therefore decided to take the standard software provided so as to minimize time that might be spent in developing alternative access methods.

However, some basic limitations in the use of Line Files had to be accepted: These were:

1. An MTS record in a Line File (or line) may be variable in length but has a maximum allowable length of 255 bytes.
2. The maximum allowable size of an MTS Line File is 255 Pages (where 1 Page = 4096 bytes). If a block size equivalent to to the maximum record length stated above was selected, the maximum Index File size would be approximately 4000 blocks. Another way of stating this would be to say that the maximum allowable Index File size is approximately 1 million bytes.
3. To save space in an on-line environment, MTS performs a process known as Trimming on Line Files which removes all trailing blanks from a record, whether fixed or variable length. To be able to process fixed length Index File blocks in which control of space is carried out from within the General Purpose Inverted Indexing System, it was necessary to override this automatic facility. This is done by specifying all Line Files used as Not Trimmed (@-TRIM).

Therefore, the Index File consists of 255 byte fixed length blocks which may be accessed directly using the MTS PL/1 Subroutines IHEREAD and IHERITE.

The PL/1 Declarations for the Index File and the direct access routines are contained within the Data Management macro IXMSPM where they are

4.5.1. Storage and Retrieval on the Index File contd

defined as follows:

```
/* ***** MTS DEPENDENT FILE AND I/O DEFINITIONS ***** */
DCL ZINDX FILE UPDATE;
DCL (IHEREAD,IHERITE) ENTRY
      (,BIT(32),DEC FIXED(9,3),FILE);
DCL ZBUFF CHAR(255) EXT,
    ZMOD BIT(32) INIT ((32)'0'B) EXT,
    ZLINE DEC FIXED(9,3) EXT;
/* DEFINE DATA AREA SIZE FIELD FOR ALLOCATING STORAGE */
/* ***** INITIAL VALUE OF 255 IS MTS DEPENDENT ***** */
DCL ZSIZE BIN FIXED EXT INIT(255);
```

The PL/1 name of the Index File is ZINDX, ZBUFF is the block I/O area, and ZLINE is the Line or record number. ZMOD is a bit string used to select sequential or direct processing. To use direct processing, a certain bit of ZMOD must be turned on:

```
/* TURN ON INDEXED BIT OF ZMOD - ***** MTS DEPENDENT ***** */
SUBSTR(ZMOD,31) = '1'B;
```

The above declarations are then used for reading and/or writing to and/or from the Index File. For example, if it were required to read block/record/line number 1 into main memory, the following statements would be executed:

```
ZLINE = 1;
CALL IHEREAD(ZBUFF,ZMOD,ZLINE,ZINDX);
```

Subroutines IHEREAD and IHERITE are used in all Data Management routines as well as in Utilities IXUSTP and IXUSTX.

4.5.2. Invocation of a Sort from a PL/1 Program

At three points within the General Purpose Inverted Indexing System there is a requirement to sort records into a specified sequence. In the Index Creation program, it is required to sort records containing a Field Value and associated Record Address into ascending sequence; while in the Index Selection program, a sort is required in two places to obtain lists of record addresses in ascending sequence. A Sort program is provided by the MTS system which may be invoked from a PL/1 program as follows:

1. First, the Sort Parameters are defined:

```
DCL F1 FIXED BIN(31) INIT(1) STATIC,
    SORT ENTRY,
    PL1RC RETURNS (FIXED BIN(31));
```

4.5.2. Invocation of a Sort from a PL/1 Program contd

followed by a data structure defining the Sort parameters:
DCL 1 CSA STATIC, etc.

2. The Sort program is called from the PL/1 program:
CALL PLCALL(SORT,F1,CSA);
3. On return from the Sort program, a check is made for errors by interrogating the variable PL1RC which contains a sort return code:

```
IF PL1RC = 0 THEN;
             ELSE DO;
             /* ERROR PROCESSING */
             END;
```

The input to the sort is placed in a temporary file invoked in the Sort program; while the output from the sort is placed in another temporary file to be used by the calling program.

4.5.3. Logging of CPU Usage

To obtain some measure of performance on the MTS system, it is necessary to know the CPU time used in processing. MTS provides the ability to do this by means of the TIME subroutine (not to be confused with the PL/1 TIME function) which allows various time measurements to be taken from the MTS system. The option selected records the CPU time usage (in milliseconds) for a program at any stage measured from a datum point. This facility is used to measure the CPU time used in various portions of the Index Creation and Selection programs.
The method of use is as follows:

1. First, the areas for use by the TIME subroutine are defined:

```
/* CPU TIME PROCESSING AREAS */
DCL TIME ENTRY,
    P3 FIXED BIN(31) INIT(3) STATIC,
    CKEY FIXED BIN(31) INIT(1),
    CPR FIXED BIN(31) INIT(0),
    CCRES BIN FLOAT(21),
    CRES BIN FIXED(31),
    CRESA(5) FIXED BIN(31);
```

2. To measure the CPU time used between two points in the program, the following statements are executed:

```
/* GET AND STORE CURRENT CPU TIME STATUS */
CALL PLCALL(TIME,P3,ADDR(CKEY),ADDR(CPR),ADDR(CRES));
CRESA(1) = CRES;
```

4.5.3. Logging of CPU Usage contd

```
/* PL/1 STATEMENTS TO BE TIMED */
/* GET AND STORE CURRENT CPU TIME STATUS */
CALL PLCALL(TIME,P3,ADDR(CKEY),ADDR(CPR),ADDR(CRES));
CRESA(2) = CRES;
```

The current CPU time in milliseconds is stored at the various positions in the program in CRESA(1) and CRESA(2).

3. At the end of processing, the following statements are executed to get and display the CPU time used:

```
CCRES = CRESA(2) - CRESA(1);
CCRES = CCRES / 1000.0;
PUT FILE(REPORT) SKIP(2) EDIT
      ('EDIT CPU TIME(SECONDS) = ',CCRES)(A,F(10,3));
```

These statements retrieve the start and end CPU times, calculate the CPU time used in milliseconds, convert it to seconds, and then display the result.

4.5.4. Emptying Work Files following Sorts

The MTS Sort program invoked from the Index Selection program makes use of temporary MTS Line Files (temporary files are identified by a minus sign preceding the file name used).

A problem however arises in that these temporary files may be used more than once in the processing of a "Complex" selection query. However, the MTS system, despite OPEN and CLOSE statements in the PL/1 program, continues to maintain control information concerning the number of records in the temporary file as last used. This can cause errors if the temporary file is used more than once as input to a further sort because of discrepancies between the expected and actual number of records on the temporary file.

Therefore, after each sort, the control information is reset to zero by use of the MTS system command EMPTY. Processing of this command is carried out as follows:

1. First, the areas used are declared:

```
/* PARAMETERS TO EMPTY SELECT WORK FILE */
DCL F2 FIXED BIN(31) INIT(2) STATIC,
    CMD ENTRY,
    EMPTCMD CHAR(10) INIT('£EMPTY -W '),
    EMPTLEN BIN FIXED INIT(10);
```

2. The code to invoke the EMPTY command:

```
/* EMPTY PREVIOUS SELECT WORK FILE */
CALL PLCALL(CMD,F2,EMPTCMD,ADDR(EMPTLEN));
```

4.5.4. Emptying Work Files following Sorts contd

In the above case the temporary file is called -W.

A current limitation of the General Purpose Inverted Indexing System is that invocation of the EMPTY system command from a terminal requires a confirmatory response from the terminal user: 'OK'. This tends to slow down the speed of processing of a query and increases the entry requirements for the terminal user. Thus, the current system is being tested in batch mode only.

CHAPTER 5: SYSTEM OPERATION

5.1. Introduction

This chapter deals with the practical problems of use of the General Purpose Inverted Indexing System. The topics covered are as follows:

1. Conversion of Basic Data to Tabular Form.
2. Setup of the Blank Index File.
3. Loading of File Descriptions.
4. Estimation of Inverted Index Sizes.
5. Extension of Index File.
6. Creation of Inverted Indexes.
7. Processing of "Complex" Queries.
8. Deletion of Inverted Indexes.
9. Deletion of File Descriptions.
10. Report Utilities.

However, before considering each of these operations in detail, some points concerning the use of MTS and PL/1 should be explained.

5.1.1. MTS Job Control

Linking of PL/1 modules is carried out at execution time by use of the MTS RUN command. Together with the Indexing System modules, the MTS PL/1 Library must also be linked. This linkage is simply accomplished by entering the names of the modules to be linked separated by a plus sign. Thus, the linkage of 3 Indexing System modules together with the PL/1 Library would be accomplished by the following command:

```
£RUN IXUSTR+IXDOPN+IXDCLS+*PL1LIB etc.
```

The system commands are normally entered on punched cards. If a command is longer than one cards, it might be extended onto other cards by punching a minus sign in card column 80. In the examples shown, if the command is longer than one line, the minus sign is used to indicate a continuation.

5.1.2. Standard Logical Files

Standard logical file names are used in all programs as follows:

INFILE	Input of all processing parameters.
REPORT	Printed output generated by the programs.
ZINDX	The Index File.
ZDATA	The Data File currently in use.

In addition, other temporary work files are used in the Index Creation and Selection programs. These are discussed in the appropriate processing descriptions.

5.1.3. Physical Files

All physical files used on the Indexing System are defined using the MTS file name modifier Not Trim. This ensures that the records within the files are fixed length without the standard MTS option being applied (truncation of trailing blanks on MTS Line Files). The modifier is applied to the physical file name thus:

IXFILE@¬TRIM

5.1.4. PL/1 List I/O

All input to the processing programs (on INFILE) is carried out using PL/1 List I/O. This mode of input allows for entry of commands to the PL/1 program without any formatting statements. However, the following rules apply:

1. Character information is entered surrounded by single quotes, e.g. 'MINICOMPUTERS'.
2. Numeric information is entered normally, e.g. 20.
3. Each item of data entered to the programs must be separated from the preceding item by either a comma or at least one blank. In all the examples quoted, a blank is used as a data item separator.

5.1.5. Test Data File

In consideration of use of the General Purpose Inverted Indexing System, a test Data File is used. This is a file containing information on characteristics of Minicomputers. As used for testing, this file contains 94 records - a small file, but sufficient to demonstrate the workings of the system. The sections on system operation follow the steps involved in the creation and processing of a set of Inverted Indexes based on this test Data File.

5.2. Conversion of Basic Data to Tabular Form

The first requirement for creation of Inverted Indexes from a Data File is that the Data File be held in a tabular form. This means that each record must have the same number of fields, each of which is fixed length and using the allowable system data types. If the record is held in some other form, e.g. hierarchical - a header record followed by a number of associated data records, it must be converted to tabular form by a program specially written by the user. Even when the data is already held in tabular form (say on punched cards), a program might still be written to convert this data into various internal data types.

5.2. Conversion of Basic Data to Tabular Form contd

The MINICOMPUTER information file to be loaded was originally stored on punched cards. However, a program was written to convert it to a tabular form with internal data types such as character, half and full word binary, and full word floating point. The MTS Job Control to execute the conversion program (named -IPMTEST) was:

```
£CREATE MCFILE
£RUN -IPMTEST+*PL1LIB PAR=INFILE=*SOURCE* REPORT=*SINK* -
     ZDATA=MCFILE@¬TRIM
```

Input to the conversion program came for the card reader (*SOURCE*), printed output went to the printer (*SINK*), and the tabular Data File with internal data types was stored on an MTS Line File with no truncation of trailing blanks (MCFILE@¬TRIM).

5.3. Setup of the Blank Index File

Before any processing can be carried out on the Index File, it must first be set up. This involves the setting up of basic control information such as block size, number of blocks, size of index fields etc. as well as creation of an availability chain through all unused blocks. In its present form, the sole input to this program is the number of blocks required.

NOTE: In a system in which varying block sizes could be set up according to requirements, the block size for the Index File would be entered to this program.

It is first required to create the Index File with a nominal number of blocks - enough to hold the initial File and Field Name Directory entries. Therefore, it is defined that an Index File is to be created with 20 blocks called IXFILE. The input to the MTS system would be:

```
£CREATE IXFILE
£RUN IXUSTP+*PL1LIB PAR=INFILE=*SOURCE* REPORT=*SINK* -
     ZINDX=IXFILE@¬TRIM
20
£ENDFILE
```

Figure 5.1. shows the output listing from the above program.

5.4. Loading of File Descriptions

After creation of the blank Index File, the next step is to load the description of the Data File to be processed onto the File and Field Name Directories. A number of Data File descriptions may be entered in one File Description Addition run. The input format for each File

FIGURE 5.1: IXUSTP- SET UP A BLANK INDEX FILE

```
INDEX SETUP UTILITY
===================

NO. OF BLOCKS SPECIFIED =      20

BLOCK SIZE SPECIFIED    =     255

INDEX SETUP STARTED

INDEX SETUP SUCCESSFULLY COMPLETED

END OF RUN
```

5.4. Loading of File Descriptions contd

Description with associated Field Names and Characteristics is as follows:

file name no. of fields

followed by one card for each field containing:

field name field type field length

The file and field names may be up to a maximum of 20 characters in length. There are five allowable field types with associated field lengths:

Type No.	Description	Length (bytes)
1	Character String	1 - 255
2	Half Word Binary	2
3	Full Word Binary	4
4	Single Precision Floating Point	4
5	Double Precision Floating Point	8

Checking is carried on input data for validity according to the above rules.

It is required to load onto the Index File a description of the Data File on Minicomputers. Each record contains 12 fields of varying data types. The input to the MTS system would be:

```
£RUN IXPNEW+IXPFND+IXDOPN+IXDCLS+IXDCRE+IXDDST+IXDREA+IXDWRT+IXD-
MOV+IXDOSF+*PL1LIB PAR=INFILE=*SOURCE* REPORT=*SINK*
     ZINDX=IXFILE@¬TRIM
'MINICOMPUTERS' 12
'MANUFACTURER' 1 23
'MODEL' 1 18
'WORD LENGTH(BITS)' 2 2
'MINIMUM MEMORY(K)' 2 2
'MAXIMUM MEMORY(K)' 2 2
'CYCLE TIME(MICROSEC)' 4 4
'BASIC COST(£)' 3 4
'MEMORY COST BASIS(K)' 2 2
'FORTRAN' 1 1
'ALGOL' 1 1
'BASIC' 1 1
'COBOL' 1 1
£ENDFILE
```

Figure 5.2. shows the output listing from the above program.

FIGURE 5.2: IXPNEW - LOAD A FILE DESCRIPTION

```
ADD A NEW FILE DESCRIPTION
==========================

FILE NAME MINICOMPUTERS          ENTERED
NUMBER OF FIELDS =      12

***** SUB-FILE CREATED IN BLOCK       3

FIELD NAME MANUFACTURER          , NUMBERED       1 ENTERED
FIELD TYPE NO.  1 ENTERED, CHARACTER STRING LENGTH  23

FIELD NAME MODEL                 , NUMBERED       2 ENTERED
FIELD TYPE NO.  1 ENTERED, CHARACTER STRING LENGTH  18

FIELD NAME WORD LENGTH(BITS)     , NUMBERED       3 ENTERED
FIELD TYPE NO.  2 ENTERED, BINARY FIXED(15,0) LENGTH   2

FIELD NAME MINIMUM MEMORY(K)     , NUMBERED       4 ENTERED
FIELD TYPE NO.  2 ENTERED, BINARY FIXED(15,0) LENGTH   2

FIELD NAME MAXIMUM MEMORY(K)     , NUMBERED       5 ENTERED
FIELD TYPE NO.  2 ENTERED, BINARY FIXED(15,0) LENGTH   2

FIELD NAME CYCLE TIME(MICROSEC), NUMBERED         6 ENTERED
FIELD TYPE NO.  4 ENTERED, BINARY FLOAT(21) LENGTH   4

FIELD NAME BASIC COST(£)         , NUMBERED       7 ENTERED
FIELD TYPE NO.  3 ENTERED, BINARY FIXED(31,0) LENGTH   4

FIELD NAME MEMORY COST BASIS(K), NUMBERED         8 ENTERED
FIELD TYPE NO.  2 ENTERED, BINARY FIXED(15,0) LENGTH   2
***** OVER SUB-FILE PROCESSING ADDS BLOCK       4

FIELD NAME FORTRAN               , NUMBERED       9 ENTERED
FIELD TYPE NO.  1 ENTERED, CHARACTER STRING LENGTH   1

FIELD NAME ALGOL                 , NUMBERED      10 ENTERED
FIELD TYPE NO.  1 ENTERED, CHARACTER STRING LENGTH   1

FIELD NAME BASIC                 , NUMBERED      11 ENTERED
FIELD TYPE NO.  1 ENTERED, CHARACTER STRING LENGTH   1

FIELD NAME COBOL                 , NUMBERED      12 ENTERED
FIELD TYPE NO.  1 ENTERED, CHARACTER STRING LENGTH   1

END OF RUN
```

5.5. Estimation of Inverted Index Sizes

Before the required inverted indexes are created, it may be necessary to determine whether there is sufficient space available on the Index File. This may be done by invoking the 'EST' or Estimate option in the Index Creation program. The fields which are to be indexed are named and a calculation carried out based on the Data Area size, the Field Length and the number of records in the Data File. The assumption is made that each field value within the Data File is unique so that there are as many index entries as there are data records. Thus the maximum possible Inverted Index size is calculated. In actual creation of inverted indexes at a later stage, this estimated size is used to determine if processing can continue if space is available. It is required to estimate the index size for a number of fields to be indexed on the file containing Minicomputer specifications. The general format of the input command is:

file name option (in this case 'EST')
followed by a number of field names

The fields to be entered for estimation are Manufacturer, Cycle Time (Microsec), Basic Cost(£), FORTRAN, ALGOL, BASIC and COBOL.
The maximum possible index size in blocks for eachfield is calculated and a summary total displayed at the end of the run.
The input to the MTS system would be:

```
£RUN IXPIXC+IXPFND+IXDOPN+IXDCLS+IXDCRE+IXDDST+IXDREA+IXDWRT+IXD-
MOV+IXDOSF+*PL1LIB PAR=INFILE=*SOURCE* REPORT=*SINK*                -
     ZINDX=IXFILE@¬TRIM ZDATA=MCFILE@¬TRIM ZISRT=-A@¬TRIM            -
     ZOSRT=-B@¬TRIM
'MINICOMPUTERS' 'EST'
'MANUFACTURER' 'CYCLE TIME(MICROSEC)' 'BASIC COST(£)'
'FORTRAN' 'ALGOL' 'BASIC' 'COBOL'
£ENDFILE
```

Logical files ZISRT and ZOSRT are used in the Index Creation program when the index is actually being created to act as input and output files (temporary) for a sort of the field values and record addresses generated into ascending sequence.
Figure 5.3. shows part of the output listing from the above program.

5.6. Extension of Index File

Now that an estimate of the Index File size requirements have been obtained, it may be that currently there are insufficient blocks available to accomodate the maximum index sizes specified. It would then be necessary to extend the Index File to obtain extra available blocks. The

FIGURE 5.3: IXPIXC - ESTIMATION OF INDEX SIZES

```
CREATE AN INVERTED INDEX
========================

PROCESSING FILE MINICOMPUTERS

INDEX PROCESSING ON FIELD MANUFACTURER          REQUESTED

FILE MINICOMPUTERS          RECORD LENGTH =     61

FIELD MANUFACTURER           LENGTH =      23, START POSITION =       1

NUMBER OF INDEXING RECORDS =      94

MAXIMUM NO. OF BLOCKS REQUIRED =      14

ESTIMATE OPTION INVOKED - NO FURTHER PROCESSING

INDEX PROCESSING ON FIELD CYCLE TIME(MICROSEC) REQUESTED

FILE MINICOMPUTERS          RECORD LENGTH =     61

FIELD CYCLE TIME(MICROSEC) LENGTH =       4, START POSITION =      4

NUMBER OF INDEXING RECORDS =      94

MAXIMUM NO. OF BLOCKS REQUIRED =       6

ESTIMATE OPTION INVOKED - NO FURTHER PROCESSING

INDEX PROCESSING ON FIELD BASIC COST(£)         REQUESTED

FILE MINICOMPUTERS          RECORD LENGTH =     61

FIELD BASIC COST(£)          LENGTH =       4, START POSITION =      5

NUMBER OF INDEXING RECORDS =      94

MAXIMUM NO. OF BLOCKS REQUIRED =       6

ESTIMATE OPTION INVOKED - NO FURTHER PROCESSING

INDEX PROCESSING ON FIELD FORTRAN               REQUESTED

FILE MINICOMPUTERS          RECORD LENGTH =     61

FIELD FORTRAN                LENGTH =       1, START POSITION =      5

NUMBER OF INDEXING RECORDS =      94

MAXIMUM NO. OF BLOCKS REQUIRED =       5

ESTIMATE OPTION INVOKED - NO FURTHER PROCESSING

ETC....
```

5.6. Extension of Index File contd

Index File Extension Utility program chains through the currently available blocks and extends the availability chain to the added blocks. The input to the utility is simply the new number of blocks to which the Index File is to be extended (which must be greater than the previous number of blocks). Thus the input to increase the Index File to 60 blocks would be:

```
£RUN IXUSTX+*PL1LIB PAR=INFILE=*SOURCE* REPORT=*SINK* -
     ZINDX=IXFILE@¬TRIM
60
£ENDFILE
```

Figure 5.4. shows the output listing from the above program.

5.7. Creation of Inverted Indexes

If sufficient space is now available on the Index File, then inverted indexes for specified fields may be created. This is performed using the Index Creation program IXPIXC, which may be used to create only one inverted index at a time. This is opposed to the use of the Estimate option for this program which allows analysis of space requirements for multiple fields. The format of entry to create an inverted index on a specified field is:

```
file name   option   field name
```

The file and field names must be contained on the File and Field Name Directories. There are two available processing options for inverted index creation. In the course of creation of an inverted index on a specified field, it may be discovered that the field is already indexed. Two courses of action are available:

1. The existing index is not to be deleted. Therefore the proposed index creation is to be cancelled. For this option enter 'YES'.
2. The existing index may be deleted, the space occupied recovered, and the index creation may continue. Here, the processing option used is 'NO'.

Consider the example where an inverted index is to be created for the field Manufacturer on the Minicomputers Data File, deleting any previous index if existing. The MTS input would be:

```
£RUN IXPIXC+IXPFND+IXDOPN+IXDCLS+IXDCRE+IXDDST+IXDREA+IXDWRT+IXD-
MOV+IXDOSF+*PL1LIB PAR=INFILE=*SOURCE* REPORT=*SINK* -
     ZINDX=IXFILE@¬TRIM ZDATA=MCFILE@¬TRIM ZISRT=-A@¬TRIM -
     ZOSRT=-B@¬TRIM
'MINICOMPUTERS' 'NO' 'MANUFACTURER'
£ENDFILE
```

FIGURE 5.4: IXUSTX - EXTENSION OF INDEX FILE

```
INDEX EXTENSION UTILITY
=======================

NO. OF BLOCKS SPECIFIED =      60

BLOCK SIZE SPECIFIED =     255

INDEX EXTENSION STARTED

INDEX EXTENSION SUCCESSFULLY COMPLETED

END OF RUN
```

5.7. Creation of Inverted Indexes contd

Figure 5.5. shows the output listing from the above program. Note the logging of CPU time used in the various portions of the program.

5.8. Processing of "Complex" Queries

Processing of queries against the appropriate inverted indexes is carried out by the Index Selection program IXPIXS. It accepts "Complex" queries in the form of a file name followed by a number of Selection Conditions delimited by 'AND', 'OR' and 'END' logical conditions. A Selection Condition is specified thus:

field name operator field value

The available operators are Equal (EQ), Not Equal (NEQ), Greater Than (GT), Greater Than or Equal (GTE), Less Than (LT), and Less Than or Equal (LTE).

For example, a Selection Condition to retrieve all records on Minicomputers manufactured by Data General would be specified thus:

'MANUFACTURER' 'EQ' 'DATA GENERAL'

A complex query might be specified thus:

SC1 AND SC2 OR SC3 AND SC4 END

where SCn refers to a selection condition similar to that specified above. When evaluating the expression, the selection conditions linked by 'AND' logical operators are first evaluated (common records in two or more searches); followed by 'OR' processing of the resulting 'AND' lists (each unique record occurrence eliminating duplicates). The end of all selection conditions being entered is signified by the entry of the 'END' logical operator.

At the present stage of development, the Index Selection Program returns:

1. The number of records in the Data File which satisfy the selection conditions entered.
2. A list of the record addresses (sequence number in the Data File) which are sorted in ascending sequence.

In further enhancements, the program might well be enlarged to provide the above two options as well as abstracting from the Data File into a temporary file those records which satisfy the query. The user might then choose to further process these records in various ways.

A typical "Complex" query on the Minicomputers Data File might be expressed as follows:

What Minicomputers manufactured by Data General or Digital Equipment have a basic cost of less than £3000?

As explained in Chapter 4, the MTS EMPTY command is used within the Index Selection program to purge the contents of work files during

FIGURE 5.5: IXPIXC - CREATE AN INVERTED INDEX

```
CREATE AN INVERTED INDEX
========================

PROCESSING FILE MINICOMPUTERS

INDEX PROCESSING ON FIELD MANUFACTURER          REQUESTED

FILE MINICOMPUTERS          RECORD LENGTH =     61

FIELD MANUFACTURER          LENGTH =     23, START POSITION =      1

NUMBER OF INDEXING RECORDS =      94

MAXIMUM NO. OF BLOCKS REQUIRED =      14

***** SUB-FILE CREATED ON BLOCK       5

***** SUB-FILE CREATED ON BLOCK       6
***** OVER SUB-FILE PROCESSING ADDS BLOCK       7
***** OVER SUB-FILE PROCESSING ADDS BLOCK       8
***** OVER SUB-FILE PROCESSING ADDS BLOCK       9
***** OVER SUB-FILE PROCESSING ADDS BLOCK      10

INDEX CREATION STATISTICS
-------------------------

NO. OF FIELD VALUES ENTERED =       30

NO. OF INDEX CONTROL ENTRIES =       5

EDIT CPU TIME(SECONDS) =          0.063

ABSTRACT CPU TIME(SECONDS) =      0.526

SORT CPU TIME(SECONDS) =          0.585

CREATE CPU TIME(SECONDS) =        0.481

TOTAL CPU TIME(SECONDS) =         1.655

END OF RUN
```

5.8. Processing of "Complex" Queries contd

a selection run. Unless precautions are taken, MTS commands are displayed in the selection listing. This is avoided by writing the Selection program output to a temporary file -SLIST and listing it following completion of the program. Thus, the MTS input to answer the above query would be as follows:

```
£RUN IXPIXS+IXPEDT+IXPSEL+IXPCMP+IXPFND+IXDOPN+IXDCLS+IXDREA+IXD-
MOV+IXDOSF+*PL1LIB PAR=INFILE=*SOURCE* REPORT=-SLIST            -
     ZINDX=IXFILE@¬TRIM ZWADD=-W@¬TRIM ZAADD=-A@¬TRIM            -
     ZOADD=-O@¬TRIM
'MINICOMPUTERS'
'MANUFACTURER' 'EQ' 'DATA GENERAL' 'AND'
'BASIC COST(£)' 'LT' '3000' 'OR'
'MANUFACTURER' 'EQ' 'DIGITAL EQUIPMENT' 'AND'
'BASIC COST(£)' 'LT' '3000' 'END'
£ENDFILE
£LIST -SLIST
£EMPTY -SLIST
```

The EMPTY command purges the contents of the temporary file -SLIST in case another invocation of the Selection program is to be carried out immediately following the above query.

The logical files ZWADD, ZAADD and ZOADD are used as work files for 'AND' processing, 'OR' processing, and sorting within the Index Selection program.

Figure 5.6. shows the output listing from the above program. The search finds 9 records on the Data File which satisfy the Selection Conditions and displays their addresses. Note the logging of CPU time used in various portions of the program.

5.9. Deletion of Inverted Indexes

If an inverted index for a particular field name is no longer required, it may be deleted and the space recovered for system availability. The input commands to the Index Deletion Program IXPIXD comprises two items:

```
file name   field name
```

Thus, if the inverted index for the field COBOL in the Minicomputers file is to be deleted, the MTS input would be:

```
£RUN IXPIXD+IXPFND+IXDOPN+IXDCLS+IXDDST+IXDREA+IXDWRT+IXDMOV+IXD-
OSF+*PL1LIB PAR=INFILE=*SOURCE* REPORT=*SINK* ZINDX=IXFILE@¬TRIM
'MINICOMPUTERS' 'COBOL'
£ENDFILE
```

Figure 5.7. shows the output listing from the above program.

FIGURE 5.6: IXPIXS - PROCESSING OF "COMPLEX" QUERIES

```
SELECT RECORDS USING INVERTED INDEX
===================================

INPUT EDIT PHASE
----------------

PROCESSING FILE MINICOMPUTERS

***** SELECT CONDITION  1
SEARCH INDEX ON (MANUFACTURER          ) EQ (DATA GENERAL)

***** LOGICAL DELIMITER AND ENTERED

***** SELECT CONDITION  2
SEARCH INDEX ON (BASIC COST(£)         ) LT (3000)

***** LOGICAL DELIMITER OR ENTERED

***** SELECT CONDITION  3
SEARCH INDEX ON (MANUFACTURER          ) EQ (DIGITAL EQUIPMENT)

***** LOGICAL DELIMITER AND ENTERED

***** SELECT CONDITION  4
SEARCH INDEX ON (BASIC COST(£)         ) LT (3000)

***** LOGICAL DELIMITER END ENTERED

SELECT PROCESSING PHASE
-----------------------

ADDRESS DISPLAY PHASE
---------------------

NO. OF RECORD ADDRESSES =       9 - LIST FOLLOWS
      14       15       16       17       18       26       29       30
      31

INDEX SELECTION STATISTICS
--------------------------

NO. OF COMPARISONS =       127

EDIT CPU TIME(SECONDS) =         0.186

AND CPU TIME(SECONDS) =          1.584

OR CPU TIME(SECONDS) =           0.337

LIST CPU TIME(SECONDS) =         0.087

TOTAL CPU TIME(SECONDS) =        2.194

END OF RUN
```

FIGURE 5.7: IXPIXD - DELETE AN INVERTED INDEX

```
DELETE INDEX FROM INVERTED FILE
===============================

DELETE INDEX ON MINICOMPUTERS          (COBOL )
***** SUB-FILE BLOCK      31 DELETED
***** SUB-FILE BLOCK      32 DELETED
***** SUB-FILE BLOCK      30 DELETED

END OF RUN
```

5.10. Deletion of File Descriptions

If a File Description with associated Field Names, Descriptions and Inverted Indexes is no longer required, it may be deleted and the space occupied recovered for system availability. The input specifying a deletion command to the File Description Deletion program IXPDEL is simply the file name to be deleted.

Thus, if the entire information on Minicomputers is to be scratched from the Index File, the MTS input would be:

```
&RUN IXPDEL+IXPFND+IXDOPN+IXDCLS+IXDDST+IXDREA+IXDWRT+IXDMOV+IXD-
OSF+*PL1LIB PAR=INFILE=*SOURCE* REPORT=*SINK* ZINDX=IXFILE@¬TRIM
'MINICOMPUTERS'
&ENDFILE
```

Figure 5.8. shows the output listing from the above program. Note that the field name COBOL is defined as not indexed, the inverted index for COBOL having been deleted in the run described in Section 5.9. above.

5.11. Report Utilities

In addition to the Index Setup and Extension Utilities previously described, three other utility programs are provided within the General Purpose Inverted Indexing System. They are:

1. The Index File Status Report.
2. The Directory Report.
3. The Inverted Index Report.

Each of these utilities is described below.

5.11.1. Index File Status Report

The current status of the Index File is displayed, including the block and data area sizes, the total number of blocks in the Index File and the total number of available blocks. The MTS Job Control to invoke the Index File Status Report is:

```
&RUN IXUSTR+IXDOPN+IXDCLS+*PL1LIB PAR=REPORT=*SINK*
    ZINDX=IXFILE@¬TRIM
```

Figure 5.9. shows the output listing from the above program.

5.11.2. Directory Report

A complete listing of the entries on the File and associated Field Name Directories may be displayed using the Directory Report Utility. The Directory Control Information (number of file name and deleted entries) is first displayed, followed by the File Descriptions, together with Field Names, Type, Length and Whether Indexed. The MTS Job Control to

FIGURE 5.8: IXPDEL - DELETE A FILE DESCRIPTION

```
DELETE A FILE DESCRIPTION
=========================

FILE NAME MINICOMPUTERS          TO BE DELETED

DELETE FIELD NAME MANUFACTURER          , - INDEXED
***** SUB-FILE BLOCK      6 DELETED
***** SUB-FILE BLOCK      7 DELETED
***** SUB-FILE BLOCK      8 DELETED
***** SUB-FILE BLOCK      9 DELETED
***** SUB-FILE BLOCK     10 DELETED
***** SUB-FILE BLOCK      5 DELETED

DELETE FIELD NAME MODEL                 , - NOT INDEXED

DELETE FIELD NAME WORD LENGTH(BITS)     , - NOT INDEXED

DELETE FIELD NAME MINIMUM MEMORY(K)     , - NOT INDEXED

DELETE FIELD NAME MAXIMUM MEMORY(K)     , - NOT INDEXED

DELETE FIELD NAME CYCLE TIME(MICROSEC), - INDEXED
***** SUB-FILE BLOCK     12 DELETED
***** SUB-FILE BLOCK     13 DELETED
***** SUB-FILE BLOCK     14 DELETED
***** SUB-FILE BLOCK     11 DELETED

DELETE FIELD NAME BASIC COST($)         , - INDEXED
***** SUB-FILE BLOCK     16 DELETED
***** SUB-FILE BLOCK     17 DELETED
***** SUB-FILE BLOCK     18 DELETED
***** SUB-FILE BLOCK     19 DELETED
***** SUB-FILE BLOCK     20 DELETED
***** SUB-FILE BLOCK     15 DELETED

DELETE FIELD NAME MEMORY COST BASIS(K), - NOT INDEXED

DELETE FIELD NAME FORTRAN               , - INDEXED
***** SUB-FILE BLOCK     22 DELETED
***** SUB-FILE BLOCK     23 DELETED
***** SUB-FILE BLOCK     21 DELETED

DELETE FIELD NAME ALGOL                 , - INDEXED
***** SUB-FILE BLOCK     25 DELETED
***** SUB-FILE BLOCK     26 DELETED
***** SUB-FILE BLOCK     24 DELETED

DELETE FIELD NAME BASIC                 , - INDEXED
***** SUB-FILE BLOCK     28 DELETED
***** SUB-FILE BLOCK     29 DELETED
***** SUB-FILE BLOCK     27 DELETED

DELETE FIELD NAME COBOL                 , - NOT INDEXED
***** SUB-FILE BLOCK      3 DELETED
***** SUB-FILE BLOCK      4 DELETED

END OF RUN
```

FIGURE 5.9: IXUSTR - INDEX FILE STATUS REPORT

```
INDEX FILE STATUS REPORT
========================

INDEX CONTROL INFORMATION
-------------------------

BLOCK SIZE (BYTES) =     255

DATA AREA SIZE =     251

NUMBER OF BLOCKS IN INDEX FILE =          60

NUMBER OF AVAILABLE BLOCKS =          28

END OF RUN
```

5.11.2. Directory Report contd

invoke the Directory Report is as follows:

```
&RUN IXUDRP+IXDOPN+IXDCLS+IXDREA+IXDMOV+IXDOSF+*PL1LIB -
    PAR=REPORT=*SINK* ZINDX=IXFILE@¬TRIM
```

Figure 5.10. shows the output listing from the above program.

5.11.3. Inverted Index Report

A specified inverted index may be displayed in full or in part by invocation of the Inverted Index Report Utility. This program is primarily a debugging aid developed by the author in proving the General Purpose Inverted Indexing System. The information that may be displayed includes the contents of the Index Control and Value and Address Sub-Files. The input format is:

```
file name   field name   option
```

There are two options available:

1. If the entire contents of the Inverted Index are to be displayed, enter the option 'ALL'.
2. If the Index Control information only is to be displayed, enter the option 'IXC'.

The MTS Job Control to display the Inverted Index for the field FORTRAN in the Minicomputers file would be:

```
&RUN IXUIXL+IXPFND+IXDOPN+IXDCLS+IXDREA+IXDMOV+IXDOSF+*PL1LIB -
    PAR=INFILE=*SOURCE* REPORT=*SINK* ZINDX=IXFILE@¬TRIM
'MINICOMPUTERS' 'FORTRAN' 'ALL'
&ENDFILE
```

Here the option to display the entire Inverted Index has been selected. Figure 5.11. shows the output listing from the above program.

FIGURE 5.10: IXUDRP - DIRECTORY REPORT

```
DISPLAY OF DATA DESCRIPTION DIRECTORY
=====================================

DIRECTORY CONTROL INFORMATION
-----------------------------

NUMBER OF FILE NAME ENTRIES =       1

NUMBER OF DELETED ENTRIES =         0

FILE NAME - MINICOMPUTERS
---------------------------------

FILE NUMBER        1, NUMBER OF FIELDS =      12

FIELD NAME MANUFACTURER          , NUMBER =        1
FIELD TYPE CHARACTER STRING, LENGTH =       23
FIELD STATUS INDEXED, TIMES ACCESSED =        0

FIELD NAME MODEL                 , NUMBER =        2
FIELD TYPE CHARACTER STRING, LENGTH =       18
FIELD STATUS NOT INDEXED, TIMES ACCESSED =        0

FIELD NAME WORD LENGTH(BITS)     , NUMBER =        3
FIELD TYPE BINARY FIXED(15,0), LENGTH =        2
FIELD STATUS NOT INDEXED, TIMES ACCESSED =        0

FIELD NAME MINIMUM MEMORY(K)     , NUMBER =        4
FIELD TYPE BINARY FIXED(15,0), LENGTH =        2
FIELD STATUS NOT INDEXED, TIMES ACCESSED =        0

FIELD NAME MAXIMUM MEMORY(K)     , NUMBER =        5
FIELD TYPE BINARY FIXED(15,0), LENGTH =        2
FIELD STATUS NOT INDEXED, TIMES ACCESSED =        0

FIELD NAME CYCLE TIME(MICROSEC), NUMBER =        6
FIELD TYPE BINARY FLOAT(21), LENGTH =        4
FIELD STATUS INDEXED, TIMES ACCESSED =        0

FIELD NAME BASIC COST(£)         , NUMBER =        7
FIELD TYPE BINARY FIXED(31,0), LENGTH =        4
FIELD STATUS INDEXED, TIMES ACCESSED =        0

FIELD NAME MEMORY COST BASIS(K), NUMBER =        8
FIELD TYPE BINARY FIXED(15,0), LENGTH =        2
FIELD STATUS NOT INDEXED, TIMES ACCESSED =        0

FIELD NAME FORTRAN               , NUMBER =        9
FIELD TYPE CHARACTER STRING, LENGTH =        1
FIELD STATUS INDEXED, TIMES ACCESSED =        0

FIELD NAME ALGOL                 , NUMBER =       10
FIELD TYPE CHARACTER STRING, LENGTH =        1
FIELD STATUS INDEXED, TIMES ACCESSED =        0

FIELD NAME BASIC                 , NUMBER =       11
FIELD TYPE CHARACTER STRING, LENGTH =        1
FIELD STATUS INDEXED, TIMES ACCESSED =        0

FIELD NAME COBOL                 , NUMBER =       12
FIELD TYPE CHARACTER STRING, LENGTH =        1
FIELD STATUS INDEXED, TIMES ACCESSED =        0

END OF RUN
```

```
DISPLAY AN INVERTED INDEX
=========================

DISPLAY INDEX ON MINICOMPUTERS          (FORTRAN          )

FIELD TYPE CHARACTER STRING, LENGTH    1

INDEX CONTROL BLOCK INFORMATION
-------------------------------

NO. OF VALUE AND ADDRESS BLOCKS =    1

FIELD VALUE Y
IS HIGHEST KEY IN BLOCK      22, OFFSET      58

VALUE AND ADDRESS BLOCK INFORMATION
-----------------------------------

FIELD VALUE N
NO. OF RECORD ADDRESSES =       13 - LIST FOLLOWS
        1        6       12       52       53       54       61       63       64       73
       79       82       94

FIELD VALUE Y
NO. OF RECORD ADDRESSES =       81 - LIST FOLLOWS
        2        3        4        5        7        8        9       10       11       13
       14       15       16       17       18       19       20       21       22       23
       24       25       26       27       28       29       30       31       32       33
       34       35       36       37       38       39       40       41       42       43
       44       45       46       47       48       49       50       51       55       56
       57       58       59       60       62       65       66       67       68       69
       70       71       72       74       75       76       77       78       80       81
       83       84       85       86       87       88       89       90       91       92
       93

END OF RUN
```

FIGURE 5.11: IXUIXL - INVERTED INDEX REPORT

6.1. Summary

The General Purpose Inverted Indexing System has been implemented and shown to be able to logically process small files, as shown by the examples of system operation on a file of Minicomputer information containing 94 records (see Chapter 5). However, there is also a need to determine the system performance when it is asked to process larger quantities of data using more realistic queries.

It was therefore decided to examine two aspects of system performance essential to the viability of an inverted file organization:

1. Creation of Inverted Indexes.
2. Selection using "Complex" Queries.

To provide data for testing, two sets of research data were used:

1. An abstract from the Durham High Energy Physics databank.
2. Data obtained in the Northern Archaeological Survey.

This chapter examines in detail how the system performance might be measured, and how experiments were carried out on the research data.

6.2. Methods of Performance Measurement

In consideration of how the system performance might be measured, it was found that two types of information needed to be gathered:

1. Display of numbers for a given operation. These numbers might either be abstracted from the Directories or accumulated during the course of a run.
2. Determination of CPU time used in various parts of a program. The method for this was defined in Chapter 4 (Section 4.5.3).

The numbers and CPU time gathered and displayed are discussed below in relation to the particular programs.

6.2.1. Creation of Inverted Indexes

A number of items of information are collected and displayed when an inverted index is created for a specified field:

1. File and Field Names.
   These are obtained from the input request for index creation. The field type may be obtained from the display of the Directories for that Data File.
2. Record Length.
   The total length of the Data Record is displayed.
3. Field Length and Start Position.
   The length of the field to be indexed together with its starting position within the Data Record.

6.2.1. Creation of Inverted Indexes contd

4. Number of Indexing Records.
   The number of records in the Data File (and hence the number of record addresses to be stored in the inverted index when created) is counted and displayed.
5. Maximum Number of Blocks Required.
   During the execution of the creation program, an estimate is made of the maximum inverted index size in terms of the number of blocks required. This information, which represents the worst possible case where each field value is unique, is displayed.
6. Actual Number of Blocks Used.
   When the inverted index is actually created, the blocks on the Index File allocated for this purpose are noted and their numbers displayed. By counting these, the actual number of blocks used is determined.
7. Number of Field Values Entered.
   The number of unique field values found on the field being indexed is counted and displayed. This figure, together with the number of indexing records, can provide an average list length for each inverted index entry.
8. Number of Index Control Entries.
   The number of field value/pointer entries in the top level of inverted index (the Index Control Sub-File) are counted and displayed.
9. Edit CPU Time(Seconds).
   The amount of CPU time used in editing the input data against the Directories is logged and displayed.
10. Abstract CPU Time(Seconds).
    This CPU time measurement shows the time taken to access the Data File and abstract the field value and associated record address into a temporary file.
11. Sort CPU Time(Seconds).
    The abstracted information is then sorted into field value and record address sequence. The CPU time used is displayed.
12. Create CPU Time(Seconds).
    Here the CPU time taken to create the inverted indexes (Index Control and Value and Address Sub-Files) is logged and displayed.
13. Total CPU Time(Seconds).
    The above CPU times are totalled and displayed.

6.2.2. Processing of "Complex" Queries

The following items of information are gathered and displayed when processing a "Complex" query:

1. File Name.
   The File Name as entered as a command and edited by the system for validity is displayed.
2. Selection Conditions.
   Each Selection Condition entered is numbered and displayed. From the field names specified, the appropriate information may be obtained - type and length from the Directories; and number of unique field values and average list lengths from the index creation listing.
3. Logical Delimiters.
   Three types of logical delimiters are permitted: AND, OR and END. These are displayed in their required positions to separate (AND and OR) or terminate the Selection Conditions (END).
4. Number of Record Addresses.
   The number of record addresses satisfying the input query is displayed. If no records are found, an appropriate message is printed.
5. List of Record Addresses.
   The record addresses satisfying the query conditions is displayed. This display is primarily for logical testing purposes.
6. Number of Comparisons.
   In the course of processing Selection Conditions, comparisons of the search field value are made against various portions of the inverted index. The number of these comparisons is accumulated and displayed.
7. Edit CPU Time(Seconds).
   The amount of CPU time used in editing the input against the Directories and creating the control array with selection instructions is logged and displayed.
8. AND CPU Time(Seconds).
   The time taken to logically AND the various lists together is recorded and displayed. This may include sorting.
9. OR CPU Time(Seconds).
   The CPU time taken to logically OR the resultant AND lists together is logged and displayed.

6.2.2. Processing of "Complex" Queries contd

10. List CPU Time(Seconds).
    The final list of record addresses is listed and the time displayed.
11. Total CPU Time(Seconds).
    The above CPU times are totalled and displayed.

6.3. High Energy Physics Databank

The Durham High Energy Physics Databank is used by the Physics Department of Durham University to store and access data concerned with high energy physics experiments. It is stored in card image form on a private disk on the NUMAC system in the form of a number of files. The current size of the databank is 20 files, each containing approximately 2000 cards. Thus the current file size is approaching 40000 records.
In use of this information for performance testing on the General Purpose Inverted Indexing System, a sub-set of the first 10000 records was taken. The reasons for not using the entire databank were twofold:

1. The maximum allowable Line File size on the MTS system is 255 pages. At 4096 bytes per page, this represents a maximum size of approximately 1 million bytes available for storage of control information and data. While this problem could be circumvented by concatenation of a number of Line Files, performance testing using this maximum as an upper limit was selected because of the next consideration: CPU time requirements.
2. Preliminary estimates of time to create inverted indexes (which were borne out later by experiment) suggested that to create the required inverted indexes for the entire databank would take approximately 1¼ hours of CPU time on the NUMAC System/360 Model 67. As this would be only one of a number of runs, it was determined that experiments using a smaller number of records would be more economical (provided that they gave an adequate measure of system performance).

The decision to use a maximum of 10000 records in the High Energy Physics Databank was therefore taken.
Before considering the creation of the Data File, inverted index creation and processing, it is probably worth stating (in the view of the author) that there is no logical reason why the entire databank should not be processed using the General Purpose Inverted Indexing System. A number of recommendations for modification and improvement

6.3. High Energy Physics Databank contd

of the system are presented in Chapter 7.

6.3.1. Conversion to Tabular Form

One problem immediately emerged in the use of the High Energy Physics Databank for performance testing. This is that the data (in card image form) is organized in a hierarchical fashion, while the General Purpose Inverted Indexing System requires data to be presented in a tabular form. A description of the data format follows.

The data is divided into 'experiments', each of which has a Master Card describing the reference, process and type, and units of measurement. No conversion has been carried out on the data, and values presented in graphical form have been measured by ruler, and the original values calculated.

After the Master Card comes the Decay Card which specifies the way in which the final state resonances were identified. A Comment Card may appear after this. Neither the Decay or Comment Cards necessarily occur in an experiment.

At present the Decay and Comment Cards are read as 80 byte alphanumeric lines, and are not interpreted further.

These cards are then followed by a number of Data Cards relating to the experiment. The number of Data Cards may be anything from zero (where an experiment has been obsoleted) to hundreds of readings. Some of the information on the Data Card duplicates that held on the Master Card for that experiment.

To construct a tabular Data File that could be used to create inverted indexes, it was therefore necessary to write a program to convert from hierarchical to tabular form. In creation of the tabular Data File, it was also decided for reasons of space to exclude the data contained on the Decay and Comment Cards (which was being stored for information rather than as a basis for retrieval).

The output from the conversion program was a tabular Data File named HIGH ENERGY PHYSICS containing the following 13 fields:

1.	EXPERIMENT NO.	Fullword Binary Length 4.
2.	STATUS SYMBOL	Character String Length 1.
3.	REFERENCE	Character String Length 29.
4.	PROCESS CODE	Character String Length 13.
5.	INFORMATION CODE	Character String Length 10.
6.	ENERGY CODE	Character String Length 10.
7.	ANGLE CODE	Character String Length 10.
8.	ENERGY	Floating Point Length 4.

6.3.1. Conversion to Tabular Form contd

9.	ANGLE	Floating Point Length 4.
10.	OBSERVABLE	Floating Point Length 4.
11.	ERROR	Floating Point Length 4.
12.	POINT NO.	Halfword Binary Length 2.
13.	WIDTH	Floating Point Length 4.

Thus a record of length 99 bytes was constructed. Fields 1 - 7 are taken from the Master Card, and thus will be duplicated on the data records on which the Data Card information is also loaded (Fields 8 - 13).

By running the conversion program on the original Line Files in the databank, a number of tabular data files were created which could be concatenated to provide input to the Index Creation program of various sizes: 2000, 4000, 6000, 8000 and 10000 records (approximately).

6.3.2. Inverted Index Creation

To be able to accommodate the type of queries initially expected on the High Energy Physics databank, it was decided that inverted indexes would need to be created on the following seven fields:

Field No.	Description	Type	Length
1	EXPERIMENT NO.	Fullword Binary	4
4	PROCESS CODE	Character String	13
5	INFORMATION CODE	Character String	10
6	ENERGY CODE	Character String	10
7	ANGLE CODE	Character String	10
8	ENERGY	Floating Point	4
9	ANGLE	Floating Point	4

To test the effects of Data File size on Index Creation performance, the creation runs were to be run using differing numbers of records. As described above, tabular Data Files were created. The number of records used for each run were 1980, 4077, 6163, 7941 and 9635 respectively. The procedure prior to the runs was as follows:

1. Set up the blank Index File with an initial allocation of 20 blocks.
2. Load the description of the Data File onto the File and associated Field Name Directories.
3. Run an estimate of the total blocks required for the number of records to be processed.
4. Extend the Index File to provide enough space to accomodate the number of blocks estimated above.

The Data File with the requisite number of records was then accessed for

6.3.2. Inverted Index Creation contd

each field in turn in an Index Creation run.
Results obtained from the set of runs are shown in Figure 6.1. grouped by field. Some anomalous results were obtained during the test runs (denoted by an asterisk). These figures were checked by later reruns, but appeared to give substantially the same results.
The results obtained from the Index Creation runs are analysed in Chapter 7.

6.3.3. Processing of Queries

In discussion with Dr. Cooper of the Physics Department, seven queries were defined. The first four were variations on the same type of query with the expected number of records (in qualitative terms e.g. none) to be found from each query; while the last three defined standard type queries on the databank. The format of the queries was as follows:

1. PROCESS CODE PI+ P=PI+ P AND INFORMATION CODE D
   Amount of records satisfying query: large.
   The query however cannot be answered with just 2 Selection Conditions as there are 6 Information Codes classified under D: D(MB/G2), D(MUB/G2), D(MB/ST), D(MUB/ST), D(NB/G2) and DMUB*SMT2. Using the rules defined for specification of complex queries, the query may be defined using 12 Selection Conditions as follows (Note: the above Process Code is stored as PI+ P):

   'HIGH ENERGY PHYSICS'
   'PROCESS CODE' 'EQ' ' PI+ P' 'AND'
   'INFORMATION CODE' 'EQ' 'D(MB/G2)' 'OR'
   'PROCESS CODE' 'EQ' ' PI+ P' 'AND'
   'INFORMATION CODE' 'EQ' 'D(MUB/G2)' 'OR'
   'PROCESS CODE' 'EQ' ' PI+ P' 'AND'
   'INFORMATION CODE' 'EQ' 'D(MB/ST)' 'OR'
   'PROCESS CODE' 'EQ' ' PI+ P' 'AND'
   'INFORMATION CODE' 'EQ' 'D(MUB/ST)' 'OR'
   'PROCESS CODE' 'EQ' ' PI+ P' 'AND'
   'INFORMATION CODE' 'EQ' 'D(NB/ST)' 'OR'
   'PROCESS CODE' 'EQ' ' PI+ P' 'AND'
   'INFORMATION CODE' 'EQ' 'DMUB*SMT2' 'END'

2. PROCESS CODE PI+ P=PI+ P AND INFORMATION CODE DU
   Amount of records satisfying query: none.
   The query is defined with 12 Selection Conditions as above in 1. using 6 Information Codes classified under DU: DU(MB/G2),

FIELD			NO. OF INDEXING RECORDS	EST. NO. OF BLOCKS	ACTUAL NO. OF BLOCKS	NO. OF CONTROL ENTRIES	NO. OF UNIQUE KEYS	CPU TIME USED (SECONDS)				
NO.	TYPE	LENGTH						EDIT	ABST.	SORT	CREATE	TOTAL
1	3	4	1980	99	34	18	39	0.062	7.363	5.954	5.827	19.206
			4077	203	71	43	106	0.060	15.220	12.476	12.625	40.381
			6163	307	108	70	189	0.060	23.451	19.307	19.468	62.287
			7941	396	139	86	242	0.061	30.737	25.113	24.955	80.866
			9635	480	167	97	266	0.061	36.999	30.503	30.259	97.822
4	1	13	1980	179	34	9	11	0.060	8.282	6.852	5.756	20.950
			4077	368	70	22	36	0.063	43.579*	42.664*	12.056	98.362
			6163	556	105	33	46	0.060	27.744	24.112	18.342	70.258
			7941	716	134	36	54	0.062	36.973	32.667	23.578	93.280
			9635	869	161	39	54	0.061	47.975	41.400	28.384	117.820
5	1	10	1980	152	34	7	11	0.060	9.249	7.940	5.728	22.977
			4077	312	67	13	15	0.061	21.783	19.604	11.574	53.022
			6163	471	102	16	24	0.061	36.754	33.237	17.773	87.825
			7941	607	132	19	27	0.061	52.999	48.350	22.907	124.317
			9635	736	158	18	27	0.061	132.725*	89.253*	27.947	249.986
6	1	10	1980	152	33	1	1	0.061	7.033	5.666	5.417	18.177
			4077	312	67	1	1	0.061	14.316	11.730	11.085	37.192
			6163	471	100	1	1	0.061	21.592	18.080	16.938	56.671
			7941	607	128	1	1	0.062	28.251	23.405	22.008	73.726
			9635	736	155	1	1	0.063	35.765	28.942	27.063	91.833

*ANOMALOUS RESULTS

FIGURE 6.1: INDEX CREATION - HIGH ENERGY PHYSICS

FIELD			NO. OF INDEXING RECORDS	EST. NO. OF BLOCKS	ACTUAL NO. OF BLOCKS	NO. OF CONTROL ENTRIES	NO. OF UNIQUE KEYS	CPU TIME USED(SECONDS)				
NO.	TYPE	LENGTH						EDIT	ABST.	SORT	CREATE	TOTAL
7	1	10	1980	152	33	4	5	0.060	7.020	5.711	5.580	18.371
			4077	312	67	4	5	0.060	14.249	11.829	11.248	37.386
			6163	471	100	5	6	0.061	21.656	18.273	17.235	57.225
			7941	607	128	6	7	0.062	27.947	23.479	22.457	73.945
			9635	736	155	6	7	0.060	34.149	28.663	26.679	89.551
8	4	4	1980	99	37	35	105	0.061	9.653	8.601	6.909	25.224
			4077	203	75	67	205	0.060	22.508	20.732	13.606	56.906
			6163	307	117	100	445	0.060	39.819	37.385	21.339	98.603
			7941	396	148	121	486	0.062	48.866	45.309	26.846	121.083
			9635	480	176	147	500	0.061	76.494*	73.026*	32.915	182.496
9	4	4	1980	99	63	60	886	0.062	6.960	5.995	9.575	22.592
			4077	203	117	111	1461	0.060	13.907	12.288	18.893	45.148
			6163	307	169	155	1980	0.061	21.237	19.055	27.592	67.945
			7941	396	209	192	2314	0.060	26.867	23.906	35.361	86.194
			9635	480	245	224	2557	0.061	33.465	30.530	41.712	105.768

*ANOMALOUS RESULTS

FIGURE 6.1 CONTD: INDEX CREATION - HIGH ENERGY PHYSICS

6.3.3. Processing of Queries contd

DU(MUB/G2), DU(MB/ST), DU(MUB/ST), DU(NB/G2) and DU(MB/ST)L.

3. PROCESS CODE PI+ P=P PI+ AND INFORMATION CODE D
   Amount of records satisfying query: none.
   The query is defined with 12 Selection Conditions as in Query 1. using Information Codes classified under D and the above Process Code stored as P PI+.
4. PROCESS CODE PI+ P=P PI+ AND INFORMATION CODE DU
   Amount of records satisfying query: some.
   The query is defined with 12 Selection Conditions as in Query 1. using Information Codes classified under DU and the above Process Code stored as P PI+.

The above four queries therefore each have 12 Selection Conditions, comprising 6 ANDs and 5 ORs. Each Selection Condition is an Equal Compare.

5. ANGLE CODE CTHETA AND ANGLE > 0.5
   The Angle Code CTHETA is stored as CTH. The query is entered to the system thus:
   'ANGLE CODE' 'EQ' 'CTH' 'AND'
   'ANGLE' 'GT' '0.5' 'END'
6. ANGLE CODE T(GEV) AND ANGLE < 0.2
   This query is defined as in 5 above.
7. ENERGY CODE P(GEV) AND ENERGY > 5.0
   This query also is defined as in 5 above.

Therefore, the above queries each have 2 Selection Conditions containing 1 AND. Each query contains an Equal Compare and a Range Search (Greater Than or Less Than).
Results obtained from the set of query runs are shown in Figure 6.2. These results are analysed in Chapter 7.

6.4. Archaeological Data

The location, period, type and other information on archaeological sites is stored by the Archaeological Department on a MTS Sequential File and accessed to answer queries. The current file size is 3280 records at 18 bytes per record, the records being blocked with a factor of 400.

6.4.1. Inverted Index Creation

The archaeological Data File described above was stored in a tabular form, but a small conversion program was required to present it in a form for use in the Index Creation program comprising some 24 PL/1

QUERY NO.	NO. OF SELECT CONDS.	NO. OF ANDs	NO. OF ORs	NO. OF RECORDS IN FILE	NO. OF COMPS.	NO. OF RECORDS FOUND	CPU TIME USED(SECONDS)				
							EDIT	AND	OR	LIST	TOTAL
1	12	6	5	1980	48	0	0.275	1.307	0.0	0.0	1.587
				4077	101	305	0.276	11.910	0.669	0.492	13.347
				6163	115	499	0.280	15.583	0.810	0.692	17.365
				7941	131	499	0.361	25.065	0.996	0.690	27.112
				9635	137	1510	0.280	35.871	2.000	1.810	39.961
2	12	6	5	1980	48	0	0.276	0.944	0.0	0.0	1.224
				4077	123	0	0.278	5.620	0.0	0.0	5.902
				6163	132	0	0.278	7.329	0.0	0.0	7.612
				7941	150	0	0.364	13.016	0.0	0.0	13.385
				9635	159	0	0.277	19.756	0.0	0.0	20.037
3	12	6	5	1980	48	0	0.277	0.940	0.0	0.0	1.221
				4077	78	0	0.278	1.223	0.0	0.0	1.506
				6163	109	0	0.279	10.027	0.0	0.0	10.311
				7941	137	0	0.365	11.972	0.0	0.0	12.342
				9635	131	0	0.276	14.780	0.0	0.0	15.060
4	12	6	5	1980	40	0	0.277	0.948	0.0	0.0	1.229
				4077	78	0	0.277	1.227	0.0	0.0	1.509
				6163	126	13	0.282	4.076	0.331	0.128	4.817
				7941	159	13	0.361	4.353	0.387	0.118	5.219
				9635	153	182	0.277	5.634	0.483	0.288	6.682

FIGURE 6.2: QUERY PROCESSING - HIGH ENERGY PHYSICS

FIGURE 6.2 CONTD: PROCESSING QUERIES - HIGH ENERGY PHYSICS

QUERY NO.	NO. OF SELECT CONDS.	NO. OF ANDs	NO. OF ORs	NO. OF RECORDS IN FILE	NO. OF COMPS.	NO. OF RECORDS FOUND	CPU TIME USED(SECONDS)				
							EDIT	AND	OR	LIST	TOTAL
5	2	1	0	1980	371	270	0.077	4.795	0.629	0.419	5.920
				4077	723	497	0.076	8.255	0.856	0.707	9.894
				6163	1089	659	0.079	10.760	0.970	0.858	12.667
				7941	1315	659	0.155	13.174	1.113	0.920	15.362
				9635	1456	753	0.077	15.138	1.097	0.932	17.244
6	2	1	0	1980	832	196	0.077	4.277	0.556	0.338	5.248
				4077	1094	474	0.077	7.928	0.828	0.682	9.515
				6163	1247	788	0.078	11.867	1.161	1.010	14.116
				7941	1349	1093	0.150	16.145	1.629	1.350	19.274
				9635	1499	1391	0.079	19.459	1.849	1.652	23.039
7	2	1	0	1980	55	367	0.077	5.062	0.727	0.546	6.412
				4077	103	792	0.076	9.832	1.218	1.033	12.159
				6163	222	1378	0.076	15.013	1.810	1.671	18.570
				7941	253	1936	0.152	19.782	2.699	2.362	24.995
				9635	277	2030	0.076	22.777	2.503	2.371	27.727

6.4.1. Inverted Index Creation contd

statements. The reason for this is that at present the General Purpose Inverted Indexing System processes only Data Files in the form of unblocked records on an MTS Line File (a restriction that could be easily removed at a later stage).

The output from this run was a tabular Data File named GEOGRAPHIC DATA containing the following 9 fields:

1. SQUARE REFERENCE	Character String Length 2.
2. EASTING	Halfword Binary Length 2.
3. NORTHING	Halfword Binary Length 2.
4. ACCURACY	Halfword Binary Length 2.
5. PERIOD	Halfword Binary Length 2.
6. TYPE	Halfword Binary Length 2.
7. CONDITION	Halfword Binary Length 2.
8. PERSON REF.	Halfword Binary Length 2.
9. EXTRA	Halfword Binary Length 2.

The first three fields - SQUARE REFERENCE, EASTING and NORTHING uniquely identify by a Grid Reference an archaeological site. To be able to accommodate the queries initially expected, it was decided that inverted indexes would be created on these three fields.

The procedure for the runs was as follows:

1. Load the description of the Data File onto the File and associated Field Name Directories.
2. Run an estimate of the total blocks required for the number of records to be processed. In this case, as the Index File being used also held the High Energy Physics data previously mentioned plus available space, no extension of the Index File was required.
3. An inverted index for each of the fields SQUARE REFERENCE, EASTING and NORTHING was created and stored on the Index File.

The results for the creation runs on the archaeological data are shown in Figure 6.3. and are analysed in Chapter 7.

6.4.2. Processing of Queries

In discussion with Mr. Clack of the Archaeological Department, a basic query requirement was defined. In this query, it is required to find the sites which are located within a certain area. This search request may be qualified by further search parameters such as the period, type or condition of the sites.

FIGURE 6.3: INDEX CREATION - GEOGRAPHIC DATA

FILE CONTAINS 3280 RECORDS

FIELD			EST. NO. OF BLOCKS	ACTUAL NO. OF BLOCKS	NO. OF CONTROL ENTRIES	NO. OF UNIQUE KEYS	CPU TIME USED(SECONDS)				
NO.	TYPE	LENGTH					EDIT	ABST.	SORT	CREATE	TOTAL
1	1	2	136	54	3	3	0.156	11.624	9.991	8.778	30.549
2	2	2	136	108	104	2151	0.152	10.417	8.659	19.065	38.293
3	2	2	136	112	107	2290	0.151	10.367	8.633	19.031	38.182

6.4.2. Processing of Queries contd

It was therefore decided to define a set of queries based on the basic search required. The format of the queries was:

> Find all sites within Square Reference NZ where the Easting is Greater Than or Equal to 1000 and Less Than or Equal to N1, and the Northing is Greater Than or Equal to 1000 and Less Than or Equal to N1.

The values of N1 specified were 1500, 2000, 2500, 3000, 3500 and 4000 respectively to give 6 queries. Figure 6.4. shows the areas covered by each query. The format of the query as presented to the Index Selection Program is (taking N1 - 1500):

```
'GEOGRAPHIC DATA'
'SQUARE REFERENCE' 'EQ' 'NZ' 'AND'
'EASTING' 'GTE' '1000' 'AND'
'EASTING' 'LTE' '1500' 'AND'
'NORTHING' 'GTE' '1000' 'AND'
'NORTHING' 'LTE' '1500' 'END'
```

Thus, each query contains 5 Selection Conditions containing 4 ANDs. The Selection Conditions comprise one Equal Compare and four Range Searches (GTE or LTE).

Results obtained from the set of queries are displayed in Figure 6.5. and analysed in Chapter 7.

FIGURE 6.4: GEOGRAPHIC DATA - QUERY SEARCH AREAS

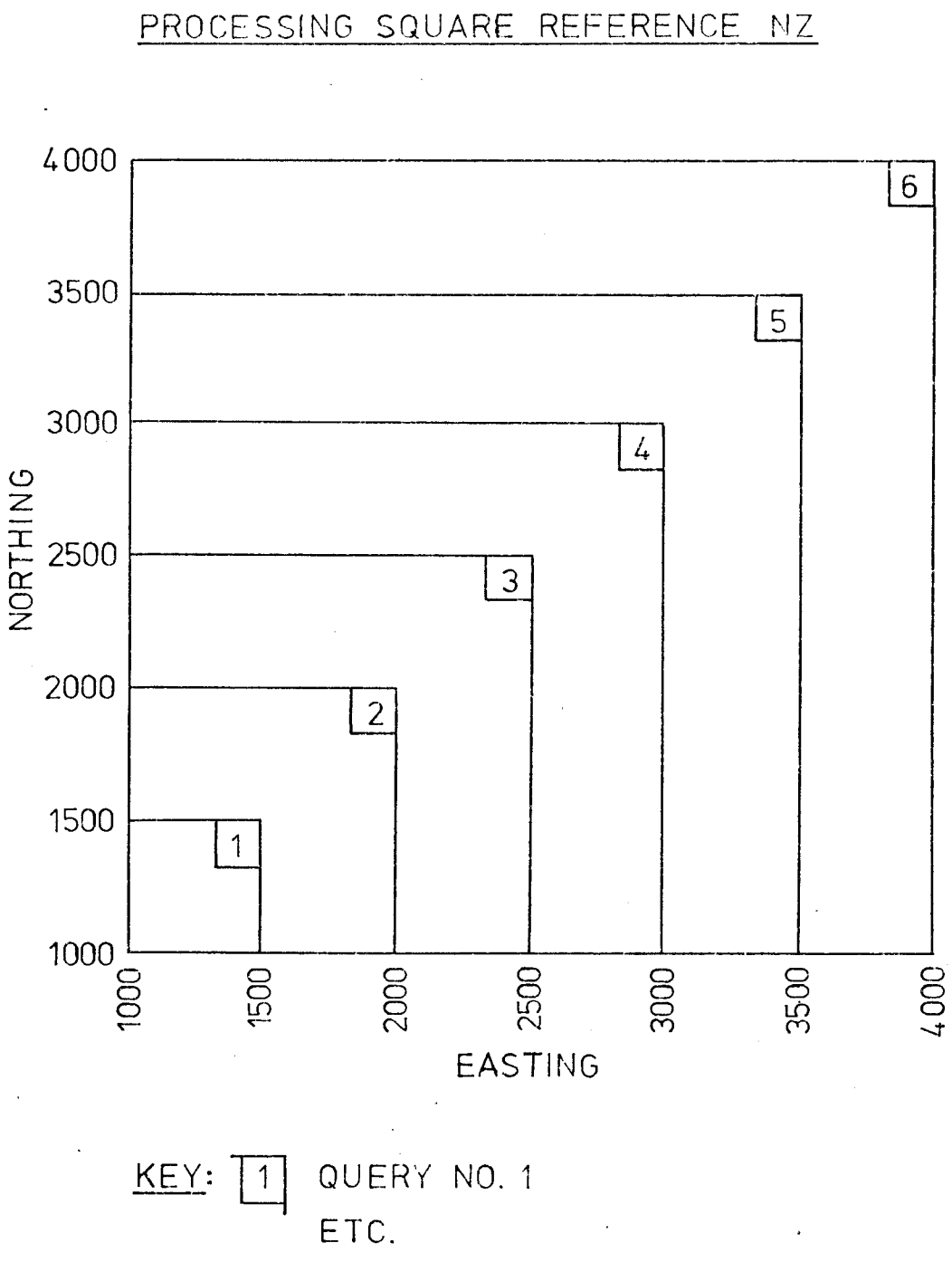


FIGURE 6.5: QUERY PROCESSING - GEOGRAPHIC DATA

FILE CONTAINS 3280 RECORDS

5 SELECTION CONDITIONS, 4 ANDs

QUERY NO.	NO. OF COMPS.	NO. OF RECORDS FOUND	CPU TIME USED(SECONDS)				
			EDIT	AND	OR	LIST	TOTAL
1	5712	3	0.222	24.328	0.320	0.081	24.951
2	6344	34	0.221	25.216	0.349	0.114	25.900
3	6840	89	0.222	26.611	0.398	0.171	27.402
4	7384	148	0.221	28.432	0.459	0.235	29.347
5	7832	271	0.221	29.945	0.582	0.368	31.116
6	8248	338	0.224	30.856	0.653	0.453	32.186

7.1. Introduction

The General Purpose Inverted Indexing System has been set up and is working on the NUMAC IBM System/360 Model 67 under MTS. It was originally logically tested using a Data File of Minicomputer data (see Chapter 5); and this was followed up by Performance Testing using Data Files of High Energy Physics and Geographic data (see Chapter 6). This chapter considers the following points arising from the above work:

1. Analysis of the Performance Testing data obtained and discussion of the results.
2. Identification of various ways by which the performance of the system might be improved.
3. Definition of areas in which the function of the system could be further developed.

7.2. Analysis of Performance Testing Data

Performance Testing was carried out on two Data Files - High Energy Physics, and Geographic Data on Archaeological Sites. The procedure in each case followed the same sequence:

1. Ensure that the Data File is in a Tabular unblocked form. If not, a program must be written to convert it from its original form. For the High Energy Physics Data which was stored in a hierarchical form, this involved writing a full conversion program; while for the Geographic Data, a program simply to unblock the original Data File sufficed.
2. Load the File Description into the Index File Directories.
3. Determine which fields in the Data File need to be indexed to answer the required queries.
4. Run an estimate of the Index File space required for the inverted indexes specified and allocate enough space to accommodate them.
5. Create the required inverted indexes on the Index File.
6. Run realistic queries using the Index Selection program and display the results.

In analysing the performance of the system, it was decided to concentrate on the two portions of the system which consumed most CPU time. These were:

1. Index Creation.
2. Query Processing.

Coding was inserted in the appropriate programs to record the CPU time used in the various portions of the program, runs were made, and the

7.2. Analysis of Performance Testing Data contd

results logged (see Chapter 6).

It may be of some help at this point to define the author's concept of performance testing. This may be expressed as follows:

> The objective of Performance Testing on the system is to determine the system performance in processing realistic quantities of data, where a realistic quantity using an actual Data File will almost always be in excess of 1000 records. The processing of this data is to duplicate the processing methods that might be used in other systems. Thus, the performance figures obtained are meant to provide guidelines as to how the system will react to real-life situations, rather than to analyse the internal performance of the system in detail.

Analysis and discussion of the results obtained for Index Creation and Selection follows below.

7.2.1. Inverted Index Creation

As discussed in Chapter 6, two Data Files were used for performance testing - the High Energy Physics Databank and Geographical Data on Archaeological Sites.

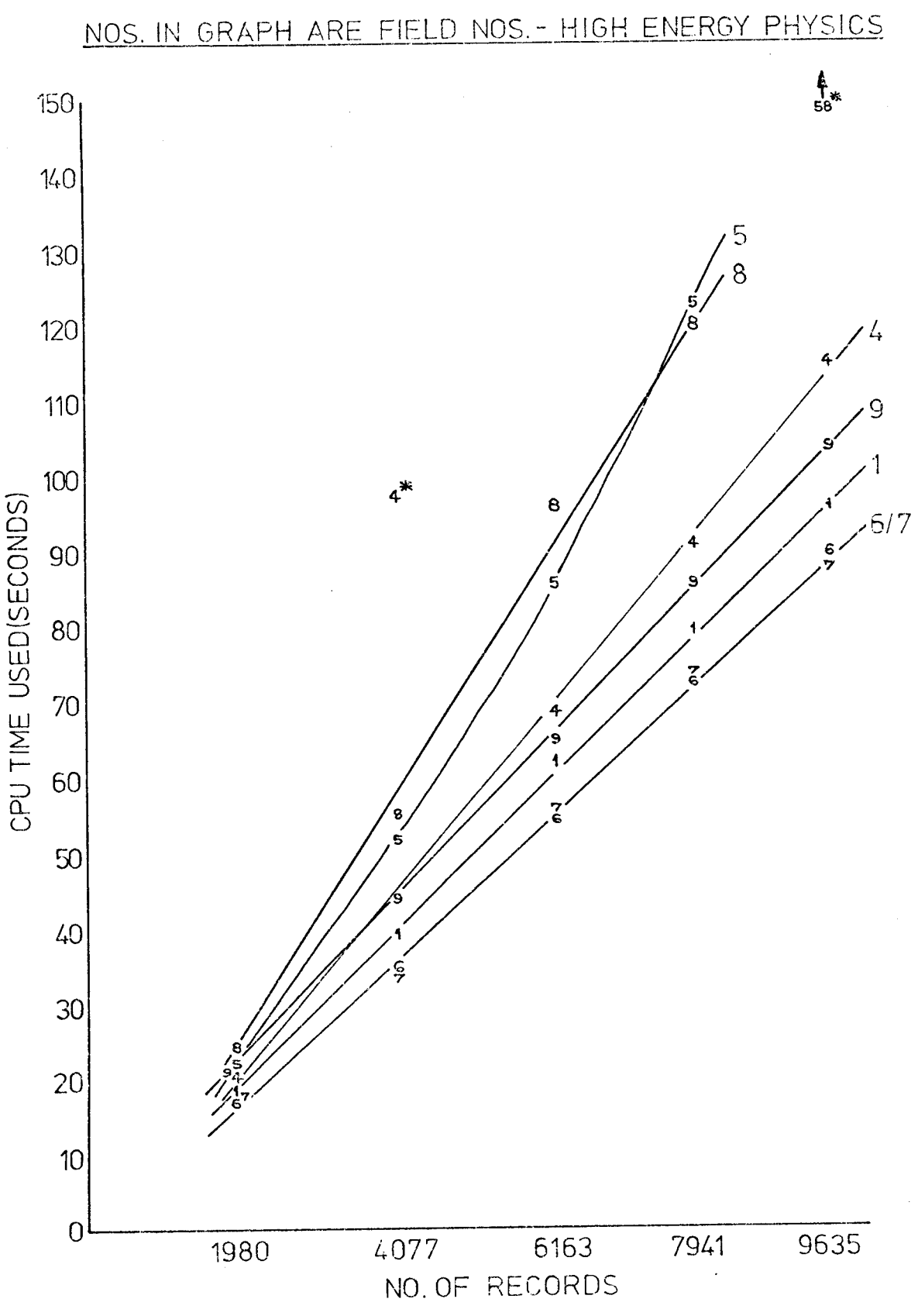
In testing on the High Energy Physics data, it was decided that inverted indexes would need to be created on 7 fields:

Field No.	Name	Type
1	EXPERIMENT NO.	Fullword Binary Length 4.
4	PROCESS CODE	Character String Length 13.
5	INFORMATION CODE	Character String Length 10.
6	ENERGY CODE	Character String Length 10.
7	ANGLE CODE	Character String Length 10.
8	ENERGY	Floating Point Length 4.
9	ANGLE	Floating Point Length 4.

Inverted indexes were created for these fields in turn from tabular Data Files abstracted from the main databank containing 1980, 4077, 6163, 7941 and 9635 records respectively. The timings in CPU time used (seconds) against number of records are graphed in Figure 7.1. The numbers in the graph indicate the above field numbers in the Tabular High Energy Physics record.

Next, in processing the Geographic Data, it was decided to create inverted indexes on 3 fields:

FIGURE 7.1: GRAPH INDEX CREATION TIME VS. NO. OF RECORDS



*ANOMALOUS RESULTS

7.2.1. Inverted Index Creation contd

Field No.	Name	Type
1	SQUARE REFERENCE	Character String Length 2.
2	EASTING	Halfword Binary Length 2.
3	NORTHING	Halfword Binary Length 2.

Inverted indexes were created for these fields from a tabular Data File containing 3280 records.

Analysis of the results obtained from the above runs yields a number of points for consideration:

1. For creation of an inverted index on a specified field, there appears in most cases to be a linear relationship between the number of records in the Data File (and hence the number of field values to be processed) and the CPU time used. Therefore, it should be possible to be able to predict approximately the time that would be taken to create an inverted index for a specified number of records.
2. On looking at Figure 7.1., it appears that differing amounts of CPU time per entry are required for inverted index creation on different fields. The average figures for both the High Energy Physics and Geographic Data indexes are shown in Figure 7.2. (anomalous results are excluded). An average taken over all the High Energy Physics index creation runs gives an overall average CPU time of 0.0132 seconds/entry.
3. There appears to be varying correlations in index creation times between like field types.
   For example, in the High Energy Physics Data File, fields 5, 6 and 7 are all Character Strings of length 10 bytes. While fields 6 (ENERGY CODE) and 7 (ANGLE CODE) give virtually identical creation times, field 5 (INFORMATION CODE) uses considerably more CPU time and may (if the result for 9635 records is not anomalous) have not a linear relationship but an almost exponential one to the number of records.
   Similarly, for High Energy Physics also, both fields 8 (ENERGY) and 9 (ANGLE) are both Fullword Floating Point numbers of length 4 bytes, but give very different creation rates.
   Conversely, in the Geographic Data fields indexed, fields 2 (EASTING) and 3 (NORTHING) give virtually similar results.

FIGURE 7.2: INDEX CREATION - CPU TIMES PER INDEX ENTRY

1. HIGH ENERGY PHYSICS(EXCLUDING ANOMALOUS RESULTS)

FIELD NO.	FIELD TYPE	FIELD LENGTH	AVERAGE CPU TIME
1	3	4	0.0100
4	1	13	0.0115
5	1	10	0.0137
6	1	10	0.0093
7	1	10	0.0093
8	4	4	0.0145
9	4	4	0.0111

AVERAGE = 0.0132 SECS/ENTRY

2. GEOGRAPHIC DATA

FIELD NO.	FIELD TYPE	FIELD LENGTH	AVERAGE CPU TIME
1	1	2	0.0093
2	2	2	0.0117
3	2	2	0.0116

AVERAGE = 0.0109 SECS/ENTRY

7.2.1. Inverted Index Creation contd

4. The reasons for varying correlations found in 3. above may be due to variations in either (a) field length, or (b) the average list length when the inverted index has been created.
   It was therefore decided to analyse the CPU time used in the various portions of the Index Creation program with the following results:
   (a) The time taken to Edit the input index creation command is minimal and constant.
   (b) In the Abstract and Sort phases, there appears to be no correlation with either field length or average list length (the latter not being surprising as the inverted index has not yet been created).
   (c) In the Create stage when the inverted index is finally formed, there appears to be a direct correlation - The shorter the average list length, the longer the creation time per entry (as again would be expected as more disk I/O must take place with more writing to disk of unique field values).
5. Analysis of the CPU time spent in the various phases of the Index Creation program gives the following percentage breakdown:

Edit	0.1
Abstract	39.7
Sort	33.9
Create	26.3

   Consideration for performance improvement should therefore be aimed at the Abstract, Sort and Create phases.
6. There appear to be some anomalous results in creating inverted indexes on the High Energy Physics data - these were rerun but again gave similar results. No explanation is advanced for these results, nor is further investigation recommended as later recommendations for performance improvements would tend to extensively modify the system operation.

From the above analysis, we may arrive in summary at specimen CPU times on the NUMAC IBM System/360 Model 67 for creation of sufficient inverted indexes on a Data File to meet a specified demand:

1. High Energy Physics.
   Create inverted indexes on 7 fields from 9635 records at an average of 0.0132 seconds CPU time per entry:
   Total CPU Time required = 890 seconds ≙ 15 minutes

7.2.1. Inverted Index Creation contd

2. Geographic Data.

Create inverted indexes on 3 fields from 3280 records at an average of 0.0109 seconds CPU time per entry:
Total CPU Time required = 107 seconds ≏ 1.75 minutes

7.2.2. Query Processing

As described in Chapter 6, a number of queries were defined and solved using the General Purpose Inverted Indexing System. Three main types of query could be defined:

A. Queries 1 - 4 on the High Energy Physics inverted indexes. These were all queries containing 12 Selection Conditions linked by 6 AND and 5 OR logical processes. It should be noted that queries 2 and 3 find no records on the Data File satisfying the search criteria - the effects of this are discussed later. All of the Selection Conditions involved Equal Compares on Character fields. Each query was run on inverted indexes created from varying sizes of tabular Data Files abstracted from the databank containing 1980, 4077, 6163, 7941 and 9635 records respectively.

B. Queries 5 - 7 on the High Energy Physics inverted indexes. These each contained 2 Selection Conditions linked by 1 AND logical process. Each query contained 1 Selection Condition giving an Equal Compare on a Character field ANDed with 1 Selection Condition performing a Range Search on a Floating Point field. Each query was run on inverted indexes created from varying sizes of tabular Data files abstracted from the databank containing 1980, 4077, 6163, 7941 and 9635 records respectively.

C. Queries 1 - 6 on the Geographic Data inverted indexes. These queries each contained 5 Selection Conditions linked by 4 AND logical processes. In each query, the first Selection Condition required an Equal Compare on a Character field followed by 4 Selection Conditions each containing a Range Search on a Binary field. The queries performed varying searches on inverted indexes created from a Data File containing 3280 records.

The timings for query types A and B (the High Energy Physics queries) as expressed in CPU time (seconds) against number of records in the Data File are graphed in Figure 7.3.

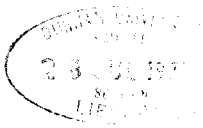
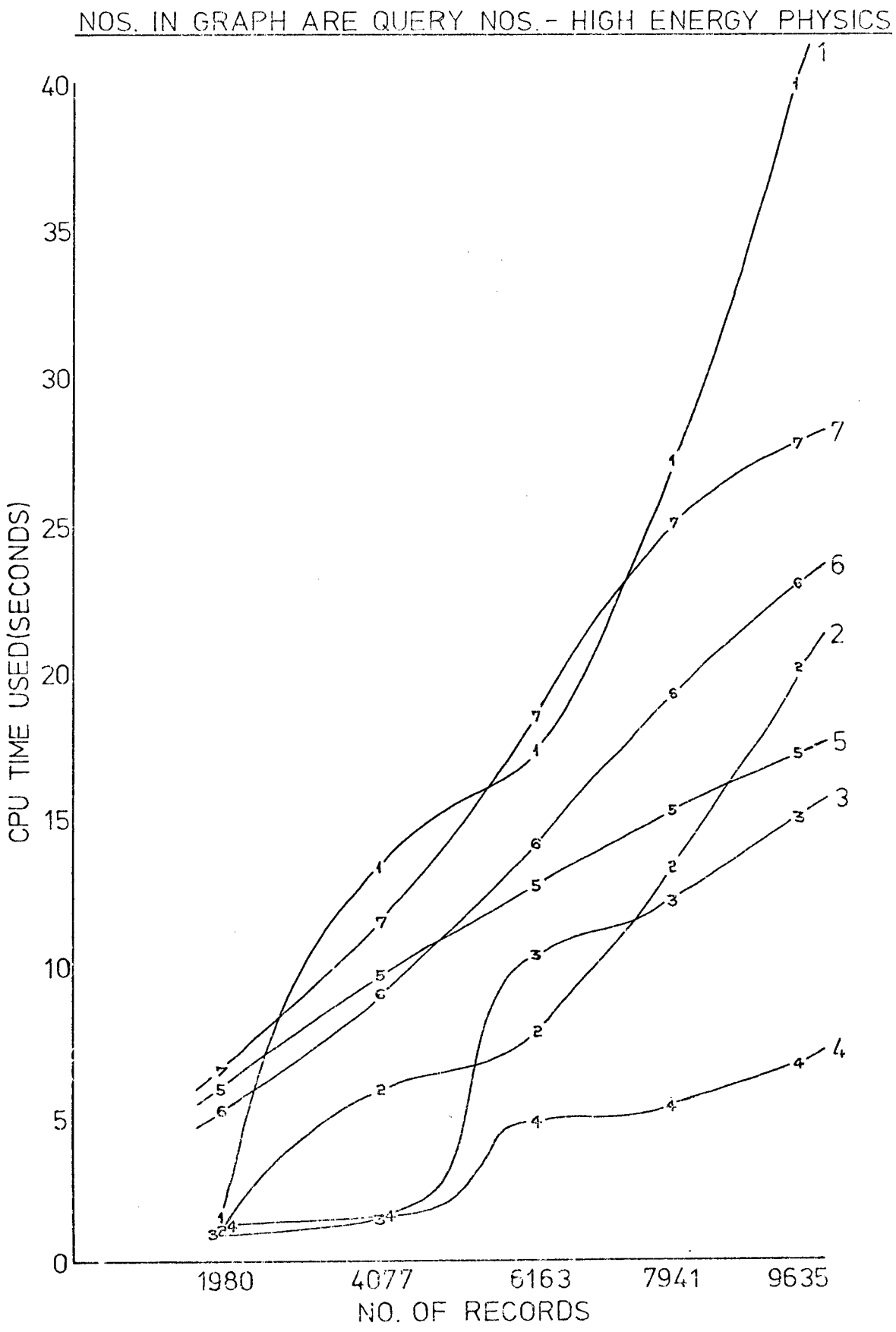


FIGURE 7.3: GRAPH QUERY PROCESSING TIME VS. NO. OF RECORDS



7.2.2. Query Processing contd

Analysis of the results obtained from processing the above queries raises the following points for consideration:

1. Inspection of Figure 7.3. would appear to show that there is no linear correlation between CPU time used to answer queries and the number of records in the inverted indexes accessed. This would be as expected, as perusal of the coding would appear to show that retrieval times would be dependent on a number of factors - number of Selection Conditions, number of comparisons made, and number of addresses found.
2. Further analysis of the CPU time used in the component phases of the Index Selection program gives the following results:
   (a) The time taken to Edit the input Selection Conditions would appear to be dependent on the number of Selection Conditions entered.
   (b) The time used in the AND phase is probably dependent on the number and type of Selection Conditions, the number of comparisons required against the inverted index entries (each of which requires a call to a subroutine), and the number of addresses found (list lengths for sorting).
   The type of Selection Condition would probably affect performance in that a range search (Less Than, Less Than or Equal, Greater Than, Greater Than or Equal) would probably take longer than an Equal Compare due to the number of comparisons required.
   (c) The time used in the OR and List stages would appear to be directly proportional to the number of record addresses found.
3. Analysis of the CPU time spent in various phases of the Index Selection program gives the following percentage breakdown:

Edit	1.4
AND	90.2
OR	4.6
List	3.8

   The AND processing phase would therefore appear to be the prime candidate for examination with a view to performance improvements.
4. No anomalous results appear to have been obtained.

7.2.2. Query Processing contd

From consideration of the types of queries and the analysis of the results, three representative results may be defined, the factors involved shown, and the CPU time used displayed as follows:

1. Query Type A.
   High Energy Physics Select Query 1.
   Data File contains 9635 records.
   12 Selection Conditions, all Equal Compares linked by 6 ANDs and 5 ORs.
   No. of comparisons 137, Records found 1510.
   CPU Time required = 39.961 ≏ 40 seconds.
2. Query Type B.
   High Energy Physics Select Query 7.
   Data File contains 9635 records.
   2 Selection Conditions comprising 1 Equal Compare and 1 Range Search linked by 1 AND.
   No. of comparisons 277, Records found 2030.
   CPU Time required = 27.727 ≏ 28 seconds.
3. Query Type C.
   Geographic Data Select Query 6.
   Data File contains 3280 records.
   5 Selection Conditions comprising 1 Equal Compare and 4 Range Searches linked by 4 ANDs.
   No. of comparisons 8248, Records found 338.
   CPU Time required = 32.186 ≏ 32 seconds.

7.3. Improvement of System Performance

The areas in which the performance of the General Purpose Inverted Indexing system may be improved are defined by analysis of the CPU time spent in the phases of the Index Creation and Selection programs. Breakdown of the totals in this way shows the following phases in which most CPU time is used:

1. In the Index Selection program, just over 90% of the CPU time is spent in the AND phase, where searching of the appropriate inverted indexes takes place together with AND processing and the sorting of lists of record addresses.
2. In the Index Creation program, almost all of the CPU time is used in the Abstract, Sort and Create phases; so that each must be examined.

7.3. Improvement of System Performance contd

Examination of the coding reveals a number of methods by which performance may be improved:

1. Improvement of disk I/O efficiency by allowing larger block sizes on the Index File (Ref: Index Selection - AND phase and Index Creation - Create phase).
2. Movement of data management processing from separate routines into the Index Creation and Selection programs, thus eliminating PL/1 Calls to the present Data Management routines (Ref: Index Selection - AND phase and Index Creation - Create phase).
3. Incorporation of the current comparison routine into the main Index Selection search module thus eliminating PL/1 Calls to this routine (Ref: Index Selection - AND phase).
4. Improvement of efficiency of abstraction of field value data from Data Files together with record addresses in the Index Creation program (Ref: Index Creation - Abstract phase).
5. Improvement of the efficiency of the sort in the Index Creation program by blocking the input and output records (Ref: Index Creation - Sort phase).

7.3.1. Provision of Larger Block Sizes on the Index File

In the initial design of the system, it was intended that the user would have the option of specifying the block size (in bytes) required for the Index File at Setup time. As discussed previously in Chapter 4, the maximum record (or Line) size in an MTS Line File is 255 bytes, so the system was therefore built round this limitation. Also, a Line File can have a maximum size of only 1 million bytes.

However, the possibility of a change to a different operating system was considered when the system was implemented, so that the current system contains MTS dependent disk I/O code only in the Data Management routines as well as in the Setup and Extension Utilities.

Therefore, to improve performance by reducing the amount of disk I/O, it would be recommended that the General Purpose Inverted Indexing System be converted to work under the IBM System/360 or /370 full Operating System, together with replacement of the MTS disk I/O coding with an OS direct access method such as PL/1 REGIONAL(1).

The advantages of this conversion would be as follows:

1. The user would be able to specify as an input parameter to the Index File Setup program the largest block size

7.3.1. Provision of Larger Block Sizes on the Index File contd

possible dependent on buffering considerations in main memory, thus decreasing the amount of disk I/O required.

2. The process of searching an inverted index is a two-stage one. First, a search is made through the top level of index - the Index Control Sub-File. When a match is obtained, a branch is made to search the appropriate block in the Value and Address Sub-File indicated by the Index Control Sub-File entry.

   As the number of entries in the Index Control Sub-File is to some extent dependent on the block size initially specified at setup time, use of larger block sizes would be expected to decrease the search time in the Index Control Sub-File.

3. The system would no longer be restricted by conditions for using Line Files placed upon it by MTS.

7.3.2. Improvement of Data Management Efficiency

Both the Index Creation and Selection programs make frequent use of the Data Management routines provided to both read and write information from disk. The original systems design provided continuity from one physical block to the next independent of positioning of data within the blocks, so that any request for data would be passed to the Data Management routines. Thus the situation has arisen that in the course of a run, many PL/1 Calls may be made for Data Management service, each Call having a certain amount of CPU overhead attached.

It is therefore suggested that the Data Management coding required to access the inverted indexes should be incorporated into the Index Creation and Selection programs where they are most used. Block I/O areas would be provided within these modules and disk I/O directly invoked The existing Data Management routines could be retained and used by other portions of the system where performance was not so important.

In this way, the CPU overhead involved at present PL/1 Calls to the Data Management routines would be removed from future considerations of system performance.

7.3.3. Improvement in Comparison Efficiency

The Index Selection program makes frequent use of a comparison routine which is accessed by a PL/1 Call. In this routine, varying comparisons are made dependent on the field type and a result returned.

7.3.3. Improvement in Comparison Efficiency contd

As in 7.3.2. above, performance could be improved by incorporating the comparison code into the Index Selection program where it is exclusively used. Thus, the CPU overhead involved in PL/1 Calls from the Index Selection program to the Comparison routine would be eliminated.

7.3.4. Index Creation - Improvement of Abstracting Efficiency

There are two recommendations as to how system performance could be improved in this phase. First, input tabular Data Files could be processed in a blocked form (initial implementation of this not being immediately feasible because of the limitation on MTS Line File record sizes, as well as the problems of not being able to specify record sizes prior to the entry of system commands). Second, the coding involved with abstracting the field value from its appropriate place in the record which is at present based on use of the PL/1 Substring (SUBSTR) function could be examined to see how it could be made more efficient.

7.3.5. Index Creation - Improvement of Sort Efficiency

At present, input and output to the Sort phase of the Index Creation program are not blocked. As in 7.3.4. above, this is due to limitations in MTS record sizes on Line Files together with the problems of not being able to specify record sizes prior the entry of a system command identifying the field which is to be indexed.
Further work to determine how abstracted field values with associated record addresses could be input to the Sort in a blocked form would improve the system performance.

7.4. Functional Enhancements to the System

The recommendations for performance improvement described above would have the effect of converting the General Purpose Inverted Indexing System from an experimental implementation to a system with the capability for processing problems on much larger Data Files. However, it is also considered that a number of improvements might enhance both the function and the user-friendly aspects of the system. The following possibilities might be considered:

1. Incorporation of the present system in a complete information retrieval system.
2. Addition of Interactive processing to the present Batch retrieval capabilities.
3. Addition of limited update capability to Data Files with corresponding update to inverted indexes.

7.4. Functional Enhancements to the System contd

4. Provision of varying means of output from Index Selection.

7.4.1. A Complete Information Retrieval System

The General Purpose Inverted Indexing System was designed to investigate the application of inverted file organization in a research environment. In the implementation, the functions involving the processing of the inverted indexes are treated as a virtually separate entity independent of the Data File (with the exception of abstraction of field values during Index Creation). The input to the system is a series of values which taken in context provide system commands; while the output from the system is a number of record addresses obtained from an index search. The major functional enhancement to the present system would be to use it as the Data Management section in a complete Information Retrieval system. The overall system would provide:

1. An information retrieval language with which to invoke various functions, among which would be the data management capabilities of the present system.
2. Provision of a modular structure which provides extensibility of functions and a common means of invocation. The functions provided by the present system would be:
   (a) Create a File Description.
   (b) Delete a File Description with associated inverted indexes.
   (c) Create an Inverted Index on a specified field.
   (d) Delete an Inverted Index.
   (e) Answer queries by searching Inverted Indexes.
3. Data manipulation and Report Generation facilities for processing data obtained from either answering of queries or other sources.

The present system utilities would be incorporated in the future system as Data Management Utilities.

7.4.2. Addition of Interactive Processing

The current system operates in batch mode only due to limitations in the use of MTS in the Index Selection program. The provision of interactive retrieval and processing facilities should be considered in any functional enhancement. This would have a direct bearing on provision of an information retrieval language as emphasis should be placed on the user-friendliness of the system.

7.4.3 Provision of Update Facilities

At present, if a record in the Data File is modified, the only way to modify the corresponding index entry is to delete the appropriate inverted index and then re-create it. While this may be adequate in most cases due to minimal update requirements (modifications simply being deferred until a major restructuring of the Data File becomes necessary); it would be reasonable in an enhanced system to provide an update capability. This would allow the user to add, delete, or update in place data records with corresponding modification of the appropriate inverted indexes. This might be done by leaving a certain amount of free space following each inverted index entry to accommodate additions.

7.4.4. Processing of Output Lists

The present output from the Index Selection program comprises two sets of information:

1. The number of record addresses found satisfying the entered Selection Conditions.
2. A list of record addresses.

While these outputs were satisfactory for proving and testing an experimental system, they would probably be inadequate in a production system. While the number of record addresses found could be either used or displayed, a display of record addresses (used in the research system for manual checking of results) would serve no useful purpose.

Three possible outputs from the Index Selection program might be defined for an enhanced system:

1. The number of record addresses found.
2. A list of record addresses stored on a work file for further processing.
3. An abstract of the records in the Data File satisfying the query stored on a work file for further processing.

A feedback from output to input could also be provided by including the facility of entering a list of record addresses obtained from a previous query into a further set of Selection Conditions.

An example of this type of use would be an interactive user having defined a query being presented with a result in the form of a large list of record addresses. Rather than display these at once because the query may not have been defined in enough detail, a further query would be defined which would narrow down the area of search.

APPENDIX A: GLOSSARY OF TERMS

NOTE: All terms described below relate to nomenclature used in development of the General Purpose Inverted Indexing System. It is assumed that the reader is familiar with File Organization concepts.
The terms are presented in the order in which they are introduced in the text, followed in each case by the Chapter and Section or Sub-Section reference in which they first appear.

Data File (1.1.)

A file of information held in tabular form. The file contains a number of data records, each of which contains the same number and type of data fields.

Index File (1.2.)

A direct access file on which a number of File Descriptions together with associated Inverted Indexes may be stored. The space within the Index File is controlled by the system.

Block (3.2.1.)

A fixed length directly addressable physical record within the Index File. The Index File is initially set up with a forward availability chain through all the blocks for space allocation and recovery.

Sub-File (3.2.1.)

A number of blocks on the Index File chained together for a particular usage. Processing within Sub-Files is forward sequential.

Index File Control Sub-File (3.2.1.)

A Sub-File comprising one block (Block No. 1) on the Index File which contains control information for both space allocation and recovery and control of the File Name Directory Sub-File (see below). It is brought into main memory at the start of a run and remains there till the end, when it is written back to disk.

File Name Directory Sub-File (3.3.1.)

Contains the names of all Data Files described in the system together with the number of fields.

Field Name Directory Sub-File (3.3.1)

There is one of these Sub-Files for each File Name on the File Name Directory, and it contains names and characteristics of each field in the data record.

Index Control Sub-File (3.4.1.)
Provides the top level in a two-level tree structure for inverted index searching. It is pointed to from the field name entry in the appropriate Field Name Directory Sub-File which is to be indexed. It contains a number of pointers to various points in the bottom level of the inverted index (Value and Address Sub-File - see below).

Value and Address Sub-File (3.4.1.)
The bottom level in the inverted index created for a specified field. It contains the field values and associated inverted lists abstracted from the Data File at index creation time.

Selection Condition (3.4.3.)
The basis on which queries on the system may be formulated. It specifies a search on a specified field for either a single field value or a number of field values according to a selection operator e.g. Equal, Greater Than, Less Than etc.

Logical Operator (3.4.3.)
An instruction to perform a Boolean logical operation on two lists of record addresses obtained from the system. Available logical operators are AND and OR.

"Complex" Query (3.4.3.)
A query on a Data File formulated by a number of Selection Conditions separated by Logical Operators.

Data Management (3.5.)
The lowest level (3) of the system. Controls Index File setup and processing including disk I/O.

File Description Directories (3.5.)
The next level (2) of the system, which controls the File Name and associated Field Name Directory Sub-Files.

Inverted Index Processing (3.5.)
The top level of the system (1). The functions provided here include inverted index creation and query processing.

Index File Control Record (4.2.1.)
Contained within the Index File Control Sub-File, this record controls this Index File space allocation and recovery.

Directory Control Record (4.2.1.)

This record is contained within the Index File Control Sub-File and controls the allocation of file name entries on the File Name Directory Sub-File.

APPENDIX B: BIBLIOGRAPHY

BLEIER AND VORHAUS (1968)
File Organization in the SDC Time-Shared Data Management System.
Bleier,R.E. and Vorhaus,A.H. Proc IFIP 1968, pp 1245 - 1252.

CARDENAS (1973)
Evaluation and Selection of File Organization - a Model and System.
A.F.Cardenas, Comm ACM, September 1973, Vol. 16, No. 9.

CODASYL (1971)
CODASYL Data Base Task Group Report - April 1971.

CODASYL (1971)
Feature Analysis of Generalized Data Management Systems.
Published by CODASYL Data Base Task Group, May 1971.
Chapter 1 in Computer Bulletin, Vol. 15, No. 4, April 1971.

CODD (1970)
A Relational Model of Data for Large Shared Data Banks.
E.F.Codd, Comm ACM, June 1970, Vol. 13, No. 3.

CODD (1971)
Normalized Data Base Structures - a Brief Tutorial.
E.F.Codd, IBM Research Report RJ 935, November 1971.

DODD (1969)
Elements of Data Management Systems.
George C. Dodd, General Motors Research Laboratories.
Computing Surveys, Vol. 1, No. 2, June 1969.

INGLIS (1974)
Inverted Indexes and Multi-List Structures.
Inglis,J. Computer Journal, Vol. 17, No. 1, February 1974.

LEFKOVITZ (1969)
File Structures for On-Line Systems.
D.Lefkovitz, Spartan Books, 1969.

MRI (1972)
System 2000 Data Base Management System. General Information Manual.
MRI Systems Corporation (US), 1972.

NOTLEY (1972)
The Peterlee IS/1 System.
M.G.Notley, IBM UK Scientific Centre Report UKSC-0018.

ROBERTS (1972)
File Organization Techniques.
D.C.Roberts, pp 115 - 174, Advances in Computers (Book), Vol. 12,
M.Rubinoff (Editor), Academic Press, 1972.

VOSE AND RICHARDSON (1972)
An Approach to Inverted Index Maintenance.
Vose,M.R. and Richardson,J.S. Computer Bulletin, Vol. 16, No. 5, May 1972.

APPENDIX C: SOURCE LISTINGS

```
£SIG MSV1 PW=BLANK P=100 PRINT=19 PROLTE=CNTR 'B.H.PEREIRA'
**LAST SIGNON WAS: 19:26.52
 USER "MSV1" SIGNED ON AT  20:34.15 ON 01-27-75
£SET CMDSKP=ON
£COPY *SOURCE* *SINK*
        /* IXMSPM - MTS DEPENDENT FILE AND I/O DEFINITIONS */
        DCL ZINDX FILE UPDATE;
        DCL (IHEREAD,IHERITE) ENTRY
              (,BIT(32),DEC FIXED(9,3),FILE);
        DCL ZBUFF CHAR(255) EXT,
            ZMOD BIT(32) INIT ((32)'0'B) EXT,
            ZLINE DEC FIXED(9,3) EXT;
        /* DEFINE DATA AREA SIZE FIELD FOR ALLOCATING STORAGE */
        /* ***** INITIAL VALUE OF 255 IS MTS DEPENDENT ***** */
        DCL ZSIZE BIN FIXED EXT INIT(255);
        /* DEFINE BLOCK AND OFFSET INDICATOR */
        DCL 1 ZPOSN EXT,
              2 BLOCK BIN FIXED(31,0),
              2 OFFST BIN FIXED;
        /* DEFINE STORAGE FOR INDEX FILE CONTROL RECORD */
        DCL ZCAREA CHAR(16) EXT,
              ZCPTR PTR EXT;
        DCL 1 ZCTRL BASED(ZCPTR),
              2 FAVAL BIN FIXED(31,0),
              2 (NOBLKS,NABLKS) BIN FIXED(31,0),
              2 DASIZE BIN FIXED,
              2 FIXPTR BIN FIXED;
        /* DEFINE I/O AREA AND WRITE SWITCH FOR PROCESSING BLOCKS */
        DCL ZIOPTR PTR EXT;
        DCL 1 ZIOAR BASED(ZIOPTR),
              2 NAVAL BIN FIXED(31,0),
              2 DAREA CHAR(251); /* ***** MTS DEPENDENT ***** */
        DCL ZWRSW BIN FIXED INIT(0) EXT;
        /* DEFINE STORAGE FOR BLOCK IN CORE INDICATOR */
        DCL ZABLK BIN FIXED(31,0) EXT;
```

```
£COPY *SOURCE* *SINK*
        /* IXUSTP - SET UP A BLANK INDEX FILE */
ZUSTP:  PROC OPTIONS(MAIN);
        /* THIS PROGRAM SETS UP A BLANK INDEX AND INITIALIZES
           REQUIRED VALUES */
        /* DEFINE INPUT AND PRINT FILES */
        DCL INFILE FILE INPUT, REPORT FILE OUTPUT;
        %INCLUDE IXMSPM;
        %INCLUDE IXMDPM;
        /* TURN ON INDEXED BIT OF ZMOD - ***** MTS DEPENDENT ***** */
        SUBSTR(ZMOD,21) = '1'B;
        /* MAKE CONTROL RECORD AND I/O PROCESSING AREAS ADDRESSABLE */
        ZCPTR = ADDR(ZCAREA);
        ZIOPTR = ADDR(ZBUFF);
        /* MAKE DIRECTORY CONTROL RECORD ADDRESSABLE */
        ZDPTR = ADDR(ZDAREA);
        /* CARD INPUT GIVES THE NUMBER OF BLOCKS TO BE INITIALIZED */
        /** ***** WHEN NOT RUNNING UNDER MTS, IT WILL ALSO BE
           NECESSARY TO READ IN THE BLOCK SIZE INTO ZSIZE ***** */
        GET FILE(INFILE) LIST(NOBLKS);
        /* PRINT HEADING AND INPUT PARAMETERS */
        PUT FILE(REPORT) SKIP(2) EDIT
              ('INDEX SETUP UTILITY')(A);
        PUT FILE(REPORT) SKIP EDIT
              ('====================')(A);
        PUT FILE(REPORT) SKIP(2) EDIT
              ('NO. OF BLOCKS SPECIFIED = ',NOBLKS)(A,F(6));
        PUT FILE(REPORT) SKIP(2) EDIT
              ('BLOCK SIZE SPECIFIED    = ',ZSIZE)(A,F(6));
        PUT FILE(REPORT) SKIP(2) EDIT
              ('INDEX SETUP STARTED')(A);
        /* INITIALIZE INDEX FILE CONTROL INFORMATION */
        FAVAL = 3; /* RECORD 2 RESERVED FOR FILE NAME INDEX */
        NABLKS = NOBLKS - 2;
        DASIZE = ZSIZE - 4;
        FIXPTR = 2;
        ZBUFF = ZCAREA;
        /* INITIALIZE DIRECTORY CONTROL RECORD AND MOVE IT INTO
           THE INDEX FILE CONTROL RECORD */
        DINUM = 0;
        DUNUM = 0;
        DUPTR = 0;
        FILEN = 28;
        FLDLN = 32;
        SUBSTR(ZBUFF,17,10) = ZDAREA;
        /* WRITE INDEX FILE CONTROL BLOCK */
        /* WRITE INDEX FILE CONTROL BLOCK */
        ZLINE = 1;
        CALL IHERITE(ZBUFF,ZMOD,ZLINE,ZINDX);
        /* INITIALIZE AND WRITE BLANK INDEX BLOCK */
        NAVAL = -1;
        ZLINE = 2;
        CALL IHERITE(ZBUFF,ZMOD,ZLINE,ZINDX);
        /* INITIALIZE AVAILABILITY CHAIN IN REMAINING BLOCKS */
        DO I = 3 TO NOBLKS;
              IF I = NOBLKS THEN NAVAL = -1;
                            ELSE NAVAL = I + 1 ;
              ZLINE = I;
              CALL IHERITE(ZBUFF,ZMOD,ZLINE,ZINDX);
```

```
END;
/* END OF SETUP - PRINT MESSAGE AND EXIT */
PUT FILE(REPORT) SKIP(2) EDIT
      ('INDEX SETUP SUCCESSFULLY COMPLETED')(A);
PUT FILE(REPORT) SKIP(2) EDIT('END OF RUN')(A);
END ZUSTP;
```

```
£COPY *SOURCE* *SINK*
        /* IXUSTX - EXTEND AN EXISTING INDEX FILE */
ZUSTX:  PROC OPTIONS(MAIN);
        /* THIS PROGRAM ENLARGES AN EXISTING INDEX BY CHAINING ON
           EXTRA AVAILABLE BLOCKS. NOTE THAT THE PROGRAM ASSUMES
           EXTENSION OF A FILE OR DATA SET IN PLACE IS POSSIBLE -
           THIS MAY BE MTS DEPENDENT */
        /* DEFINE INPUT AND PRINT FILES */
        DCL INFILE FILE INPUT,
            REPORT FILE OUTPUT;
        %INCLUDE IXMSPM;
        %INCLUDE IXMDPM;
        /* DEFINE EXTENSION SIZE STORAGE */
        DCL NFINX BIN FIXED(31,0);
        /* TURN ON INDEXED BIT OF ZMOD - ***** MTS DEPENDENT ***** */
        SUBSTR(ZMOD,31) = '1'B;
        /* MAKE CONTROL RECORD AND I/O PROCESSING AREAS ADDRESSABLE */
        ZCPTR = ADDR(ZCAREA);
        ZIOPTR = ADDR(ZBUFF);
        /* MAKE DIRECTORY CONTROL RECORD ADDRESSABLE */
        ZDPTR = ADDR(ZDAREA);
        /* CARD INPUT GIVES THE NUMBER OF BLOCKS SPECIFIED */
        /* ***** WHEN NOT RUNNING UNDER MTS, IT WILL ALSO BE
           NECESSARY TO READ THE BLOCK SIZE INTO ZSIZE ***** */
        GET FILE(INFILE) LIST(NFINX);
        /* PRINT HEADING AND INPUT PARAMETERS */
        PUT FILE(REPORT) SKIP(2) EDIT
              ('INDEX EXTENSION UTILITY')(A);
        PUT FILE(REPORT) SKIP EDIT
              ('======================')(A);
        PUT FILE(REPORT) SKIP(2) EDIT
              ('NO. OF BLOCKS SPECIFIED = ',NFINX)(A,F(6));
        PUT FILE(REPORT) SKIP(2) EDIT
              ('BLOCK SIZE SPECIFIED = ',ZSIZE)(A,F(6));
        /* GET INDEX FILE CONTROL INFORMATION */
        ZLINE = 1;
        CALL IHEREAD(ZBUFF,ZMOD,ZLINE,ZINDX);
        ZCAREA = ZBUFF;
        ZDAREA = SUBSTR(ZBUFF,17,10);
        /* DETERMINE IF BLOCK ENTERED EXCEEDS PRESENT NO. OF BLOCKS.
           IF NOT PRINT AN ERROR MESSAGE AND EXIT */
        IF NFINX > NOBLKS THEN ;
                          ELSE DO;
              PUT FILE(REPORT) SKIP(2) EDIT
('NUMBER OF BLOCKS ENTERED DOES NOT EXCEED PRESENT NO. OF BLOCKS')(A);
              GO TO STX4;
              END;
        /* PRINT START OF INDEX EXTENSION MESSAGE */
        PUT FILE(REPORT) SKIP(2) EDIT
              ('INDEX EXTENSION STARTED')(A);
        /* LOOP THROUGH AVAILABITY CHAIN TO DETERMINE LAST AVAILABLE
           BLOCK ON THE CURRENT FILE */
        IF FAVAL = -1 THEN DO;
              FAVAL = NOBLKS + 1;
              GO TO STX3;
              END;
        ZLINE = FAVAL;
STX1:   CALL IHEREAD(ZBUFF,ZMOD,ZLINE,ZINDX);
        IF NAVAL = -1 THEN GO TO STX2;
```

```
         ZLINE = NAVAL;
         GO TO STX1;
         /* WE NOW HAVE THE LAST BLOCK ON THE AVAILABILITY CHAIN.
            UPDATE THE FORWARD POINTER AND WRITE IT TO THE NEW INDEX */
STX2:    NAVAL = NOBLKS + 1;
         CALL IWRITE(ZBUFF,ZMOD,ZLINE,ZINDX);
         /* NOW SET UP CHAIN THROUGH THE REST OF THE NEW BLOCKS */
STX3:    DO I = (NOBLKS + 1) TO NFINX;
               IF I = NFINX THEN NAVAL = -1;
                            ELSE NAVAL = I + 1;
               ZLINE = I;
               CALL IWRITE(ZBUFF,ZMOD,ZLINE,ZINDX);
         END;
         /* MODIFY THE INDEX CONTROL RECORD AND WRITE IT BACK */
         NABLKS = NABLKS + (NFINX - NOBLKS) ;
         NOBLKS = NFINX;
         ZBUFF = ZCAREA;
         SUBSTR(ZBUFF,17,10) = ZDAREA;
         ZLINE = 1;
         CALL IWRITE(ZBUFF,ZMOD,ZLINE,ZINDX);
         /* PRINT SUCCESSFUL JOB COMPLETION MESSAGE */
         PUT FILE(REPORT) SKIP(2) EDIT
               ('INDEX EXTENSION SUCCESSFULLY COMPLETED')(A);
         /* PRINT END OF RUN MESSAGE AND EXIT */
STX4:    PUT FILE(REPORT) SKIP(2) EDIT('END OF RUN')(A);
         END ZUSTX;
```

```
£COPY *SOURCE* *SINK*
        /* IXUSTR - DISPLAY STATUS OF INDEX FILE */
ZUSTR:  PROC OPTIONS(MAIN);
        /* DEFINE PRINT OUTPUT FILE */
        DCL REPORT FILE OUTPUT;
        %INCLUDE IXMSPM;
        /* OPEN THE INDEX FILE */
        CALL ZOOPN;
        /* DISPLAY REPORT HEADING */
        PUT FILE(REPORT) SKIP(2) EDIT
              ('INDEX FILE STATUS REPORT')(A);
        PUT FILE(REPORT) SKIP EDIT
              ('========================')(A);
        /* DISPLAY FILE STATUS INFORMATION */
        PUT FILE(REPORT) SKIP(2) EDIT
              ('INDEX CONTROL INFORMATION')(A);
        PUT FILE(REPORT) SKIP EDIT
              ('-------------------------')(A);
        PUT FILE(REPORT) SKIP(2) EDIT
              ('BLOCK SIZE (BYTES) = ',ZSIZE)(A,F(6));
        PUT FILE(REPORT) SKIP(2) EDIT
              ('DATA AREA SIZE = ',DASIZE)(A,F(6));
        PUT FILE(REPORT) SKIP(2) EDIT
              ('NUMBER OF BLOCKS IN INDEX FILE = ',NOBLKS)(A,F(10));
        PUT FILE(REPORT) SKIP(2) EDIT
              ('NUMBER OF AVAILABLE BLOCKS = ',NABLKS)(A,F(10));
        /* PRINT END OF RUN MESSAGE, CLOSE INDEX FILE AND EXIT */
        PUT FILE(REPORT) SKIP(2) EDIT('END OF RUN')(A);
        CALL ZOCLS;
        RETURN;
        END ZUSTR;
```

```
£COPY *SOURCE* *SINK*
        /* IXDOPN - OPEN THE INDEX FILE */
ZDOPN:  PROC;
        /* THIS ROUTINE OPENS THE INDEX FILE, LOADS CONTROL INFORMATION
           AND INITIALIZES VARIABLES */
        %INCLUDE IXMSPM;
        %INCLUDE IXMOPM;
        /* TURN ON INDEXED BIT OF MOD - ***** MTS DEPENDENT ***** */
        SUBSTR(ZMOD,31) = '1'B;
        /* MAKE CONTROL RECORD AND I/O PROCESSING AREAS ADDRESSABLE */
        ZOPTR = ADDR(ZOAREA);
        ZIOPTR = ADDR(ZBUFF);
        /* MAKE DIRECTORY CONTROL RECORD ADDRESSABLE */
        ZDPTR = ADDR(ZDAREA);
        /* READ INDEX FILE CONTROL RECORD INTO CORE */
        ZLINE = 1;
        CALL IHEREAD(ZBUFF,ZMOD,ZLINE,ZINDX);
        ZOAREA = ZBUFF;
        /* MOVE DIRECTORY CONTROL RECORD INTO PROCESSING AREA */
        ZDAREA = SUBSTR(ZBUFF,17,10);
        /* SET ZPOSN TO POINT TO START OF INDEX */
        BLOCK = FIXPTR;
        OFFST = 1;
        /* SET BLOCK IN CORE INDICATOR TO NULL */
        ZABLK = -1;
        /* SET WRITE INDICATOR TO ZERO */
        ZWRSW = 0;
        /* COMPLETION OF OPEN ROUTINE */
        RETURN;
        END ZDOPN;
```

```
£COPY *SOURCE* *SINK*
        /* IXDCLS - CLOSE THE INDEX FILE */
ZDCLS:  PROC;
        /* THIS ROUTINE TERMINATES PROCESSING ON THE INDEX FILE */
        %INCLUDE IXMSPM;
        %INCLUDE IXMDPM;
        /* IF THE WRITE INDICATOR IS SET, WRITE THE LAST BLOCK BACK
           FROM THE I/O AREA TO DISK */
        IF ZWRSW = 1 THEN DO;
              ZLINE = ZABLK;
              CALL IOWRITE(ZBUFF,ZMOD,ZLINE,ZINDX);
        END;
        /* THE LAST BLOCK HAS NOW BEEN WRITTEN. WRITE INDEX FILE
           CONTROL RECORD BACK TO DISK */
        ZBUFF = ZDAREA;
        /* MOVE DIRECTORY CONTROL RECORD INTO OUTPUT BUFFER */
        SUBSTR(ZBUFF,17,10) = ZDAREA;
        ZLINE = 1;
        CALL IOWRITE(ZBUFF,ZMOD,ZLINE,ZINDX);
        /* COMPLETION OF CLOSE ROUTINE */
        STOP;
        END ZDCLS;
```

```
£COPY *SOURCE* *SINK*
        /* IXDCRE - CREATE A NEW SUB-FILE */
ZDCRE:  PROC;
        /* THIS ROUTINE CREATES A ONE-BLOCK SUB-FILE ON REQUEST */
        /* DEFINE PRINT OUTPUT FILE */
        DCL REPORT FILE OUTPUT;
        %INCLUDE IXMSPM;
        /* IF NO STORAGE IS AVAILABLE, PRINT AN ERROR MESSAGE AND
           TERMINATE PROCESSING */
        IF FAVAL = -1 THEN DO;
              PUT FILE(REPORT) SKIP(2) EDIT
                    ('NO BLOCKS AVAILABLE - JOB TERMINATED')(A);
              CALL ZDCLS;
        END;
        BLOCK = FAVAL;
        /* FIRST CHECK TO SEE IF THE BLOCK REQUIRED IS IN CORE */
        IF ZABLK = BLOCK THEN GO TO CRE1;
        /* IF THE PREVIOUS BLOCK IN THE I/O AREA HAS BEEN UPDATED,
           WRITE IT BACK TO DISK AND RESET THE WRITE INDICATOR */
        IF ZWRSW = 1 THEN DO;
              ZLINE = ZABLK;
              CALL IHERITE(ZBUFF,ZMOD,ZLINE,ZINDX);
              ZWRSW = 0;
        END;
        /* NOW GET THE REQUIRED BLOCK */
        ZLINE = BLOCK;
        CALL IHEREAD(ZBUFF,ZMOD,ZLINE,ZINDX);
        ZABLK = BLOCK;
        /* RESET THE POINTERS TO INDICATE NEW VALUES */
CRE1:   FAVAL = NAVAL;
        NAVAL = -1;
        ZWRSW = 1;
        /* PRINT SUB-FILE CREATION MESSAGE */
        PUT FILE(REPORT) SKIP(2) EDIT
              ('***** SUB-FILE CREATED ON BLOCK ',BLOCK)(A,F(6));
        /* DECREMENT THE NUMBER OF AVAILABLE BLOCKS BY 1 */
        NABLKS = NABLKS - 1;
        /* SET OFFSET TO 1 */
        OFFST = 1;
        /* SUB-FILE IS CREATED - EXIT */
        RETURN;
        END ZDCRE;
```

```
£COPY *SOURCE* *SINK*
        /* IXDDST - DESTROY A SUB- FILE */
ZDDST:  PROC;
        /* THIS ROUTINE DESTROYS A SUB-FILE OF N BLOCKS AND RETURNS
           THEM TO THE AVAILABILITY CHAIN. THE ADDRESS OF THE FIRST
           BLOCK IS PASSED IN ZDPOSN */
        %INCLUDE IXMSPM;  DCL REPORT FILE OUTPUT;
        /* DEFINE A WORK AREA TO STORE FIRST AVAILABLE BLOCK */
        DCL WKADDR BIN FIXED(31,0);
        /* STORE FIRST AVAILABLE BLOCK IN WORK AREA */
        WKADDR = FAVAL;
        /* MOVE ADDRESS OF SUB-FILE TO BE DESTROYED INTO FAVAL */
        FAVAL = BLOCK;
        /* CHECK TO SEE IF THE FIRST BLOCK IS IN CORE */
        IF ZABLK = BLOCK THEN GO TO DST2;
        /* IF THE PREVIOUS BLOCK IN THE I/O AREA HAS BEEN UPDATED,
           WRITE IT BACK TO DISK AND RESET THE WRITE INDICATOR */
        IF ZWRSW = 1 THEN DO;
              ZLINE = ZABLK;
              CALL IHEWRITE(ZBUFF,ZMOD,ZLINE,ZINDX);
              ZWRSW = 0;
        END;
        /* GET THE REQUIRED BLOCK */
DST1:   ZLINE = BLOCK;
        CALL IHEREAD(ZBUFF,ZMOD,ZLINE,ZINDX);
        ZABLK = BLOCK;
        /* PRINT A MESSAGE SAYING BLOCK IS TO BE DELETED */
DST2:   PUT FILE(REPORT) SKIP EDIT
              ('***** SUB-FILE BLOCK ',BLOCK,' DELETED')(A,F(6),A);
        /* ADD 1 TO THE AVAILABLE NUMBER OF BLOCKS */
        NABLKS = NABLKS + 1;
        /* CHECK TO SEE IF THIS IS LAST BLOCK IN CHAIN */
        IF NAVAL = -1 THEN;
                      ELSE DO;
                     BLOCK = NAVAL;
                     GO TO DST1; /* BRANCH TO GET ANOTHER BLOCK */
        END;
        /* LINK UP DELETED SUB-FILE CHAIN WITH AVAILABILITY CHAIN,
           SET WRITE INDICATOR TO 1 AND EXIT */
        NAVAL = WKADDR;
        ZWRSW = 1;
        RETURN;
        END ZDDST;
```

```
£COPY *SOURCE* *SINK*
        /* IXDREA - GET NREAD CHARACTERS FROM A SUB-FILE */
ZDREA:  PROC(NREAD,PAREA);
        /* THIS ROUTINE GETS NREAD CHARACTERS FROM THE SUB-FILE
           POINTED TO BY ZPOSN AND MOVES THEM INTO THE AREA PAREA.
           ON COMPLETION THE OFFSET POINTER IS SET TO THE NEXT
           OFFSET POSITION FOLLOWING THE DATA MOVED */
        %INCLUDE IXMSPM;
        /* DEFINE A LABEL RETURN VARIABLE */
        DCL LABRC LABEL;
        /* DEFINE MASK OVER PASSED DATA AREA */
        DCL PAREA CHAR(8000);
        /* MOVE PASSED NO. OF CHARACTERS PARAMETER INTO A WORK AREA */
        NWORK = NREAD;
        /* SET UP A COUNTER FOR TRANSFERRING DATA */
        MWORK = 1;
        /* GET THE INDICATED BLOCK INTO CORE */
REA1:   LABRC = LABRC1;
        GO TO GETB;
        /* BEFORE STARTING TO MOVE DATA, CHECK TO SEE IF THE OFFSET
           IS WITHIN THE RANGE OF THE CURRENT BLOCK, AS IT MAY HAVE
           BEEN MODIFIED BY A FORWARD PSEUDO MOVE WHICH PROCESSED
           THE OFFSET */
LABRC1: IF OFFST > DASIZE THEN;
                          ELSE GO TO REA2;
        /* IF OFFSET IS GREATER THAN DATA AREA SIZE, THEN SET OFFSET
           TO DATA AREA SIZE AND THEN CALL ZDMOV WITH THE REMAINING
           NUMBER OF CHARACTERS TO BE MOVED */
        NCHAR = OFFST - DASIZE;
        OFFST = DASIZE;
        CALL ZDMOV(NCHAR);
        /* CHECK TO SEE IF READ CAN BE SATISFIED WITH THIS BLOCK */
REA2:   IF (DASIZE + 1 - OFFST) >= NWORK THEN GO TO REA3;
        /* CHECK TO SEE IF POINTER TO NEXT BLOCK IS NULL */
        IF NAVAL = -1 THEN DO;
              PUT FILE(REPORT) SKIP(2) EDIT
                    ('ATTEMPT TO READ OVER END OF SUB-FILE')(A);
              CALL ZDCLS;
        END;
        /* WE CAN MOVE A CERTAIN NUMBER OF CHARACTERS INTO THE DATA
           AREA. SET UP THE NUMBER OF CHARACTERS TO BE MOVED */
        NCHARS = DASIZE + 1 - OFFST;
        /* MOVE THE CHARACTERS INTO THE DATA AREA */
        SUBSTR(PAREA,MWORK,NCHARS) = SUBSTR(DAREA,OFFST,NCHARS);
        /* SET POINTERS TO NEW POSITIONS */
        MWORK = MWORK + NCHARS;
        NWORK = NWORK - NCHARS;
        OFFST = 1;
        /* LOAD NEXT BLOCK ADDRESS AND BRANCH TO READ IT */
        BLOCK = NAVAL;
        GO TO REA1;
        /* FINAL MOVE OF DATA */
REA3:   SUBSTR(PAREA,MWORK,NWORK)  = SUBSTR(DAREA,OFFST,NWORK);
        /* SET OFFSET TO ITS NEW VALUE */
        OFFST = OFFST + NWORK;
        /* WE MUST NOW CHECK IF OFFSET IS NOW GREATER THAN THE DATA
           AREA SIZE. THIS WILL OCCUR IF THE OFFSET IS SET TO 1, AND
           A READ EQUAL TO THE DATA AREA SIZE IS REQUESTED. IF SO,
           WE MUST CHAIN TO THE NEXT BLOCK OR ADD A NEW BLOCK, SET
```

```
            THE BLOCK TO POINT TO THE NEXT BLOCK AND THE OFFSET TO 1 */
         IF OFFST > DASIZE THEN DO;
               OFFST = 1;
               IF NAVAL = -1 THEN;
                             ELSE GO TO REA4;
         NBLKS = 1;
         CALL ZDOSF(NBLKS);
         LABRD = LABRD2; GO TO GETB;
LABRD2:
REA4:    BLOCK = NAVAL;
         END;
         /* END OF READ ROUTINE - EXIT */
         RETURN;
         /* GETB - IN-LINE CODING FOR GETTING A BLOCK INTO CORE */
         /* CHECK TO SEE IF THE BLOCK REQUIRED IS IN CORE */
GETB:    IF ZABLK = BLOCK THEN GO TO GETB1;
         /* IF THE PREVIOUS BLOCK IN THE I/O AREA HAS BEEN UPDATED;
            WRITE IT BACK TO DISK AND RESET THE WRITE INDICATOR */
         IF ZWRSW = 1 THEN DO;
               ZLINE = ZABLK;
               CALL IHERITE'ZBUFF,ZMOD,ZLINE,ZINDX);
               ZWRSW = 0;
         END;
         /* NOW GET THE REQUIRED BLOCK */
         ZLINE = BLOCK;
         CALL IHEREAD(ZBUFF,ZMOD,ZLINE,ZINDX);
         ZABLK = BLOCK;
GETB1:   GO TO LABRD;
         END ZDREA;
```

```
£COPY *SOURCE* *SINK*
        /* IXDWRT - PUT NWRTE CHARACTERS TO A SUB-FILE */
ZDWRT:  PROC(NWRTE,PAREA);
        /* THIS ROUTINE WRITES NWRTE CHARACTERS FROM THE AREA PAREA
           INTO THE SUB-FILE POINTED TO BY THE VARIABLE ZPOSN. ON
           COMPLETION THE OFFSET POINTER IS SET TO THE NEXT OFFSET
           POSITION FOLLOWING THE DATA WRITTEN OUT */
        %INCLUDE IXMSPM;
        /* DEFINE A LABEL RETURN VARIABLE */
        DCL LABWRT LABEL;
        /* DEFINE MASK OVER PASSED DATA AREA */
        DCL PAREA CHAR(8000);
        /* MOVE PASSED NUMBER OF CHARACTERS PARAMETER INTO A WORK
           AREA */
        NWORK = NWRTE;
        /* SET UP A COUNTER FOR TRANSFERRING DATA */
        MWORK = 1;
        /* GET THE INDICATED BLOCK INTO CORE */
WRT1:   LABWRT = LABWRT1;
        GO TO GETB;
        /* BEFORE STARTING TO MOVE DATA, CHECK TO SEE IF THE OFFSET
           IS WITHIN THE RANGE OF THE CURRENT BLOCK, AS IT MAY HAVE
           BEEN MODIFIED BY A FORWARD PSEUDO MOVE WHICH PROCESSED
           THE OFFSET */
LABWRT1:IF OFFST > DASIZE THEN;
                          ELSE GO TO WRT2;
        /* IF OFFSET IS GREATER THAN DATA AREA SIZE, THEN SET OFFSET
           TO DATA AREA SIZE AND THEN CALL ZDMOV WITH THE REMAINING
           NUMBER OF BYTES TO BE MOVED */
        NCHAR = OFFST - DASIZE;
        OFFST = DASIZE;
        CALL ZDMOV(NCHAR);
        /* CHECK TO SEE IF WRITE CAN BE SATISFIED WITH THIS BLOCK */
WRT2:   IF (DASIZE + 1 - OFFST) >= NWORK THEN GO TO WRT4;
        /* WE CAN MOVE A CERTAIN NUMBER OF CHARACTERS INTO THE OUTPUT
           AREA. SET UP THE NUMBER OF CHARACTERS TO BE MOVED */
        NCHARS = DASIZE + 1 - OFFST;
        /* MOVE THE CHARACTERS INTO THE I/O AREA */
        SUBSTR(DAREA,OFFST,NCHARS) = SUBSTR(PAREA,MWORK,NCHARS);
        /* SET POINTERS TO NEW POSITIONS */
        MWORK = MWORK + NCHARS;
        NWORK = NWORK - NCHARS;
        OFFST = 1;
        /* SET WRITE INDICATOR TO 1 */
        ZWRSW = 1;
        /* CHECK TO SEE IF POINTER TO NEXT BLOCK IS NULL */
        IF NAVAL = -1 THEN GO TO WRT3;
        /* LOAD NEXT BLOCK ADDRESS AND BRANCH TO READ IT */
WRT25:  BLOCK = NAVAL;
        GO TO WRT1;
        /* DETERMINE THE NUMBER OF BLOCKS TO BE ADDED AND CALL THE
           OVER SUB-FILE ROUTINE TO CREATE THEM */
WRT3:   NBLKS = 1 + ((NWORK - 1) / DASIZE);
        CALL ZDOSF(NBLKS);
        /* WE NOW HAVE THE EXTRA BLOCKS: CONTINUE PROCESSING ON
           PRESENT BLOCK */
        LABWRT = LABWRT2;
        GO TO GETB;
        /* BRANCH TO GET THE NEXT BLOCK */
```

```
LABWRT2:GO TO WRT25;
        /* FINAL MOVE OF DATA */
WRT4:   SUBSTR(DAREA,OFFST,NWORK) = SUBSTR(PAREA,MWORK,NWORK);
        /* SET OFFSET TO ITS NEW VALUE AND SET WRITE INDICATOR */
        OFFST = OFFST + NWORK;
        ZWRSW = 1;
        /* WE MUST NOW CHECK IF OFFSET IS NOW GREATER THAN THE DATA
           AREA SIZE. THIS WILL OCCUR IF THE OFFSET IS SET TO 1, AND
           A WRITE EQUAL TO THE DATA AREA SIZE IS REQUESTED. IF SO
           WE MUST CHAIN TO THE NEXT BLOCK OR ADD A NEW BLOCK, SET
           THE BLOCK TO POINT TO THE NEXT BLOCK AND OFFSET TO 1 */
        IF OFFST > DASIZE THEN DO;
              OFFST = 1;
              IF NAVAL = -1 THEN;
                            ELSE GO TO WRT5;
        NBLKS = 1;
        CALL ZDOSF(NBLKS);
        LABWRT = LABWRT3;
        GO TO GETB;
 LABWRT3:
WRT5:   BLOCK = NAVAL;
        END;
        /* END OF WRITE ROUTINE - EXIT */
        RETURN;
        /* GETB - IN-LINE CODING FOR GETTING A BLOCK INTO CORE */
        /* FIRST CHECK TO SEE IF THE BLOCK REQUIRED IS IN CORE */
GETB:   IF ZABLK = BLOCK THEN GO TO GETB1;
        /* IF THE PREVIOUS BLOCK IN THE I/O AREA HAS BEEN UPDATED,
           WRITE IT BACK TO DISK AND RESET THE WRITE INDICATOR */
        IF ZWRSW = 1 THEN DO;
              ZLINE = ZABLK;
              CALL IWRITE(ZBUFF,ZMOD,ZLINE,ZINDX);
              ZWRSW = 0;
        END;
        /* NOW GET THE REQUIRED BLOCK */
        ZLINE = BLOCK;
        CALL IWREAD(ZBUFF,ZMOD,ZLINE,ZINDX);
        ZABLK = BLOCK;
GETB1:  GO TO LABWRT;
        END ZDWRT;
```

```
£COPY *SOURCE* *SINK*
        /* IXDMCV - MOVE THE SUB-FILE POINTER */
ZDMCV:  PROC(NCHAR);
        /* THIS ROUTINE MOVES THE OFFSET POINTER WITHIN A SUB-FILE.
           ONLY FORWARD MOVEMENT IS PERMITTED. IF AN OFFSET BEYOND
           THE SUB-FILE BOUNDARY IS REQUESTED, NEW BLOCKS ARE CHAINED
           ON AS REQUIRED */
        /* ALL CALLS TO ZDMOV ARE GENERATED BY ZDREA AND ZDWRT. THESE
           CALLS ARE ONLY MADE WHEN A MOVE IS REQUIRED OUTSIDE A
           BLOCK */
        /* FSELDC MOVES ARE GENERATED IN OTHER ROUTINES BY MOVING THE
           OFFSET FORWARD AS REQUIRED */
        %INCLUDE IXMSPM;
        /* DEFINE A LABEL RETURN VARIABLE */
        DCL LABMCV LABEL;
        /* SET UP NUMBER OF CHARACTERS OFFSET IS TO BE MOVED */
        NWORK = NCHAR;
        /* START A LOOP TO GET FORWARD BLOCKS, BUT FIRST CHECK TO
           SEE IF MORE BLOCKS MUST BE ADDED */
MOV1:   M = NWORK - DASIZE + OFFST;
        NWBLK = 1 + ((M - 1) / DASIZE);
        /* DECREMENT NUMBER OF CHARACTERS OFFSET IS TO BE MOVED */
        NWORK = NWORK - DASIZE + OFFST - 1;
        OFFST = 1;
        IF NAVAL = -1 THEN GO TO MOV3;
        /* GET NEXT BLOCK INTO CORE AND CHECK FOR END OF LOOP */
MOV2:   BLOCK = NAVAL;
        LABMOV = LABMCV1;
        GO TO GETB;
LABMCV1:NWBLK = NWBLK - 1;
        IF NWBLK = 0 THEN GO TO MCV4;
        GO TO MCV1;
        /* NEED MORE BLOCKS - CALL OVER SUB-FILE ROUTINE TO CREATE
           THEM */
MOV3:   NBLKS = NWBLK;
        CALL ZDCSF(NBLKS);
        /* WE NOW HAVE THE EXTRA BLOCKS. CONTINUE PROCESSING ON THE
           PRESENT BLOCK */
        LABMOV = LABMCV2;
        GO TO GETB;
LABMCV2:GO TO MCV2;
        /* WE ARE NOW AT THE BLOCK WHERE THE OFFSET POINTER IS TO BE
           POSITIONED. LOAD THE NEW OFFSET */
MOV4:   OFFST = OFFST + NWORK;
        /* END OF MOVE ROUTINE - EXIT */
        RETURN;
        /* GETB - IN-LINE CODING FOR GETTING A BLOCK INTO CORE */
        /* FIRST CHECK TO SEE IF THE BLOCK REQUIRED IS IN CORE */
GETB:   IF ZABLK = BLOCK THEN GO TO GETB1;
        /* IF THE PREVIOUS BLOCK IN THE I/O AREA HAS BEEN UPDATED,
           WRITE IT BACK TO DISK AND RESET THE WRITE INDICATOR */
        IF ZWRSW = 1 THEN DO;
              ZLINE = ZABLK;
              CALL IHERITE(ZBUFF,ZMOD,ZLINE,ZINDX);
        ZWRSW = 0;
        END;
        /* NOW GET THE REQUIRED BLOCK */
        ZLINE = BLOCK;
        CALL IHEREAD(ZBUFF,ZMOD,ZLINE,ZINDX);
```

```
        ZABLK = BLOCK;
GETB1:  GO TO LABMOV;
        END ZDMOV;
```

```
£COPY *SOURCE* *SINK*
        /* IXDOSF - ADD N BLOCKS TO A SUB-FILE */
ZDOSF:  PROC(NBLKS);
        /* THIS ROUTINE IS CALLED WHEN EXTRA BLOCKS ARE TO BE ADDED
           TO A SUB-FILE */
        %INCLUDE IXMSPM;
        /* DEFINE A LABEL RETURN VARIABLE */
        DCL LABOSF LABEL;
        /* STORE THE CURRENT BLOCK POSITION IN A WORK AREA */
        DCL WKADDR1 BIN FIXED(31,0);
        WKADDR1 = BLOCK;
        /* LOCATE FIRST AVAILABLE STORAGE SPACE. IF NONE IS AVAILABLE,
           PRINT A MESSAGE AND STOP THE RUN */
        IF FAVAL = -1 THEN GO TO OSF1;
                      ELSE GO TO OSF15;
OSF1:       PUT FILE(REPORT) SKIP(2) EDIT
                ('NO BLOCKS AVAILABLE - JOB TERMINATED')(A);
            CALL ZDCLS;
        /* SET UP FORWARD POINTER IN ORIGINAL LAST BLOCK */
OSF15:  BLOCK = WKADDR1;
        LABOSF = LABOSF1;
        GO TO GETB;
LABOSF1:NAVAL = FAVAL;
        ZWRSW = 1;
        BLOCK = FAVAL;
        /* START A LOOP TO CREATE THE NEW BLOCKS */
OSF2:   LABOSF = LABOSF2;
        GO TO GETB;
        /* PRINT A MESSAGE INDICATING BLOCK HAS BEEN ADDED */
LABOSF2:PUT FILE(REPORT) SKIP EDIT
        ('***** OVER SUB-FILE PROCESSING ADDS BLOCK ',BLOCK)(A,F(6));
        /* DECREMENT THE NUMBER OF AVAILABLE BLOCKS BY 1 */
        NABLKS = NABLKS - 1;
        /* SET UP POINTERS */
        FAVAL = NAVAL;
        ZWRSW = 1;
        NBLKS = NBLKS - 1;
        /* END OF LOOP */
        IF NBLKS = 0 THEN;
                     ELSE DO;
        /* IF THERE IS NO AVAILABLE STORAGE, PRINT A MESSAGE */
        IF FAVAL = -1 THEN GO TO OSF1;
        BLOCK = FAVAL;
        GO TO OSF2;
        END;
        /* RESTORE CURRENT BLOCK POSITION FROM WORK AREA */
        BLOCK = WKADDR1;
        /* SET FORWARD POINTER TO NULL AND EXIT */
        NAVAL = -1;
        RETURN;
        /* GETB - IN-LINE CODING TO GET A BLOCK INTO CORE */
        /* FIRST CHECK TO SEE IF BLOCK REQUIRED IS IN CORE */
GETB:   IF ZABLK = BLOCK THEN GO TO GETB1;
        /* IF THE PREVIOUS BLOCK IN THE I/O AREA HAS BEEN UPDATED,
           WRITE IT BACK TO DISK AND RESET THE WRITE INDICATOR */
        IF ZWRSW = 1 THEN DO;
            ZLINE = ZABLK;
            CALL IWRITE(ZBUFF,ZMOD,ZLINE,ZINDX);
        ZWRSW = 0;
```

```
          END;
          /* NOW GET THE REQUIRED BLOCK */
          ZLINE = BLOCK;
          CALL IFREAD(ZBUFF,ZMOD,ZLINE,ZINDX);
          ZABLK = BLOCK;
GETB1:    GO TO LABOSF;
          END ZDOSF;
```

```
£COPY *SOURCE* *SINK*
        /* IXMDPM - DEFINE STORAGE FOR DIRECTORY CONTROL RECORD */
        DCL ZDAREA CHAR(10) EXT,
            ZDPTR PTR EXT;
        DCL 1 ZDCTRL BASED(ZDPTR),
              2 (DINUM,DLNUM,DLPTR,FILEN,FLDLN) BIN FIXED;
        /* DEFINE STORAGE FOR CURRENT FILE NAME DIRECTORY ENTRY */
        DCL 1 FLENT ALIGNED,
              2 FDPTR BIN FIXED(31,0),
              2 FLNUM BIN FIXED,
              2 NOFLDS BIN FIXED,
              2 FLNAM CHAR(20);
        /* DEFINE STORAGE FOR FIELD NAME DIRECTORY ENTRIES */
        DCL 1 FDENT ALIGNED,
              2 FDIXPTR BIN FIXED(31,0),
              2 FDNUM BIN FIXED,
              2 FDTYP BIN FIXED,
              2 FDLEN BIN FIXED,
              2 FDSTAT BIN FIXED,
              2 FDNAM CHAR(20);
```

```
£COPY *SOURCE* *SINK*
        /* IXPFND - FIND A FILE NAME ON THE DIRECTORY */
ZPFND:  PROC(FILENAME);
        /* THIS ROUTINE DETERMINES WHETHER A FILE NAME ENTERED IS
           ON THE DIRECTORY. IF A MATCH IS FOUND, THE DIRECTORY
           ENTRY NUMBER AND POSITION ARE RETURNED. OTHERWISE ZERO
           AND THE POSITION AT THE END OF THE DIRECTORY ARE RETURNED */
        %INCLUDE IXMSPM;
        %INCLUDE IXMDPM;
     /* DEFINE TEMPORARY STORAGE FOR BLOCK AND OFFSET */
        DCL IBLOCK BIN FIXED(31,0),
            IOFFST BIN FIXED;
        /* DEFINE ZPFND ENTRY AND RETURN PARAMETERS */
        DCL ZPFND ENTRY RETURNS (BIN FIXED(15,0));
        /* DEFINE PASSED FILE NAME AND RETURN NUMBER */
        DCL FILENAME CHAR(20),
            RETNO BIN FIXED(15,0);
        /* START A LOOP TO PROCESS THE DIRECTORY ENTRIES. IT IS
           ASSUMED THAT THE POSITION INDICATOR IS SET TO THE START
           OF THE DIRECTORY */
        DO I = 1 TO DINUM;
             /* STORE POSITION BEFORE READING */
             IBLOCK = BLOCK;
             IOFFST = OFFST;
             /* GET A DIRECTORY ENTRY */
             CALL ZDREA(FILEN,FLENT);
             /* CHECK FOR A DELETED ENTRY */
             IF I = FLNUM THEN;
                          ELSE GO TO FND1;
             /* CHECK FOR A MATCH ON FILE NAMES */
             IF FILENAME = FLNAM THEN;
                          ELSE GO TO FND1;
             /* WE HAVE A MATCH - SET POSITION AND NUMBER AND RETURN
                TO CALLING ROUTINE */
             BLOCK = IBLOCK;
             OFFST = IOFFST;
             RETNO = FLNUM;
             RETURN(RETNO);
        /* END OF DIRECTORY PROCESSING LOOP */
FND1:   END;
        /* NO MATCH FOUND - RETURN ZERO TO CALLING ROUTINE */
        RETNO = 0;
        RETURN(RETNO);
        /* END OF ROUTINE */
        END ZPFND;
```

```
$COPY *SOURCE* *SINK*
        /* IXPNEW - ADD A FILE DESCRIPTION TO THE INVERTED INDEX */
ZPNEW:  PROC OPTIONS(MAIN);
        /* THIS PROGRAM LOADS A NEW FILE NAME WITH ASSOCIATED
           FIELD NAMES AND DESCRIPTIONS ONTO THE DATA DESCRIPTION
           DIRECTORY */
        /* DEFINE INPUT AND OUTPUT FILES */
        DCL INFILE FILE INPUT,
            REPORT FILE OUTPUT;
        %INCLUDE IXMSPM;
        %INCLUDE IXMDPM;
        /* DEFINE TEMPORARY STORAGE FOR BLOCK AND OFFSET */
        DCL IBLOCK BIN FIXED(31,0),
            IOFFST BIN FIXED;
        DCL JBLOCK BIN FIXED(31,0);
        /* DEFINE ROUTINE FOR SEARCHING DIRECTORY */
        DCL ZPFND ENTRY RETURNS (BIN FIXED(15,0));
        /* DEFINE INPUT FILE NAME AND NO. OF FIELDS STORAGE */
        DCL FILENAME CHAR(20),
            NFIELDS BIN FIXED;
        /* DEFINE FIELD TYPE CHARACTER STRING */
        DCL FIELDTYPE CHAR(20) VAR;
        /* END OF INPUT PROCESSING */
        ON ENDFILE(INFILE) GO TO NEW3;
        /* CONVERSION ERROR PROCESSING */
        ON CONVERSION BEGIN;
              PUT FILE(REPORT) SKIP(2) EDIT
                    ('CONVERSION ERROR IN INPUT DATA')(A);
              REVERT CONVERSION;
              IERROR = 1;
        END;
        /* PRINT STARTUP MESSAGE */
        PUT FILE(REPORT) SKIP(2) EDIT
              ('ADD A NEW FILE DESCRIPTION')(A);
        PUT FILE(REPORT) SKIP EDIT
              ('==========================')(A);
        /* BRING CONTROL BLOCKS INTO STORAGE */
        CALL ZOOPN;
        /* SET ZPOSN TO POINT TO START OF INDEX AND ERROR SWITCH TO
           ZERO */
NEW1:   BLOCK = FIXPTR;
        OFFST = 1;
        IERROR = 0;
        /* GET THE FILE NAME AND NUMBER OF FIELDS */
        GET FILE(INFILE) LIST(FILENAME,NFIELDS);
        /* DISPLAY INPUT DATA */
        PUT FILE(REPORT) SKIP(2) EDIT
              ('FILE NAME ',FILENAME,' ENTERED')(A,A(20),A);
        PUT FILE(REPORT) SKIP EDIT
              ('NUMBER OF FIELDS = ',NFIELDS)(A,F(6));
        /* SEARCH DIRECTORY FOR FILE NAME. IF VALUE RETURNED BY ZPFND
           IS NOT ZERO THEN A FILE OF THE SAME NAME ALREADY EXISTS -
           WRITE ERROR MESSAGE AND EXIT */
        IF ZPFND(FILENAME) = 0 THEN;
                               ELSE DO;
              PUT FILE(REPORT) SKIP(2) EDIT
('FILE NAME ',FILENAME,' ALREADY EXISTS IN THE DIRECTORY')(A,A(20),A);
              IERROR = 1;
        END;
```

```
          /* STORE END OF DIRECTORY POSITION IF REQUIRED */
          IBLOCK = BLOCK;
          IOFFST = OFFST;
          /* ALLOCATE DISK SPACE TO STORE FIELDS AND STORE START BLOCK */
          CALL ZOORE;
          JBLOCK = BLOCK;
          /* START A LOOP TO READ IN THE FIELD NAME DESCRIPTIONS
             AND STORE THEM */
          DO I = 1 TO NFIELDS;
               GET FILE(INFILE) LIST(FDNAM,FDTYP,FDLEN);
               /* DISPLAY FIELD NAME */
               PUT FILE(REPORT) SKIP(2) EDIT
('FIELD NAME ',FDNAM,', NUMBERED ',I,' ENTERED')(A,A(20),A,F(6));
               /* SET ERROR CHECK CONDITION TO ZERO */
               ICOND = 0;
               /* VERIFY TYPE 1 ENTRY - CHARACTER */
               IF FDTYP = 1 THEN DO;
                         FIELDTYPE = 'CHARACTER STRING';
                         IF FDLEN < 1 | FDLEN > 256 THEN ICOND=1;
                         GO TO NEW1;
               END;
               /* VERIFY TYPE 2 ENTRY - BINARY FIXED(15,0) */
               IF FDTYP = 2 THEN DO;
                         FIELDTYPE = 'BINARY FIXED(15,0)';
                         IF FDLEN = 2 THEN;
                                      ELSE ICOND = 1;
                         GO TO NEW1;
               END;
               /* VERIFY TYPE 3 ENTRY - BINARY FIXED(31,0) */
               IF FDTYP = 3 THEN DO;
                         FIELDTYPE = 'BINARY FIXED(31,0)';
                         IF FDLEN = 4 THEN;
                                           ELSE ICOND = 1;
                         GO TO NEW1;
               END;
               /* VERIFY TYPE 4 ENTRY - BINARY FLOAT(21) */
               IF FDTYP = 4 THEN DO;
                         FIELDTYPE = 'BINARY FLOAT(21)';
                         IF FDLEN = 4 THEN;
                                         ELSE ICOND = 1;
                         GO TO NEW1;
               END;
               /* VERIFY TYPE 5 ENTRY - BINARY FLOAT (53) */
               IF FDTYP = 5 THEN DO;
                         FIELDTYPE = 'BINARY FLOAT(53)';
                         IF FDLEN = 8 THEN;
                                         ELSE ICOND = 1;
                         GO TO NEW1;
               END;
               /* INVALID FIELD TYPE ENTERED - PRINT MESSAGE AND
                  SET OVERALL ERROR CONDITION */
               PUT FILE(REPORT) SKIP EDIT
                         ('INVALID FIELD TYPE NO. ',FDTYP,' ENTERED')
                         (A,F(2),A);
               IERROR = 1;
               GO TO NEW2;
               /* DISPLAY FIELD TYPE AND LENGTH ENTERED */
NEW1:          PUT FILE(REPORT) SKIP EDIT
('FIELD TYPE NO. ',FDTYP,' ENTERED, ',FIELDTYPE,' LENGTH ',FDLEN)
               (A,F(2),A,A,A,F(3));
```

```
                /* IF INCORRECT LENGTH ENTERED, PRINT A MESSAGE AND
                   SET OVERALL ERROR CONDITION */
                IF ICOND = 1 THEN DO;
                          PUT FILE(REPORT) SKIP EDIT
                          ('INCORRECT FIELD LENGTH ENTERED')(A);
                          IERROR = 1;
                END;
                /* OTHERWISE STORE REMAINING FIELD DATA */
                ELSE DO;
                          FDNUM = I; /* FIELD NUMBER */
                          FDIXPTR = -1; /* NULL INDEX POINTER */
                          FDSTAT = 0; /* RUN STATISTICS ZERO */
                          CALL ZDWRT(FLDLN,FDENT);
                END;
          /* END OF FIELD NAME PROCESSING LOOP */
NEW2:     END;
          /* CHECK FOR ERRORS IN LOADING FILE NAME AND FIELD DATA */
          IF IERROR = 1 THEN DO;
                /* PRINT ERROR MESSAGE AND EXIT */
                PUT FILE(REPORT) SKIP(2) EDIT
('ERRORS DETECTED IN INPUT DATA - FILE DESCRIPTION REJECTED')(A);
                /* DESTROY FIELD NAME SUBFILE */
                BLOCK = JBLOCK;
                CALL ZDDST;
                GO TO NEW0;
          END;
          /* IF THERE ARE NO ENTRIES, ADD ENTRY AT END OF DIRECTORY */
          IF DLPTR = 0 THEN DO;
                /* ZPOSN IS SET TO END OF DIRECTORY */
                FLNUM = DINUM + 1;
                /* INCREMENT NUMBER OF ENTRIES BY 1 */
                DINUM = DINUM + 1;
                /* SET ZPOSN TO POINT TO END OF DIRECTORY */
                BLOCK = IBLOCK;
                OFFST = IOFFST;
          END;
          /* THERE IS A SPACE FROM A PREVIOUS DELETION */
                     ELSE DO;
                /* RESET ZPOSN TO START OF DIRECTORY */
                BLOCK = FIXPTR;
                OFFST = 1;
                /* RESET ZPOSN TO POINT TO DELETED ENTRY */
                OFFST = OFFST + (DLPTR - 1) * FILEN;
                /* GET DELETED BLOCK INTO CORE TO UPDATE DELETION
                   CHAIN */
                IBLOCK = BLOCK;
                IOFFST = OFFST;
                CALL ZDREA(FILEN,FLENT);
                /* UPDATE POINTERS */
                IDUMMY = FLNUM;
                FLNUM = DLPTR;
                DLPTR = IDUMMY;
                /* RESTORE BLOCK AND OFFSET POSITION */
                BLOCK = IBLOCK;
                OFFST = IOFFST;
                /* DECREMENT NUMBER OF DELETED ENTRIES BY 1 */
                DLNUM = DLNUM - 1;
          END;
          /* SET UP NUMBER OF FIELDS TO BE ENTERED */
          NOFLDS = NFIELDS;
```

```
         /* LOAD FILE NAME AND FIELD DATA POINTER INTO FILE NAME
            DIRECTORY ENTRY */
         FLNAM = FILENAME;
         FDPTR = JBLOCK;
         /* WRITE FILE NAME ENTRY TO DIRECTORY */
         CALL ZDWRT(FILEN,FLENT);
         /* BRANCH TO PROCESS ANOTHER FILE DESCRIPTION */
         GO TO NEW3;
         /* END OF INPUT PROCESSING - PRINT A MESSAGE AND EXIT */
NEW3:    PUT FILE(REPORT) SKIP(2) EDIT('END OF RUN')(A);
         CALL ZDCLS;
         RETURN;
         END ZPNEW;
```

```
£COPY *SOURCE* *SINK*
       /* IXPDEL - DELETE A FILE DESCRIPTION FROM THE INVERTED INDEX */
ZPDEL:  PROC OPTIONS(MAIN);
        /* THIS PROGRAM DELETES A FILE NAME ENTRY IN THE DATA
           DESCRIPTION DIRECTORY TOGETHER WITH FIELD DESCRIPTION
           AND INDEXING DATA */
        /* DEFINE INPUT AND OUTPUT FILES */
        DCL INFILE FILE INPUT,
            REPORT FILE OUTPUT;
        %INCLUDE IXMSPM;
        %INCLUDE IXMDPM;
        /* DEFINE ROUTINE FOR SEARCHING DIRECTORY */
        DCL ZPFND ENTRY RETURNS (BIN FIXED(15,0));
        /* DEFINE INPUT FILE NAME STORAGE */
        DCL FILENAME CHAR(2));
        /* DEFINE TEMPORARY STORAGE FOR BLOCK AND OFFSET */
        DCL (IBLOCK,JBLOCK) BIN FIXED(31,0),
            (IOFFST,JOFFST) BIN FIXED;
        /* TEMPORARY DATA AREAS FOR INDEX DELETION */
        DCL DISPAREA CHAR(FDLEN) CTL;
        DCL FOUR BIN FIXED INIT(4),
            FOURCHAR CHAR(4) BASED(FOURPTR),
            FOURPTR PTR,
            KBLOCK BIN FIXED(31,0);
        /* END OF INPUT PROCESSING */
        ON ENDFILE(INFILE) GO TO DELO;
        /* PRINT STARTUP MESSAGE */
        PUT FILE(REPORT) SKIP(2) EDIT
              ('DELETE A FILE DESCRIPTION')(A);
        PUT FILE(REPORT) SKIP EDIT
              ('==========================')(A);
        /* GET CONTROL BLOCKS INTO STORAGE */
        CALL ZDOPN;
        /* SET ZPOSN TO POINT TO START OF INDEX */
DELE:   BLOCK = FIXPTR;
        OFFST = 1;
        /* GET THE FILE NAME TO BE DELETED */
        GET FILE(INFILE) LIST(FILENAME);
        /* DISPLAY INPUT DATA */
        PUT FILE(REPORT) SKIP(2) EDIT
              ('FILE NAME ',FILENAME,' TO BE DELETED')(A,A(20),A);
        /* SEARCH DIRECTORY FOR FILE NAME. IF VALUE RETURNED BY
           ZPFND IS ZERO, THEN FILE NAME ENTERED DOES NOT EXIST ON
           THE DIRECTORY - PRINT AN ERROR MESSAGE AND BRANCH TO
           PROCESS THE NEXT DELETION REQUEST */
        IF ZPFND(FILENAME) = 0 THEN DO;
              PUT FILE(REPORT) SKIP(2) EDIT
                    ('FILE ',FILENAME,' IS NOT IN THE DIRECTORY')
                    (A,A(20),A);
              GO TO DELE;
        END;
        /* ZPOSN POINTS TO THE APPROPRIATE FILE NAME ENTRY. GET IT
           INTO STORAGE */
        IBLOCK = BLOCK;
        IOFFST = OFFST;
        CALL ZDREA(FILEN,FLENT);
        /* SET ZPOSN TO POINT TO START OF FIELD NAME ENTRIES */
        JBLOCK = FDPTR;
        JOFFST = 1;
```

```
        /* START A LOOP TO EXAMINE EACH OF THE FIELD SPECIFICATIONS.
           IF THE FIELD IS INDEXED, THE INDEX ENTRIES ARE DELETED */
        DO I = 1 TO NOFLDS;
        /* GET A FIELD NAME ENTRY */
        BLOCK = JBLOCK;
        OFFST = JOFFST;
        CALL ZDREA(FLDLN,FDENT);
        JBLOCK = BLOCK;
        JOFFST = OFFST;
        IF FDIXPTR = -1 THEN DO;
              PUT FILE(REPORT) SKIP(2) EDIT
('DELETE FIELD NAME ',FDNAM,', - NOT INDEXED')(A,A(20),A);
        END;
                              ELSE DO;
              PUT FILE(REPORT) SKIP(2) EDIT
('DELETE FIELD NAME ',FDNAM,', - INDEXED')(A,A(20),A);
              /* GET START ADDRESS OF VALUE AND ADDRESS SUB-FILE AND
                 DELETE IT */
              BLOCK = FDIXPTR;
              OFFST = 3;
        ALLOCATE DISPAREA;
        CALL ZDREA(FDLEN,DISPAREA);
        FREE DISPAREA;
              FOURPTR = ADDR(KBLOCK);
              CALL ZDREA(FOUR,FOURCHAR);
              BLOCK = KBLOCK;
              CALL ZDDST;
              /* DELETE INDEX CONTROL SUB-FILE AND SET INDEX POINTER
                 TO NULL */
              BLOCK = FDIXPTR;
              CALL ZDDST;
              FDIXPTR = -1;
        END;
        /* END OF FIELD PROCESSING LOOP */
        END;
        /* DELETE FIELD SPECIFICATIONS */
        BLOCK = FDPTR;
        CALL ZDDST;
        FDPTR = -1;
        /* RETURN FILE STORAGE TO DIRECTORY AVAILABILITY AND UPDATE
           THE CONTROL INFORMATION */
        FLNAM = ' ';
        IWORK = FLNUM;
        FLNUM = DLPTR;
        DLPTR = IWORK;
        DLNUM = DLNUM + 1;
        NOFLDS = 0;
        /* WRITE THE REVISED ENTRY WITH CONTROL INFORMATION BACK TO
           DISK */
        BLOCK = IBLOCK;
        OFFST = IOFFST;
        CALL ZDWRT(FILEN,FLENT);
        /* BRANCH TO PROCESS NEXT DELETION REQUEST */
        GO TO DELE;
        /* END OF PROGRAM - PRINT A MESSAGE AND EXIT */
DELC:   PUT FILE(REPORT) SKIP(2) EDIT('END OF RUN')(A);
        CALL ZDCLS;
        RETURN;
        END ZPDEL;
```

```
£COPY *SOURCE* *SINK*
        /* IXUDRP - DISPLAY CONTENTS OF DATA DESCRIPTION DIRECTORY */
ZUDRP:  PROC OPTIONS(MAIN);
        /* THIS PROGRAM PRINTS OUT A LIST OF FILE NAMES TOGETHER
           WITH ASSOCIATED FIELD NAMES IN THE DIRECTORY. FIELD
           PROPERTIES ARE ALSO DISPLAYED */
        /* DEFINE PRINT OUTPUT FILE */
        DCL REPORT FILE OUTPUT;
        %INCLUDE IXMSPM;
        %INCLUDE IXMDPM;
        /* DEFINE TEMPORARY STORAGE FOR BLOCK AND OFFSET */
        DCL IBLOCK BIN FIXED(31,0),
            IOFFST BIN FIXED;
        /* DEFINE OUTPUT MESSAGE AREAS */
        DCL (FIELDTYPE,INDEXMSG) CHAR(20) VAR;
        /* OPEN THE INDEX FILE */
        CALL ZDOPN;
        /* DISPLAY REPORT HEADING */
        PUT FILE(REPORT) SKIP(2) EDIT
              ('DISPLAY OF DATA DESCRIPTION DIRECTORY')(A);
        PUT FILE(REPORT) SKIP EDIT
              ('====================================')(A);
        /* DISPLAY DIRECTORY STATUS INFORMATION */
        PUT FILE(REPORT) SKIP(2) EDIT
              ('DIRECTORY CONTROL INFORMATION')(A);
        PUT FILE(REPORT) SKIP EDIT
              ('-----------------------------')(A);
        IWORK = DINUM - DLNUM;
        PUT FILE(REPORT) SKIP(2) EDIT
              ('NUMBER OF FILE NAME ENTRIES = ',IWORK)(A,F(6));
        PUT FILE(REPORT) SKIP(2) EDIT
              ('NUMBER OF DELETED ENTRIES = ',DLNUM)(A,F(6));
        /* CHECK FOR NO DIRECTORY ENTRIES */
        IF DINUM - DLNUM = 0 THEN GO TO DRP2;
        /* START A LOOP (I) TO PROCESS FILE INFORMATION */
        DO I = 1 TO DINUM;
        /* READ IN A DIRECTORY ENTRY AND STORE POSITION DATA */
        CALL ZDREA(FILEN,FLENT);
        IBLOCK = BLOCK;
        IOFFST = OFFST;
        /* CHECK FOR A DELETED ENTRY */
        IF I = FLNUM THEN;
                     ELSE GO TO DRP1;
        /* WE HAVE A DIRECTORY ENTRY. DISPLAY FILE NAME INFORMATION */
        PUT FILE(REPORT) SKIP(2) EDIT
              ('FILE NAME - ',FLNAM)(A,A(20));
        PUT FILE(REPORT) SKIP EDIT
              ('-------------------------------')(A);
        PUT FILE(REPORT) SKIP(2) EDIT
('FILE NUMBER ',FLNUM,', NUMBER OF FIELDS = ',NOFLDS)(A,F(6),A,F(6));
        /* SET ZPOSN TO START OF FIELD DESCRIPTIONS */
        BLOCK = FDPTR;
        OFFST = 1;
        /* START A LOOP (J) TO PROCESS FIELD INFORMATION */
        DO J = 1 TO NOFLDS;
        /* GET A FIELD ENTRY */
        CALL ZDREA(FLDLN,FDENT);
        /* SET UP FIELD TYPE AND INDEXING INFORMATION */
        IF FDTYP = 1 THEN FIELDTYPE = 'CHARACTER STRING';
```

```
        IF FDTYP = 2 THEN FIELDTYPE = 'BINARY FIXED(15,0)';
        IF FDTYP = 3 THEN FIELDTYPE = 'BINARY FIXED(31,0)';
        IF FDTYP = 4 THEN FIELDTYPE = 'BINARY FLOAT(21)';
        IF FDTYP = 5 THEN FIELDTYPE = 'BINARY FLOAT(53)';
        IF FDIXPTR = -1 THEN INDEXMSG = 'NOT INDEXED';
                        ELSE INDEXMSG = 'INDEXED';
        /* DISPLAY FIELD NAME INFORMATION */
        PUT FILE(REPORT) SKIP(2) EDIT
              ('FIELD NAME ',FDNAM,', NUMBER = ',FDNUM)
              (A,A(20),A,F(6));
        PUT FILE(REPORT) SKIP EDIT
              ('FIELD TYPE ',FIELDTYPE,', LENGTH = ',FDLEN)
              (A,A,A,F(6));
        PUT FILE(REPORT) SKIP EDIT
        ('FIELD STATUS ',INDEXMSG,', TIMES ACCESSED = ',FDSTAT)
              (A,A,A,F(6));
        /* END OF FIELD PROCESSING (J) LOOP */
        END;
        /* RESET POSITION AND LOOP BACK TO GET ANOTHER FILE NAME
           ENTRY */
        BLOCK = IBLOCK;
        OFFST = IOFFST;
        /* END OF FILE NAME PROCESSING (I) LOOP */
DRP1:   END;
        /* PRINT END OF RUN MESSAGE, CLOSE INDEX FILE AND EXIT */
DRP2:   PUT FILE(REPORT) SKIP(2) EDIT('END OF RUN')(A);
        CALL ZDCLS;
        RETURN;
        END ZUDRP;
```

```
£COPY *SOURCE* *SINK*
        /* IXPIXC - CREATE AN INVERTED INDEX */
ZPIXC:  PROC OPTIONS(MAIN);
        /* THIS PROGRAM CREATES AN INVERTED INDEX FROM THE VALUES
          CONTAINED WITHIN A SPECIFIED FIELD NAMED WITHIN A FILE */
        /* DEFINE INPUT AND OUTPUT FILES */
        DCL INFILE FILE INPUT,
            REPORT FILE OUTPUT;
        %INCLUDE IXMSPM;
        %INCLUDE IXMDPM;
        /* CPU TIME PROCESSING AREAS */
        DCL TIME ENTRY,
            P3 FIXED BIN(31) INIT(3) STATIC,
            CKEY FIXED BIN(31) INIT(1),
            CPR FIXED BIN(31) INIT(0),
            CCRES BIN FLOAT(21),
            CRES BIN FIXED(31),
            CRESA(5) FIXED BIN(31);
        /* DEFINE ROUTINE FOR SEARCHING DIRECTORY */
        DCL ZPFND ENTRY RETURNS (BIN FIXED(15,0));
        /* DEFINE INPUT FILE AND FIELD NAME STORAGE */
        DCL (FILENAME,FIELDNAME) CHAR(20);
        /* DEFINE A PROCESSING OPTION AREA */
        DCL POPT CHAR(3) VAR;
        /* DEFINE WORK AND STORAGE AREAS FOR STAGE 1 OF PROCESSING */
        DCL (IRLC,IFSP,IMSW,IFLC) BIN FIXED EXT,
            DFDNAM CHAR(20),
            IFDTYP BIN FIXED,
           IBLOCK BIN FIXED(31,0) EXT,
           IOFFST BIN FIXED EXT;
        /* DEFINE DATA AND SORT INPUT FILES */
        DCL ZDATA FILE RECORD SEQUENTIAL INPUT ENV(U(255)),
            ZISRT FILE RECORD SEQUENTIAL ENV(U(255));
        /* DEFINE RECORD PROCESSING AREAS */
        /* ***** MTS DEPENDENT ***** */
        DCL (RECAREA,ABSAREA) CHAR(255) VAR;
        DCL ABSMASK CHAR(8000) BASED(ABSPTR),
            ABSPTR PTR;
        DCL 1 ABSREC CTL ALIGNED,
              2 RECNO BIN FIXED(31,0),
              2 IXFLD CHAR(IFLC);
        /* DEFINE RECORD COUNTER */
        DCL RECCTR BIN FIXED(31,0) INIT(0);
        /* DEFINE INDEX SIZE CALCULATION VARIABLES */
        DCL (RECVA,RECIC,RECTOT) BIN FIXED;
        /* DEFINE BLOCK ESTIMATE COUNTER */
        DCL ESTBLOCK BIN FIXED INIT(0);
        /* SORT PARAMETERS */
        DCL F1 FIXED BIN(31) INIT(1) STATIC,
            SORT ENTRY,
            PLIRC RETURNS (FIXED BIN(31));
        DCL 1 CSA STATIC,
              2 A1 CHAR(2) INIT('S='),
              2 A2 CHAR(2),
              2 A3 CHAR(4) INIT(',,5,'),
              2 A4 PIC '999',
              2 A5 CHAR(21) INIT(',BI,,1,4 I=-A@¬TRIM,,'),
              2 A6 PIC '999',
              2 A7 CHAR(13) INIT(' O=-B@¬TRIM,,'),
```

```
      2 A8 PIC '999',
      2 A9 CHAR(3) INIT(' R='),
      2 A10 PIC '(6)9',
      2 A11 CHAR(5) INIT(',END ');
/* DEFINE SORT OUTPUT FILE */
DCL ZOSRT FILE RECORD SEQUENTIAL ENV(U(255));
/* DEFINE WORK AND STORAGE AREAS FOR STAGE 4 PROCESSING */
DCL DISPAREA CHAR(FDLEN) CTL;
DCL (JBLOCK,KBLOCK) BIN FIXED(31,0),
    (JOFFST,KOFFST) BIN FIXED;
DCL FOUR BIN FIXED INIT(4);
DCL TWO BIN FIXED INIT(2);
DCL FOURCHAR CHAR(4) BASED(FOURPTR),
    TWOCHAR CHAR(2) BASED(TWOPTR),
    (FOURPTR,TWOPTR) PTR;
DCL (NVAE,FSWCH,SPCTR) BIN FIXED;
/* DEFINE PREVIOUS FIELD VALUE STORAGE */
DCL PREVAREA CHAR(IFLC) CTL;
/* DEFINE WORKING STORAGE FOR NO. OF RECORD ADDRESSES
   POSITION AND NUMBER */
DCL HBLOCK BIN FIXED(31,0),
    HOFFST BIN FIXED,
    NRECA BIN FIXED(31,0);
/* SAVE AREA FOR CURRENT FIELD VALUE AND BLOCK ADDRESS */
DCL SAVEAREA CHAR(IFLC) CTL,
    SAVEBLOCK BIN FIXED(31,0),
    SAVEOFFST BIN FIXED;
/* INDEX BLOCK PROCESSING SWITCH */
DCL IXSWCH BIN FIXED;
/* VALUE AND ADDRESS BLOCK PROCESSING AREAS */
DCL LMOVE BIN FIXED,
    LABIX LABEL;
/* DEFINE FIELD VALUE COUNTER */
DCL FVCTR BIN FIXED(31,0) INIT(0);
/* END OF INPUT FILE PROCESSING */
ON ENDFILE(INFILE) GO TO IXC9;
/* ************************************************************ */
/* THIS PROGRAM DIVIDES ITSELF INTO FOUR STAGES AS FOLLOWS: */
/*    1. PROCESS DATA DESCRIPTION DIRECTORY TO OBTAIN          */
/*       CONTROL INFORMATION.                                  */
/*    2. ABSTRACT VALUES AND RECORD ADDRESSES FROM THE         */
/*       DATA FILE TO A TEMPORARY FILE.                        */
/*    3. SORT TEMPORARY FILE RECORDS BY ASCENDING VALUE        */
/*       AND RECORD ADDRESS AND STORE ON ANOTHER TEMPORARY     */
/*       FILE.                                                 */
/*    4. LOAD INDEX INFORMATION ONTO THE INVERTED INDEX.       */
/* ************************************************************ */
/* INITIALIZE RUN STATISTICS FIELDS */
NVAE = 0;
DO I = 1 TO 5;
    CRESA(I) = 0;
END;
/* GET AND STORE CURRENT CPU TIME STATUS */
CALL PLCALL(TIME,P3,ADDR(CKEY),ADDR(CPR),ADDR(CRES));
CRESA(1) = CRES;
/* ***** START STAGE 1 - PROCESS DIRECTORY INFORMATION ***** */
/* PRINT STARTUP MESSAGE */
PUT FILE(REPORT) SKIP(2) EDIT
    ('CREATE AN INVERTED INDEX')(A);
PUT FILE(REPORT) SKIP EDIT
```

```
        ('======================')(A);
/* BRING CONTROL BLOCKS INTO STORAGE */
CALL ZOOPN;
/* GET FILE NAME AND PROCESSING OPTION */
GET FILE(INFILE) LIST(FILENAME,POPT);
PUT FILE(REPORT) SKIP(2) EDIT
      ('PROCESSING FILE ',FILENAME)(A,A(20));
/* SEARCH DIRECTORY FOR FILE NAME. IF VALUE RETURNED BY
   ZPFND IS ZERO, THEN NO MATCH HAS BEEN FOUND - PRINT AN
   ERROR MESSAGE AND EXIT */
IF ZPFND(FILENAME) = 0 THEN DO;
      PUT FILE(REPORT) SKIP(2) EDIT
            ('FILE ',FILENAME,'IS NOT IN THE DIRECTORY')
            (A,A(20),A);
      CALL ZOCLS;
END;
/* ZPOSN POINTS TO THE APPROPRIATE FILE NAME ENTRY. GET IT
   INTO STORAGE */
CALL ZOREA(FILEN,FLENT);
/* GET THE FIELD NAME */
IXCO: GET FILE(INFILE) LIST(FIELDNAME);
/* DISPLAY INPUT FIELD NAME */
PUT FILE(REPORT) SKIP(2) EDIT
      ('INDEX PROCESSING ON FIELD ',FIELDNAME,' REQUESTED')
      (A,A(20),A);
/* SET ZPOSN TO POINT TO THE START OF THE FIELD NAME ENTRIES */
BLOCK = FDPTR;
OFFST = 1;
/* SET FILE RECORD LENGTH COUNTER TO ZERO, FIELD START
   POSITION TO 1, AND MATCH SWITCH TO ZERO */
IRLO = 0;
IFSP = 1;
IMSW = 0;
/* START A LOOP TO PROCESS FIELD DESCRIPTIONS */
DO I = 1 TO NOFLDS;
/* STORE FIELD NAME ENTRY POINTER IF MATCH SWITCH IS ZERO */
IF IMSW = 0 THEN DO;
      IBLOCK = BLOCK;
      IOFFST = OFFST;
END;
/* GET A FIELD NAME ENTRY */
CALL ZOREA(FLOLN,FDENT);
/* INCREMENT RECORD LENGTH COUNTER BY FIELD LENGTH */
IRLO = IRLO + FDLEN;
/* CHECK FOR MATCH ON FIELD NAME */
IF FDNAM = FIELDNAME THEN;
                     ELSE GO TO IXO1;
/* SET MATCH SWITCH TO 1 */
IMSW = 1;
/* STORE FIELD LENGTH */
IFLO = FDLEN;
/* STORE NAME AND TYPE OF FIELD BEING INDEXED */
IFDNAM = FDNAM;
IFDTYP = FDTYP;
/* IF THE FIELD IS INDEXED PRINT A WARNING MESSAGE */
IF FDIXPTR = -1 THEN;
                ELSE DO;
      PUT FILE(REPORT) SKIP(2) EDIT
('WARNING - FIELD NAME ENTERED IS ALREADY INDEXED')(A);
      /* TEST FOR CANCEL OPTION. IF 'YES', NO FURTHER
```

```
                PROCESSING */
              IF POPT = 'YES' THEN DO;
                   PUT FILE(REPORT) SKIP(2) EDIT
              ('CANCEL OPTION INVOKED - NO FURTHER PROCESSING')(A);
                   GO TO IXC9;
              END;
         END;
         /* IF A MATCH HAS NOT BEEN MADE, INCREMENT FIELD START
            POSITION BY FIELD LENGTH JUST PROCESSED */
IXC1:    IF IMSW = 0 THEN IFSP = IFSP + FDLEN;
         /* END OF FIELD NAME PROCESSING LOOP */
         END;
         /* CHECK FOR A MATCH ON FIELD NAME ENTERED. IF NOT, PRINT
            ERROR MESSAGE AND EXIT */
        IF IMSW = 0 THEN DO;
              PUT FILE(REPORT) SKIP(2) EDIT
('FIELD NAME ',FIELDNAME,' DOES NOT EXIST IN FILE ',FILENAME)
                   (A,A(20),A,A(20));
              IF POPT = 'EST' THEN GO TO IXC0;
                              ELSE GO TO IXC9;
         END;
         /* DISPLAY STATUS INFORMATION OBTAINED FROM DIRECTORY SEARCH */
         PUT FILE(REPORT) SKIP(2) EDIT
              ('FILE ',FLNAM,' RECORD LENGTH =',IRLC)(A,A(20),A,F(6));
         PUT FILE(REPORT) SKIP(2) EDIT
('FIELD ',DFDNAM,' LENGTH = ',IFLC,', START POSITION = ',IFSP)
              (A,A(20),A,F(6),A,F(6));
         /* ***** END STAGE 1 - PROCESS DIRECTORY INFORMATION ***** */
         /* GET AND STORE CURRENT CPU TIME STATUS */
         CALL PLCALL(TIME,P3,ADDR(CKEY),ADDR(CPR),ADDR(CRES));
         CRESA(2) = CRES;
         /* ***** START STAGE 2 - ABSTRACT INDEXING INFORMATION ***** */
         /* BRANCH ROUND READING AND ABSTRACTING INDEXING INFORMATION
            IF ESTIMATE OPTION HAS BEEN INVOKED AND DATA FILE ALREADY
            READ */
         IF POPT = 'EST' & RECCTR ¬= 0 THEN GO TO IXC35;
         /* SET UP END OF DATA FILE PROCESSING */
         ON ENDFILE(ZDATA) GO TO IXC3;
         /* ALLOCATE STORAGE FOR ABSTRACT PROCESSING AREA */
         ALLOCATE ABSREC;
         ABSPTR = ADDR(ABSREC);
         /* SET RECORD COUNTER TO 1 */
         RECCTR = 1;
         /* OPEN FILES TO BE PROCESSED */
         OPEN FILE(ZDATA);
         OPEN FILE(ZISRT) OUTPUT;
         /* GET A RECORD */
IXC2:    READ FILE(ZDATA) INTO(RECAREA);
         /* BRANCH ROUND ABSTRACTING INDEXING INFORMATION IF
            ESTIMATE OPTION HAS BEEN INVOKED */
         IF POPT = 'EST' THEN GO TO IXC25;
         /* ABSTRACT INDEXING DATA AND SET UP RECORD ADDRESS */
         /* ********** NOTE - COULD BE MADE MORE EFFICIENT */
         IXFLD = SUBSTR(RECAREA,IFSP,IFLC);
         RECNO = RECCTR;
         /* WRITE INDEXING RECORD TO SORT INPUT FILE */
         ABSAREA = SUBSTR(ABSMASK,1,IFLC+4);
         WRITE FILE(ZISRT) FROM(ABSAREA);
         /* INCREMENT RECORD COUNTER AND BRANCH TO PROCESS ANOTHER
            RECORD */
```

```
IXC25:  RECCTR = RECCTR + 1;
        GO TO IXC2;
        /* CLOSE FILES */
IXC3:   CLOSE FILE(ZDATA);
        CLOSE FILE(ZISRT);
        /* DECREMENT RECCTR BY 1 */
        RECCTR = RECCTR - 1;
        /* DISPLAY NO. OF RECORDS */
IXC35:  PUT FILE(REPORT) SKIP(2) EDIT
              ('NUMBER OF INDEXING RECORDS = ',RECCTR)(A,F(6));
        /* CALCULATE MAXIMUM NO. OF VALUE & ADDRESS BLOCK REQUIRED */
        RECVA = 1 + ((RECCTR * (IFLC + 8)) / DASIZE);
        /* CALCULATE MAXIMUM NO. OF INDEX CONTROL BLOCKS REQUIRED */
        RECIC = 1 + ((2 + (RECVA * (IFLC + 6))) / DASIZE);
        /* CALCULATE TOTAL NO. OF BLOCKS REQUIRED AND DISPLAY */
        RECTOT = RECVA + RECIC;
        PUT FILE(REPORT) SKIP(2) EDIT
              ('MAXIMUM NO. OF BLOCKS REQUIRED = ',RECTOT)(A,F(6));
        /* TEST FOR ESTIMATE OPTION. IF 'EST', NO FURTHER PROCESSING */
        IF POPT = 'EST' THEN DO;
              PUT FILE(REPORT) SKIP(2) EDIT
              ('ESTIMATE OPTION INVOKED - NO FURTHER PROCESSING')(A);
        /* INCREMENT NO. OF ESTIMATED BLOCKS */
              ESTBLOCK = ESTBLOCK + RECTOT;
              GO TO IXCO;
        END;
        /* IF FIELD IS NOT INDEXED, CHECK TO SEE IF THE CALCULATED
           MAXIMUM INDEX SIZE EXCEEDS AVAILABLE SPACE ON THE INDEX
           FILE. IF SO BRANCH TO PRINT A CANCEL MESSAGE */
        IF FDIXPTR = -1 & RECTOT > NABLKS THEN GO TO IXC7;
        /* ***** END STAGE 2 - ABSTRACT INDEXING INFORMATION ***** */
        /* GET AND STORE CURRENT CPU TIME STATUS */
        CALL PLCALL(TIME,P3,ADDR(CKEY),ADDR(CPR),ADDR(CRES));
        CRESA(3) = CRES;
        /* ***** START STAGE 3 - SORT INDEXING INFORMATION ***** */
        /* SET UP VARIABLE VALUES IN SORT CONTROL STATEMENT */
        /* DETERMINE DATA TYPE OF MAJOR SORT FIELD */
        IF IFDTYP = 2 | IFDTYP = 3 THEN DO; /* BINARY */
              CSA.A2 = 'BI';
              GO TO IXC4;
        END;
        IF IFDTYP = 4 | IFDTYP = 5 THEN DO; /* FLOATING POINT */
              CSA.A2 = 'FL';
              GO TO IXC4;
        END;
        CSA.A2 = 'CH'; /* CHARACTER */
        /* SET LENGTH OF MAJOR INDEXING FIELD */
IXC4:   CSA.A4 = IFLC;
        /* SET RECORD LENGTH TO BE SORTED - INPUT AND OUTPUT */
        CSA.A6 = IFLC + 4;
        CSA.A8 = IFLC + 4;
        /* SET NUMBER OF RECORDS TO BE SORTED */
        CSA.A10 = RECCTR;
        /* CALL SORT AND CHECK FOR ERRORS */
        CALL PLCALL(SORT,FI,CSA);
        IF PL1RC = 0 THEN;
                     ELSE DO;
              PUT FILE(REPORT) SKIP(2) EDIT
        ('ERROR DETECTED IN SORT - NO FURTHER PROCESSING')(A);
              GO TO IXC9;
```

```
END;
/* ***** END STAGE 3 - SORT INDEXING INFORMATION ***** */
/* GET AND STORE CURRENT CPU TIME STATUS */
CALL PLCALL(TIME,P3,ADDR(CKEY),ADDR(CPR),ADDR(CRES));
CRESA(4) = CRES;
/* ***** START STAGE 4 - CREATE INVERTED INDEX ***** */
/* GET FIELD NAME INFORMATION FOR PROCESSING */
BLOCK = IBLOCK;
OFFST = IOFFST;
CALL ZDREA(FLDLN,FDENT);
/* IF FIELD IS ALREADY INDEXED, DELETE THE PREVIOUS INDEX */
IF FDIXPTR = -1 THEN;
                ELSE DO;
      /* GET START ADDRESS OF VALUE AND ADDRESS SUB-FILE
         AND DELETE IT */
      BLOCK = FDIXPTR;
      OFFST = 3;
      ALLOCATE DISPAREA;
      CALL ZDREA(FDLEN,DISPAREA);
      FREE DISPAREA;
      FOURPTR = ADDR(KBLOCK);
      CALL ZDREA(FOUR,FOURCHAR);
      BLOCK = KBLOCK;
      CALL ZDDST;
      /* DELETE INDEX CONTROL SUB-FILE AND SET INDEX POINTER
         TO NULL */
      BLOCK = FDIXPTR;
      CALL ZDDST;
      FDIXPTR = -1;
      /* CHECK TO SEE IF THE MAXIMUM SPACE REQUIRED EXCEEDS
         THE SPACE AVAILABLE ON THE INDEX FILE. IF SO BRANCH
         TO PRINT AN ERROR MESSAGE */
      IF RECTOT > NABLKS THEN GO TO IXC7;
END;
/* CREATE A NEW INDEX CONTROL SUB-FILE */
CALL ZDCRE;
FDIXPTR = BLOCK;
JBLOCK = BLOCK;
JOFFST = 3;
/* CREATE A NEW VALUE AND ADDRESS SUB-FILE */
CALL ZDCRE;
KBLOCK = BLOCK;
KOFFST = 1;
/* SET UP CONTROL INFORMATION FOR CREATING INDEX */
NVAB = 0; /* NO. OF VALUE AND ADDRESS BLOCK ENTRIES */
FSWCH = 0; /* FIRST TIME SWITCH */
SPCTR = 1; /* SPACE COUNTER ON VALUE AND ADDRESS BLOCKS */
/* ALLOCATE A WORK AREA FOR PREVIOUS FIELD VALUE */
ALLOCATE PREVAREA;
/* OPEN SORTED ABSTRACT FILE FOR PROCESSING */
OPEN FILE(ZOSRT) INPUT;
/* ALLOCATE A SAVE AREA FOR CURRENT FIELD VALUE */
ALLOCATE SAVEAREA;
/* START A LOOP TO READ IN INDEXING INFORMATION */
DO I = 1 TO RECCTR;
/* GET A RECORD INTO THE PROCESSING AREA */
READ FILE(ZOSRT) INTO(ABSAREA);
SUBSTR(ABSMASK,1,IFLC+4) = SUBSTR(ABSAREA,1,IFLC+4);
/* IF FIRST TIME THROUGH, SET FIRST TIME SWITCH AND BRANCH
   ROUND CHANGE OF FIELD VALUE PROCESSING */
```

```
          IF FSWCH = 0 THEN DO;
                FSWCH = 1;
                GO TO IXC5;
          END;
          /* CHECK FOR CHANGE IN FIELD VALUE */
          IF IXFLD = PREVAREA THEN GO TO IXC6;
          /* STORE NUMBER OF RECORD ADDRESSES FOR PREVIOUS FIELD
             VALUE */
          BLOCK = WBLOCK;
          OFFST = WOFFST;
          FOURPTR = ADDR(NRECA);
          CALL ZDWRT(FOUR,FOURCHAR);
          /* STORE FIELD VALUE AND CURRENT BLOCK ADDRESS ON VALUE
             AND ADDRESS SUB-FILE IN SAVE AREA */
IXC5:     SAVEAREA = IXFLD;
          SAVEBLOCK = KBLOCK;
          SAVEOFFST = KOFFST;
          /* SET INDEX CONTROL SWITCH TO ZERO */
          IXSWCH = 0;
          /* STORE FIELD VALUE ON VALUE AND ADDRESS BLOCK */
          BLOCK = KBLOCK;
          OFFST = KOFFST;
          CALL ZDWRT(IFLD,IXFLD);
          KBLOCK = BLOCK;
          KOFFST = OFFST;
          /* INCREMENT FIELD VALUE COUNTER */
          FVCTR = FVCTR + 1;
          /* CHECK FOR VALUE AND ADDRESS BLOCK OVERFLOW */
          LMOVE = IFLD;
          LABIX = LABIX1;
          GO TO OFLO;
          /* RESERVE SPACE FOR NO. OF RECORD ADDRESSES ON VALUE AND
             ADDRESS BLOCK, SET INITIAL NO. TO ZERO, AND STORE WRITE
             POSITION */
LABIX1:   BLOCK = KBLOCK;
          OFFST = KOFFST;
          WBLOCK = BLOCK;
          WOFFST = OFFST;
          NRECA = 0;
          FOURPTR = ADDR(NRECA);
          CALL ZDWRT(FOUR,FOURCHAR);
          KBLOCK = BLOCK;
          KOFFST = OFFST;
          /* CHECK FOR VALUE AND ADDRESS BLOCK OVERFLOW */
          LMOVE = 4;
          LABIX = LABIX2;
          GO TO OFLO;
          /* MOVE CONTENTS OF PRESENT FIELD TO PREVIOUS FIELD AREA */
LABIX2:   PREVAREA = IXFLD;
          /* WRITE RECORD ADDRESS TO VALUE AND ADDRESS BLOCK */
IXC6:     BLOCK = KBLOCK;
          OFFST = KOFFST;
          FOURPTR = ADDR(RECNO);
          CALL ZDWRT(FOUR,FOURCHAR);
          KBLOCK = BLOCK;
          KOFFST = OFFST;
          /* CHECK FOR VALUE AND ADDRESS BLOCK OVERFLOW */
          LMOVE = 4;
          LABIX = LABIX3;
          GO TO OFLO;
```

```
        /* INCREMENT NO. OF RECORD ADDRESSES BY 1 */
LABIX3: NRECA = NRECA + 1;
        /* END OF INDEX PROCESSING LOOP */
        END;
        /* STORE NUMBER OF RECORD ADDRESSES FOR LAST FIELD VALUE */
        BLOCK = WBLOCK;
        OFFST = WOFFST;
        FOURPTR = ADDR(NRECA);
        CALL ZDWRT(FOUR,FOURCHAR);
        /* DETERMINE IF LAST RECORD ON INDEX IS LAST RECORD ON A
           BLOCK. IF NOT, A FINAL INDEX CONTROL SUB-FILE RECORD
           MUST BE WRITTEN OUT (UNLESS IT IS ALREADY THERE) */
        IF IXSWCH = 0 THEN DO;
              BLOCK = JBLOCK;
              OFFST = JOFFST;
              CALL ZDWRT(IFLG,SAVEAREA);
              FOURPTR = ADDR(SAVEBLOCK);
              CALL ZDWRT(FOUR,FOURCHAR);
              TWOPTR = ADDR(SAVEOFFST);
              CALL ZDWRT(TWO,TWOCHAR);
              NVAB = NVAB + 1;
              /* SET INDEX CONTROL SWITCH BACK TO 1 */
              IXSWCH = 1;
        END;
        /* STORE NUMBER OF VALUE AND ADDRESS BLOCKS ON INDEX CONTROL
           SUB-FILE */
        BLOCK = FDIXPTR;
        OFFST = 1;
        TWOPTR = ADDR(NVAB);
        CALL ZDWRT(TWO,TWOCHAR);
        FREE PREVAREA,SAVEAREA;
        /* STORE FIELD ENTRY INFORMATION */
        GO TO IXC8;
        /* NO SPACE AVAILABLE ERROR PROCESSING - PRINT A MESSAGE AND
           EXIT */
IXC7:   PUT FILE(REPORT) SKIP(2) EDIT
('MAXIMUM SPACE REQUESTED FOR INDEX EXCEEDS AVAILABLE SPACE ON FILE')
              (A);
IXC8:   BLOCK = IBLOCK;
        OFFST = IOFFST;
        CALL ZDWRT(FLDLN,FDENT);
        FREE ABSREC;
        /* IF ESTIMATED NUMBER OF BLOCKS IS GREATER THAN ZERO,
           THEN DISPLAY */
IXC9: IF ESTBLOCK > 0 THEN DO;
        PUT FILE(REPORT) SKIP(3) EDIT
              ('TOTAL NO. OF ESTIMATED BLOCKS = ',ESTBLOCK)(A,F(6));
        END;
        /* ***** END STAGE 4 - CREATE INVERTED INDEX ***** */
        /* GET AND STORE CURRENT CPU TIME STATUS */
        CALL PLCALL(TIME,P3,ADDR(CKEY),ADDR(CPR),ADDR(CRES));
        CRESA(5) = CRES;
        /* CALCULATE AND DISPLAY CPU TIME USED IN THE VARIOUS STAGES
           OF CREATING THE INDEX. BY-PASS THIS SECTION IF ESTIMATE
           OPTION SPECIFIED */
        IF FOPT = 'EST' THEN;
                        ELSE DO;
        PUT FILE(REPORT) SKIP(2) EDIT
              ('INDEX CREATION STATISTICS')(A);
        PUT FILE(REPORT) SKIP EDIT
```

```
          ('-------------------------')(A);
          PUT FILE(REPORT) SKIP(2) EDIT
                ('NO. OF FIELD VALUES ENTERED =  ',FVCTR)(A,F(6));
          PUT FILE(REPORT) SKIP(2) EDIT
                ('NO. OF INDEX CONTROL ENTRIES = ',NVAB)(A,F(6));
          CORES = CRESA(2) - CRESA(1);
          CORES = CORES / 1000.0;
          PUT FILE(REPORT) SKIP(2) EDIT
                ('EDIT CPU TIME(SECONDS) =      ',CORES)(A,F(10,3));
          CORES = CRESA(3) - CRESA(2);
          CORES = CORES / 1000.0;
          PUT FILE(REPORT) SKIP(2) EDIT
                ('ABSTRACT CPU TIME(SECONDS) = ',CORES)(A,F(10,3));
          CORES = CRESA(4) - CRESA(3);
          CORES = CORES / 1000.0;
          PUT FILE(REPORT) SKIP(2) EDIT
                ('SORT CPU TIME(SECONDS) =      ',CORES)(A,F(10,3));
          CORES = CRESA(5) - CRESA(4);
          CORES = CORES / 1000.0;
          PUT FILE(REPORT) SKIP(2) EDIT
                ('CREATE CPU TIME(SECONDS) =    ',CORES)(A,F(10,3));
          END;
          CORES = CRESA(5) - CRESA(1);
          CORES = CORES / 1000.0;
          PUT FILE(REPORT) SKIP(2) EDIT
                ('TOTAL CPU TIME(SECONDS) =     ',CORES)(A,F(10,3));
          /* END OF PROCESSING - PRINT A MESSAGE AND EXIT */
          PUT FILE(REPORT) SKIP(2) EDIT('END OF RUN')(A);
          CALL ZDCLS;
          RETURN;
          /* OFLO - IN-LINE CODING FOR DETERMINING THE HIGHEST
             FIELD VALUE ON A BLOCK AND INCREMENTING THE NUMBER
             OF VALUE AND ADDRESS BLOCKS */
          /* INCREMENT SPACE COUNTER BY LENGTH OF DATA BEING WRITTEN */
OFLO:     SPCTR = SPCTR + LMOVE;
          /* DETERMINE IF THE SPACE COUNTER VALUE IS GREATER THAN
             THE DATA AREA SIZE (OVERFLOW) */
          IF SPCTR > DASIZE THEN;
                              ELSE GO TO OFLO2;
          /* IF THE INDEX CONTROL INFORMATION HAS NOT ALREADY BEEN
             WRITTEN, WRITE IT TO THE INDEX CONTROL SUB-FILE */
          IF IXSWCH = 1 THEN GO TO OFLO1;
          /* WRITE BLOCK ADDRESS AND FIELD VALUE FROM SAVE AREA TO
             INDEX CONTROL BLOCK */
          BLOCK = JBLOCK;
          OFFST = JOFFST;
          CALL ZDWRT(IFLO,SAVEAREA);
          FOURPTR = ADDR(SAVEBLOCK);
          CALL ZDWRT(IFOUR,FOURCHAR);
          TWOPTR = ADDR(SAVEOFFST);
          CALL ZDWRT(ITWO,TWOCHAR);
          JBLOCK = BLOCK;
          JOFFST = OFFST;
          /* INCREMENT NO. OF VALUE ADDRESS BLOCKS BY 1 */
          NVAB = NVAB + 1;
          /* SET INDEX CONTROL SWITCH TO 1 */
          IXSWCH = 1;
          /* SET SPACE COUNTER TO RELEVANT POSITION IN NEW BLOCK */
OFLO1:    SPCTR = SPCTR - DASIZE;
          /* END OF OVERFLOW PROCESSING */
```

```
OFLC2:  GO TO LABIX;
        END ZPIXC;
```

```
£CCPY *SOURCE* *SINK*
        /* IXPIXD - DELETE A SPECIFIED INDEX */
ZPIXD:  PROC OPTIONS(MAIN);
        /* THIS PROGRAM DELETES AN INDEX FOR A SPECIFIED FIELD VALUE */
        /* DEFINE INPUT AND OUTPUT FILES */
        DCL  INFILE FILE INPUT,
             REPORT FILE OUTPUT;
        %INCLUDE IXMSPM;
        %INCLUDE IXMDPM;
        /* DEFINE INPUT FILE AND FIELD NAMES */
        DCL (FILENAME,FIELDNAME) CHAR(20);
        /* DEFINE ROUTINE FOR SEARCHING DIRECTORY */
        DCL ZPFND ENTRY RETURNS(BIN FIXED(15,0));
        /* DEFINE PROCESSING WORK AREAS */
        DCL DISPAREA CHAR(FDLEN) CTL;
        DCL (WBLOCK,DBLOCK) BIN FIXED(31,0),
            WOFFST BIN FIXED,
            FOURCHAR CHAR(4) BASED(FOURPTR),
            FOURPTR PTR,
            FOUR BIN FIXED INIT(4);
        /* PRINT STARTUP MESSAGE */
        PUT FILE(REPORT) SKIP(2) EDIT
              ('DELETE INDEX FROM INVERTED FILE')(A);
        PUT FILE(REPORT) SKIP EDIT
              ('==============================')(A);
        /* BRING CONTROL BLOCKS INTO STORAGE */
        CALL ZDOPN;
        /* GET THE INPUT DATA AND DISPLAY IT */
        GET FILE(INFILE) LIST(FILENAME,FIELDNAME);
        PUT FILE(REPORT) SKIP(2) EDIT
              ('DELETE INDEX ON ',FILENAME,'(',FIELDNAME,')')
              (A,A(20),A,A(20),A);
        /* SEARCH DIRECTORY FOR FILE NAME. IF VALUE RETURNED BY
           ZPFND IS ZERO, THEN NO MATCH HAS BEEN FOUND - PRINT
           AN ERROR MESSAGE AND EXIT */
        IF ZPFND(FILENAME) = 0 THEN DO;
              PUT FILE(REPORT) SKIP(2) EDIT
              ('FILE ',FILENAME,' IS NOT IN THE DIRECTORY')(A,A(20),A);
              CALL ZDCLS;
        END;
        /* ZPOSN POINTS TO THE APPROPRIATE FILE NAME ENTRY. GET IT
           INTO STORAGE */
        CALL ZDREA(FILEN,FLENT);
        /* SET ZPOSN TO POINT AT THE START OF THE FIELD NAME ENTRIES */
        BLOCK = FDPTR;
        OFFST = 1;
        /* START A LOOP TO PROCESS FIELD DESCRIPTIONS */
        DO I = 1 TO NOFLDS;
              /* GET A FIELD NAME ENTRY */
              WBLOCK = BLOCK; WOFFST = OFFST;
              CALL ZDREA(FLDLN,FDENT);
              /* CHECK FOR A MATCH ON FIELD NAME */
              IF FDNAM = FIELDNAME THEN;
                                   ELSE GO TO IXD1;
              /* IF THE FIELD IS NOT INDEXED, CANCEL RUN */
              IF FDIXPTR = -1 THEN DO;
                    PUT FILE(REPORT) SKIP(2) EDIT
                    ('FIELD ',FIELDNAME,' IS NOT INDEXED')(A,A(20),A);
                    CALL ZDCLS;
```

```
            END;
            /* WE  HAVE A VALID INDEXED FIELD. BRANCH ROUND
               ERROR PROCESSING */
            GO TO IXD2;
         /* END OF FIELD NAME PROCESSING LOOP */
IXD1:    END;
         /* NO MATCH ON FIELD NAME. PRINT AN ERROR MESSAGE AND EXIT */
         PUT FILE(REPORT) SKIP(2) EDIT
               ('FIELD NAME ',FIELDNAME,' NOT FOUND IN FILE ',FILENAME)
               (A,A(20),A,A(20));
         CALL ZDCLS;
         /* GET START ADDRESS OF VALUE AND ADDRESS SUB-FILE AND
            DELETE IT */
IXD2:    BLOCK = FDIXPTR;
         OFFST = 3;
         ALLOCATE DISPAREA;
         CALL ZDREA(FDLEN,DISPAREA);
         FREE DISPAREA;
         FOURPTR = ADDR(DBLOCK);
         CALL ZDREA(FOUR,FOURCHAR);
         BLOCK = DBLOCK;
         CALL ZDDST;
         /* DELETE INDEX CONTROL SUBFILE AND SET INDEX POINTER TO
            NULL */
         BLOCK = FDIXPTR;
         CALL ZDDST;
         FDIXPTR = -1;
         /* WRITE BACK UPDATED FIELD NAME ENTRY */
         BLOCK = WBLOCK;
         OFFST = WOFFST;
         CALL ZDWRT(FLDLN,FDENT);
         /* END OF PROGRAM - PRINT A MESSAGE AND EXIT */
         PUT FILE(REPORT) SKIP(2) EDIT('END OF RUN')(A);
         CALL ZDCLS;
         END ZPIXD;
```

```
£COPY *SOURCE* *SINK*
        /* IXPCMP - COMPARE TWO FIELD VALUES AND RETURN A RESULT */
ZPCMP:  PROC(CMPAREA1,CMPAREA2,CMPRET);
        /* THIS ROUTINE COMPARES TWO FIELD VALUES WHICH ARE PASSED
           TO IT. THE TYPE OF COMPARISON IS DETERMINED BY THE FIELD
           TYPE DEFINED THROUGH AN EXTERNAL VARIABLE. A BINARY VALUE
           OF -1, 0 OR +1 IS RETURNED ACCORDING TO THE COMPARISON
           GIVING LESS THAN, EQUAL OR GREATER THAN RESULTS. */
        /* DEFINE COMPARISON COUNT FIELD */
        DCL ZCCMP BIN FIXED(31,0) EXT INIT(0);
        /* DEFINE FIELD TYPE INDICATOR */
        DCL ZFTYP BIN FIXED EXT;
        /* DEFINE COMPARE AREAS FOR VARIOUS FIELD TYPES */
        DCL CMPAREA1 CHAR(*) CTL,
            CMPPTR1 PTR,
            CMPTYP12 BIN FIXED(15,0) UNALIGNED BASED(CMPPTR1),
            CMPTYP13 BIN FIXED(31,0) UNALIGNED BASED(CMPPTR1),
            CMPTYP14 BIN FLOAT(21) UNALIGNED BASED(CMPPTR1),
            CMPTYP15 BIN FLOAT(53) UNALIGNED BASED(CMPPTR1);
        DCL CMPAREA2 CHAR(*) CTL,
            CMPPTR2 PTR,
            CMPTYP22 BIN FIXED(15,0) UNALIGNED BASED(CMPPTR2),
            CMPTYP23 BIN FIXED(31,0) UNALIGNED BASED(CMPPTR2),
            CMPTYP24 BIN FLOAT(21) UNALIGNED BASED(CMPPTR2),
            CMPTYP25 BIN FLOAT(53) UNALIGNED BASED(CMPPTR2);
        /* DEFINE RETURN VALUE */
        DCL CMPRET BIN FIXED;
        /* INCREMENT NO. OF COMPARISONS BY 1 */
        ZCCMP = ZCCMP + 1;
        /* MAKE VARIOUS FIELD TYPES ADDRESSABLE */
        CMPPTR1 = ADDR(CMPAREA1);
        CMPPTR2 = ADDR(CMPAREA2);
        /* DETERMINE FIELD TYPE AND BRANCH ACCORDINGLY */
        IF ZFTYP = 2 THEN GO TO CMP1;
        IF ZFTYP = 3 THEN GO TO CMP2;
        IF ZFTYP = 4 THEN GO TO CMP3;
        IF ZFTYP = 5 THEN GO TO CMP4;
        /* CHARACTER COMPARISON */
        IF CMPAREA1 < CMPAREA2 THEN GO TO CMP5;
        IF CMPAREA1 = CMPAREA2 THEN GO TO CMP6;
        GO TO CMP7;
        /* BINARY FIXED(15,0) COMPARISON */
CMP1:   IF CMPTYP12 < CMPTYP22 THEN GO TO CMP5;
        IF CMPTYP12 = CMPTYP22 THEN GO TO CMP6;
        GO TO CMP7;
        /* BINARY FIXED(31,0) COMPARISON */
CMP2:   IF CMPTYP13 < CMPTYP23 THEN GO TO CMP5;
        IF CMPTYP13 = CMPTYP23 THEN GO TO CMP6;
        GO TO CMP7;
        /* BINARY FLOAT(21) COMPARISON */
CMP3:   IF CMPTYP14 < CMPTYP24 THEN GO TO CMP5;
        IF CMPTYP14 = CMPTYP24 THEN GO TO CMP6;
        GO TO CMP7;
        /* BINARY FLOAT(53) COMPARISON */
CMP4:   IF CMPTYP15 < CMPTYP25 THEN GO TO CMP5;
        IF CMPTYP15 = CMPTYP25 THEN GO TO CMP6;
        GO TO CMP7;
        /* LESS THAN CONDITION */
CMP5:   CMPRET = -1;
```

```
        RETURN;
        /* EQUALS CONDITION */
CMP6:   CMPRET = 0;
        RETURN;
        /* CREATER THAN CONDITION */
CMP7:   CMPRET = 1;
        RETURN;
        /* END OF COMPARISON ROUTINE */
        END ZPCMP;
```

```
£CCPY *SCURCE* *SINK*
        /* IXPEDT - EDIT SELECT INPUT DATA */
ZPEDT:  PRCC(FILENAME,FIELDNAME,OPERATCR,FIELDVALUE);
        /* THIS ROUTINE ACCEPTS AS INPUT ONE SELECT CONDITICN,
           DETERMINES IF IT IS VALID, AND IF SC STCRES THE RELEVANT
           PROCESSING INSTRUCTIONS CN A TABLE */
        /* DEFINE CUTPUT FILE */
        DCL REPORT FILE OUTPUT;
        %INCLUDE IXMSPM;
        %INCLUDE IXMCPM;
        /* DEFINE INPUT FILE NAME, FIELD NAME, SELECTICN CPERATOR
           AND FIELD VALUE PASSED AS PARAMETERS */
        DCL (FILENAME,FIELDNAME) CHAR(20),
            OPERATCR CHAR(3) VAR,
            FIELDVALLE CHAR(255) VAR;
        /* DEFINE FIRST TIME SWITCH FCR EDITING FILE NAME */
        DCL ZEDSW BIN FIXED EXT INIT(0);
        /* DEFINE ROUTINE FOR SEARCHING DIRECTCRY */
        DCL ZPFND ENTRY RETURNS (BIN FIXED(15,0));
        /* DEFINE EDIT ERRCR FLAG FCR RETURN */
        DCL ZEDTF BIN FIXED EXT INIT(0);
        /* DEFINE TABLE AND NC. OF ENTRIES FCR STCRING SELECT
           CONDITICNS ENTERED */
        DCL 1 ZSTAB(20) EXT, /* MAXIMUM 20 CONDITICNS PER CUERY */
              2 SCNUM BIN FIXED, /* SEARCH CONDITICN NUMBER */
              2 LVNUM BIN FIXED, /* LEVEL NUMBER */
              2 IBADDR BIN FIXED(31,0), /* INDEX BLCCK NUMBER */
              2 STYPE BIN FIXED, /* SEARCH TYPE */
              2 FTYPE BIN FIXED, /* FIELD TYPE */
              2 FVLEN BIN FIXED, /* FIELD VALUE LENGTH */
              2 FVPTR PTR; /* POINTER TO FIELD VALUE STCRAGE AREA */
        DCL ZSTNC BIN FIXED EXT INIT(0); /* NUMBER CF ENTRIES */
        /* DEFINE SEARCH FIELD VALLE AREA */
        DCL SCHPTR PTR,
            SCHTYP1 CHAR(FDLEN) CTL,
            SCHTYP2 BIN FIXED(15,0) UNALIGNED BASED(SCHPTR),
            SCHTYP3 BIN FIXED(31,0) UNALIGNED BASED(SCHPTR),
            SCHTYP4 BIN FLOAT(21) UNALIGNED BASED(SCHPTR),
            SCHTYP5 BIN FLOAT(53) UNALIGNED BASED(SCHPTR);
        /* SEARCH FOR FILE NAME FIRST TIME THRCUGH CNLY */
        IF ZEDSW = 1 THEN GC TC IXE05;
        /* SET EDIT SWITCH TC 1 */
        ZEDSW = 1;
        /* DISPLAY FILE NAME IN PRCCESS */
        PUT FILE(REPCRT) SKIP(2) EDIT
              ('PRCCESSING FILE ',FILENAME)(A,A(20));
        /* SEARCH DIRECTORY FCR FILE NAME. IF VALUE RETURNED BY
           ZPFND IS ZERO, THEN NC MATCH HAS BEEN FCUND - PRINT
           AN ERRCR MESSAGE AND SET ERROR FLAG */
        IF ZPFND(FILENAME) = 0 THEN DC;
              PUT FILE(REPCRT) SKIP(2) EDIT
                    ('FILE ',FILENAME,' IS NCT IN THE DIRECTCRY')
                    (A,A(20),A);
              CALL ZCCLS;
        END;
        /* ZPOSN POINTS TO THE APPRCPRIATE FILE NAME ENTRY. GET IT
           INTC STCRAGE */
        CALL ZCREA(FILEN,FLENT);
        /* DISPLAY INPUT DATA */
```

```
IXEC5:  PUT FILE(REPORT) SKIP(2) EDIT
            ('***** SELECT CONDITION ',ZSTNO)(A,F(2));
        PUT FILE(REPORT) SKIP EDIT
            ('SEARCH INDEX ON (',FIELDNAME,') ',
              OPERATOR,' (',FIELDVALUE,')')
            (A,A(20),A,A,A,A,A);
        /* SET ZPOSN TO POINT AT THE START OF THE FIELD NAME ENTRIES */
        BLOCK = FDPTR;
        OFFST = 1;
        /* START A LOOP TO PROCESS FIELD DESCRIPTIONS */
        DO I = 1 TO NOFLDS;
        /* GET A FIELD NAME ENTRY */
        CALL ZGREA(FLDLN,FDENT);
        /* CHECK FOR A MATCH ON FIELD NAME */
        IF FDNAM = FIELDNAME THEN;
                             ELSE GO TO IXE10;
        /* IF THE FIELD IS NOT INDEXED SET ERROR FLAG */
        IF FDIXPTR = -1 THEN DO;
            PUT FILE(REPORT) SKIP(2) EDIT
('FIELD ',FIELDNAME,' NOT INDEXED - SEQUENTIAL SEARCH REQUIRED')
                (A,A(20),A);
            ZEDTF = 1;
            GO TO IXE50;
        END;
        /* WE HAVE A VALID FIELD NAME. BRANCH ROUND ERROR PROCESSING */
        GO TO IXE20;
        /* END OF FIELD NAME PROCESSING LOOP */
IXE10:  END;
        /* NO MATCH ON FIELD NAME. PRINT AN ERROR MESSAGE AND SET
           ERROR FLAG */
        PUT FILE(REPORT) SKIP(2) EDIT
            ('FIELD NAME ',FIELDNAME,' NOT FOUND IN FILE ',FILENAME)
            (A,A(20),A,A(20));
        ZEDTF = 1;
        GO TO IXE50;
        /* EDIT ENTERED OPERATOR FOR VALIDITY AND ASSIGN SEARCH TYPE */
IXE20:  IF OPERATOR = 'EQ' THEN DO;
            STYPE(ZSTNO) = 1; /* EQUAL */
            GO TO IXE30;
        END;
        IF OPERATOR = 'NEQ' THEN DO;
            STYPE(ZSTNO) = 2; /* NOT EQUAL */
            GO TO IXE30;
        END;
        IF OPERATOR = 'GT' THEN DO;
            STYPE(ZSTNO) = 3; /* GREATER THAN */
            GO TO IXE30;
        END;
        IF OPERATOR = 'GTE' THEN DO;
            STYPE(ZSTNO) = 4; /* GREATER THAN OR EQUAL */
            GO TO IXE30;
        END;
        IF OPERATOR = 'LT' THEN DO;
            STYPE(ZSTNO) = 5; /* LESS THAN */
            GO TO IXE30;
        END;
        IF OPERATOR = 'LTE' THEN DO;
            STYPE(ZSTNO) = 6; /* LESS THAN OR EQUAL */
            GO TO IXE30;
        END;
```

```
        /* INVALID OPERATOR ENTERED - PRINT AN ERROR MESSAGE AND
           SET ERROR FLAG */
        PUT FILE(REPORT) SKIP(2) EDIT
              ('INVALID OPERATOR ',OPERATOR,' ENTERED')(A,A,A);
        ZEDTF = 1;
        GO TO IXE50;
        /* SET UP FIELD TYPE */
IXE30:  FTYPE(ZSTNO) = FDTYP;
        /* DEFINE CONVERSION ERROR PROCESSING FOR INPUT FIELD VALUE */
        ON CONVERSION BEGIN;
              PUT FILE(REPORT) SKIP(2) EDIT
        ('CONVERSION ERROR IN INPUT FIELD VALUE ',FIELDVALUE)(A,A);
              ZEDTF = 1;
              GO TO IXE50;
        END;
        /* ALLOCATE STORAGE FOR FIELD VALUE AND MAKE IT ADDRESSABLE */
        ALLOCATE SCHTYP1;
        SCHPTR = ADDR(SCHTYP1);
        FVPTR(ZSTNO) = SCHPTR;
        /* CONVERT INPUT FIELD VALUE (IF NECESSARY) */
        IF FTYPE(ZSTNO) = 2 THEN DO;
              SCHTYP2 = FIELDVALUE; /* BINARY FIXED(15,0) */
              GO TO IXE40;
        END;
        IF FTYPE(ZSTNO) = 3 THEN DO;
              SCHTYP3 = FIELDVALUE; /* BINARY FIXED(31,0) */
              GO TO IXE40;
        END;
        IF FTYPE(ZSTNO) = 4 THEN DO;
              SCHTYP4 = FIELDVALUE; /* BINARY FLOAT(21) */
              GO TO IXE40;
        END;
        IF FTYPE(ZSTNO) = 5 THEN DO;
              SCHTYP5 = FIELDVALUE; /* BINARY FLOAT(53) */
              GO TO IXE40;
        END;
        SCHTYP1 = FIELDVALUE; /* CHAR(*) */
        /* STORE INDEX BLOCK ADDRESS, FIELD VALUE LENGTH AND RETURN */
IXE40:  IBAIDR(ZSTNO) = FDIXPTR;
        FVLEN(ZSTNO) = FDLEN;
IXE50:  RETURN;
        END ZPED1;
```

```
£COPY *SOURCE* *SINK*
        /* IXPSEL - SELECT RECORDS ACCORDING TO INPUT PARAMETERS */
ZPSEL:  PROC(TABNO);
        /* THIS ROUTINE RECEIVES AS A PARAMETER AN ENTRY NUMBER IN
           A TABLE. FROM THIS NUMBER A TABLE OF SELECT CONDITIONS
           IS LOCKED UP AND THE RELEVANT SELECT PROCESSING INVOKED.
           OUTPUT FROM THE ROUTINE IS A NUMBER OF RECORD ADDRESSES
           STORED IN A TEMPORARY FILE ON DISK */
        /* DEFINE OUTPUT FILE */
        DCL REPORT FILE OUTPUT;
        /* DEFINE BLOCK AND OFFSET INDICATOR */
        DCL 1 ZPOSN EXT,
              2 BLOCK BIN FIXED(31,0),
              2 OFFST BIN FIXED;
        /* DEFINE TABLE ENTRY NUMBER PASSED AS PARAMETER */
        DCL TABNO BIN FIXED;
        /* DEFINE TABLE FOR STORING SELECT CONDITIONS ENTERED */
        DCL 1 ZSTAB(20) EXT, /* MAXIMUM 20 CONDITIONS PER QUERY */
              2 SONUM BIN FIXED, /* SEARCH CONDITION NUMBER */
              2 LVNUM BIN FIXED, /* LEVEL NUMBER */
              2 IBADDR BIN FIXED(31,0), /* INDEX BLOCK ADDRESS */
              2 STYPE BIN FIXED, /* SEARCH TYPE */
              2 FTYPE BIN FIXED, /* FIELD TYPE */
              2 FVLEN BIN FIXED, /* FIELD VALUE LENGTH */
              2 FVPTR PTR; /* POINTER TO FIELD VALUE STORAGE AREA */
        /* DEFINE FIELD TYPE AREA */
        DCL ZFTYP BIN FIXED EXT;
        /* DEFINE SEARCH FIELD VALUE AREA */
        DCL SCHPTR PTR,
            MSKAREA CHAR(255) BASED(SCHPTR),
            SCHAREA CHAR(IWLEN) CTL;
        /* DEFINE START AND FINISH FIELD VALUES AND POSITIONS */
        DCL (SVALUE,FVALUE) CHAR(IWLEN) CTL,
            SBLOCK BIN FIXED(31,0);
        /* DEFINE INDEX PROCESSING VARIABLES */
        DCL NVAB BIN FIXED, FWORK BIN FIXED(31,0), HWORK BIN FIXED,
            TWO BIN FIXED INIT(2), FOUR BIN FIXED INIT(4);
        DCL FOURCHAR CHAR(4) BASED(FOURPTR),
            TWOCHAR CHAR(2) BASED(TWOPTR),
            (FOURPTR,TWOPTR) PTR;
        /* DEFINE PREVIOUS FIELD VALUE AND POSITION */
        DCL PVALUE CHAR(IWLEN) CTL,
            PBLOCK BIN FIXED(31,0),
            POFFST BIN FIXED(15,0);
        /* DEFINE CURRENT FIELD VALUE AND POSITION */
        DCL CVALUE CHAR(IWLEN) CTL,
            CBLOCK BIN FIXED(31,0),
            COFFST BIN FIXED(15,0);
        /* DEFINE NO. OF RECORD ADDRESSES FOR ONE FIELD VALUE */
        DCL NRECA BIN FIXED(31,0);
        /* DEFINE TOTAL NO. OF RECORD ADDRESSES FOUND */
        DCL ZTREC BIN FIXED(31,0) EXT;
        /* DEFINE RECORD ADDRESS WORK FILE */
        DCL ZWADD FILE RECORD SEQUENTIAL ENV(F(240,4));
        /* INITIALIZE SEARCH FIELD TYPE FOR ZPCMP */
        ZFTYP = FTYPE(TABNO);
        /* DETERMINE START AND FINISH OF SUB-FILE POSITIONS */
        /* ALLOCATE STORAGE FOR START AND FINISH FIELD VALUES */
        IWLEN = FVLEN(TABNO);
```

```
         ALLOCATE SVALUE,FVALUE;
         /* DETERMINE POSITION OF SEARCH FIELD VALUE */
         SCHPTR = FVPTR(TABNO);
         /* ALLOCATE STORAGE FOR SEARCH AREA AND MOVE IN DATA */
         ALLOCATE SCHAREA;
         SCHAREA = MSKAREA;
         /* DETERMINE NO. OF VALUE AND ADDRESS ENTRIES */
         BLOCK = IBADDR(TABNO);
         OFFST = 1;
         TWOPTR = ADDR(NVAB);
         CALL ZDREA(TWO,TWOCHAR);
         /* LOOP TO PROCESS INDEX VALUES AND STORE HIGHEST VALUE */
         FOLDPTR = ADDR(FWORK);
         TWOPTR = ADDR(HWORK);
         DO I = 1 TO NVAB;
               CALL ZDREA(IWLEN,FVALUE);
               CALL ZDREA(FOUR,FOURCHAR);
               CALL ZDREA(TWO,TWOCHAR);
               /* IF FIRST ENTRY STORE BLOCK ADDRESS */
               IF I = 1 THEN SBLOCK = FWORK;
         END;
         /* GET AND STORE LOWEST VALUE */
         BLOCK = SBLOCK;
         OFFSV = 1;
         CALL ZDREA(IWLEN,SVALUE);
         /* SET NUMBER OF RECORDS FOUND TO ZERO */
         ZTREC = 0;
         /* DETERMINE THE SEARCH TYPE CODE AND BRANCH TO THE
            APPROPRIATE PROCESSING ROUTINE */
         IF STYPE(TABNO) = 1 THEN GO TO IXS100;
         IF STYPE(TABNO) = 2 THEN GO TO IXS200;
         IF STYPE(TABNO) = 3 | STYPE(TABNO) = 4 THEN GO TO IXS300;
         IF STYPE(TABNO) = 5 | STYPE(TABNO) = 6 THEN GO TO IXS500;
         /* THE FOLLOWING STATEMENT SHOULD NEVER BE PROCESSED AS
            THE SEARCH TYPE HAS ALREADY BEEN CHECKED */
         PUT FILE(REPORT) SKIP(2) EDIT('***** SYSTEM ERROR 1')(A);
         CALL ZOOLS;
         /* ***** START OF SELECT PROCESSING TYPE 1 - EQUAL ***** */
         /* DETERMINE IF FIELD VALUE ON WHICH SEARCH IS DEFINED
            IS WITHIN THE RANGE OF FIELD VALUES WITHIN THE INDEX. IF
            NOT PRINT A MESSAGE AND EXIT */
IXS100:  CALL ZPCMP(SCHAREA,SVALUE,HWORK);
         IF HWORK = -1 THEN GO TO IXS105;
         CALL ZPCMP(SCHAREA,FVALUE,HWORK);
         IF HWORK = 1 THEN;
                                        ELSE GO TO IXS110;
IXS105:   PUT FILE(REPORT) SKIP(2) EDIT
               ('***** SELECT CONDITION NO. ',SCNUM(TABNO))(A,F(2));
          PUT FILE(REPORT) SKIP EDIT
               ('FIELD VALUE SPECIFIED IS NOT IN INDEX')(A);
         GO TO IXSRT2;
         /* SET PREVIOUS FIELD INFORMATION TO START OF INDEX */
IXS110:  ALLOCATE PVALUE,CVALUE;
         PVALUE = SVALUE;
         PBLOCK = SBLOCK;
         POFFST = 1;
         /* START A LOOP TO PROCESS THE INDEX CONTROL ENTRIES */
         BLOCK = IBADDR(TABNO);
         OFFST = 3;
         DO I = 1 TO NVAB;
```

```
        /* GET AN INDEX CONTROL ENTRY */
        CALL ZDREA(IWLEN,CVALUE);
        FOURPTR = ADDR(CBLOCK);
        CALL ZDREA(FOUR,FOURCHAR);
        TWOPTR = ADDR(COFFST);
        CALL ZDREA(TWO,TWOCHAR);
        /* COMPARE THE SEARCH VALUE WITH THE CURRENT ENTRY VALUE */
        CALL ZPCMP(SCHAREA,CVALUE,HWORK);
        IF HWORK = 0 THEN GO TO IXS125; /* EQUAL */
        IF HWORK = -1 THEN GO TO IXS115; /* LESS THAN */
        /* MOVE CURRENT ENTRY INFORMATION INTO PREVIOUS */
        PVALUE = CVALUE;
        PBLOCK = CBLOCK;
        POFFST = COFFST;
        /* END OF INDEX CONTROL PROCESSING LOOP */
        END;
        /* THE FOLLOWING STATEMENTS SHOULD NEVER BE PROCESSED */
        PUT FILE(REPORT) SKIP(2) EDIT('***** SYSTEM ERROR 2')(A);
        CALL ZCCLS;
        /* SET ZPOSN TO PREVIOUS */
IXS115: BLOCK = PBLOCK;
        OFFST = POFFST;
        /* GET A VALUE AND ADDRESS ENTRY */
IXS120: CALL ZDREA(IWLEN,CVALUE);
        /* COMPARE THE SEARCH VALUE WITH THE CURRENT ENTRY VALUE */
        CALL ZPCMP(SCHAREA,CVALUE,HWORK);
        IF HWORK = 0 THEN GO TO IXS130; /* EQUAL */
        IF HWORK = -1 THEN GO TO IXS135; /* LESS THAN - NOT FOUND */
        /* SKIP TO NEXT ENTRY */
        FOURPTR = ADDR(NRECA);
        CALL ZDREA(FOUR,FOURCHAR);
        OFFST = OFFST + 4*NRECA;
        GO TO IXS120;
        /* WE HAVE DETECTED AN EQUAL CONDITION IN THE INDEX CONTROL
           ENTRIES. SET THE POSITION INDICATOR TO POINT TO THE
           NUMBER OF RECORDS */
IXS125: BLOCK = CBLOCK;
        OFFST = COFFST + IWLEN;
        /* DETERMINE NO. OF RECORDS IN LIST */
IXS130: FOURPTR = ADDR(NRECA);
        CALL ZDREA(FOUR,FOURCHAR);
        /* SET NUMBER OF RECORD ADDRESSES FOUND */
        ZTREC = NRECA;
        /* START A LOOP TO GET AND STORE RECORD NUMBERS */
        FOURPTR = ADDR(FWORK);
        DO I = 1 TO NRECA;
              CALL ZDREA(FOUR,FOURCHAR);
              /* STORE RECORD NUMBER ON ADDRESS WORK FILE */
              WRITE FILE(ZWADD) FROM(FOURCHAR);
        END;
        FREE SVALUE,FVALUE,PVALUE,CVALUE;
        FREE SCHAREA;
        GO TO IXSRET;
        /* ***** END OF SELECT PROCESSING TYPE 1 - EQUAL ***** */
        /* ***** START OF SELECT PROCESSING TYPE 2 - NOT EQUAL ***** */
        /* SET POSITION INDICATOR TO START OF VALUE AND ADDRESS
           SUB-FILE */
IXS200: BLOCK = SBLOCK;
        OFFST = 1;
        ALLOCATE CVALUE;
```

```
        /* GET A FIELD VALUE */
IXS205: CALL ZDREA(IWLEN,CVALUE);
        /* DETERMINE NUMBER OF RECORDS ON THE LIST */
        FOLRPTR = ADDR(NRECA);
        CALL ZDREA(FOUR,FOURCHAR);
        /* IF SEARCH VALUE EQUALS CURRENT FIELD VALUE BRANCH ROUND
           DISPLAY PROCESSING */
        CALL ZPCMP(SCHAREA,CVALUE,HWORK);
        IF HWORK = 0 THEN DO;
              OFFST = OFFST + 4 * NRECA;
              GO TO IXS210;
        END;
        /* INCREMENT TOTAL NUMBER OF RECORD ADDRESSES FOUND */
        ZTREC = ZTREC + NRECA;
        /* START A LOOP TO GET AND STORE RECORD NUMBERS */
        FOLRPTR = ADDR(FWORK);
        DO I = 1 TO NRECA;
              CALL ZDREA(FOUR,FOURCHAR);
              /* STORE RECORD NUMBER ON ADDRESS WORK FILE */
              WRITE FILE(ZWADD) FROM(FOURCHAR);
        END;
        /* FIND OUT IF CURRENT FIELD VALUE IS HIGHEST ON INDEX. IF
           NOT BRANCH BACK TO GET ANOTHER FIELD VALUE */
IXS210: CALL ZPCMP(CVALUE,FVALUE,HWORK);
        IF HWORK = 0 THEN;
                                    ELSE GO TO IXS205;
        /* FREE PROCESSING AREAS AND BRANCH TO SORT AND DISPLAY
           RECORD NUMBERS */
        FREE SVALUE,FVALUE,CVALUE;
        FREE SCHAREA;
        GO TO IXSRET;
        /* ***** END OF SELECT PROCESSING TYPE 2 - NOT EQUAL ***** */
        /* ***** START OF SELECT PROCESSING TYPES 3 AND 4 -
           GREATER THAN AND GREATER THAN OR EQUAL ***** */
        /* DETERMINE WHETHER THE FIELD VALUE ON WHICH THE SEARCH IS
           DEFINED IS WITHIN THE RANGE OF FIELD VALUES WITHIN THE
           INDEX. IF BELOW, DISPLAY ALL RECORD NUMBERS; BUT IF ABOVE,
           DISPLAY A MESSAGE AND EXIT */
IXS300: CALL ZPCMP(SCHAREA,SVALUE,HWORK);
        ALLOCATE PVALUE,CVALUE;
        IF HWORK = -1 THEN DO;
              BLOCK = SBLOCK;
              OFFST = 1;
              GO TO IXS325;
        END;
        CALL ZPCMP(SCHAREA,FVALUE,HWORK);
        IF HWORK = 1 THEN;
                                    ELSE GO TO IXS305;
        PUT FILE(REPORT) SKIP(2) EDIT
              ('***** SELECT CONDITION NO. ',SCNUM(TABNO))(A,F(2));
        PUT FILE(REPORT) SKIP EDIT
('FIELD VALUE SPECIFIED IS GREATER THAN HIGHEST VALUE ON INDEX')(A);
        GO TO IXSRT2;
        /* SET PREVIOUS FIELD INFORMATION TO START OF INDEX */
IXS305: PVALUE = SVALUE;
        PBLOCK = SBLOCK;
        POFFST = 1;
        /* START A LOOP TO PROCESS THE INDEX CONTROL ENTRIES */
        BLOCK = IBADDR(TABNO);
        OFFST = 3;
```

```
        DO I = 1 TO NVAB;
        /* GET AN INDEX CONTROL RECORD */
        CALL ZDREA(IWLEN,CVALUE);
        FOURPTR = ADDR(CBLOCK);
        CALL ZDREA(FOUR,FOURCHAR);
        TWOPTR = ADDR(COFFST);
        CALL ZDREA(TWO,TWOCHAR);
        /* COMPARE THE SEARCH VALUE WITH THE CURRENT ENTRY VALUE */
        CALL ZPCMP(SCHAREA,CVALUE,HWORK);
        IF HWORK = 0 THEN DO; /* EQUAL */
              BLOCK = CBLOCK;
              OFFST = COFFST;
              CALL ZDREA(IWLEN,CVALUE);
              GO TO IXS320;
        END;
        IF HWORK = -1 THEN GO TO IXS310; /* LESS THAN */
        /* MOVE CURRENT ENTRY INFORMATION INTO PREVIOUS */
        PVALUE = CVALUE;
        PBLOCK = CBLOCK;
        POFFST = COFFST;
        /* END OF INDEX CONTROL PROCESSING LOOP */
        END;
        /* THE FOLLOWING STATEMENTS SHOULD NEVER BE PROCESSED */
        PUT FILE(REPORT) SKIP(2) EDIT('***** SYSTEM ERROR 3')(A);
        CALL ZDCLS;
        /* SET ZPOSN TO PREVIOUS */
IXS310: BLOCK = PBLOCK;
        OFFST = POFFST;
        /* GET A VALUE AND ADDRESS ENTRY */
IXS315: CALL ZDREA(IWLEN,CVALUE);
        /* COMPARE THE SEARCH VALUE WITHIN THE CURRENT ENTRY VALUE */
        CALL ZPCMP(SCHAREA,CVALUE,HWORK);
        IF HWORK = 0 THEN GO TO IXS320; /* EQUAL */
        IF HWORK = -1 THEN GO TO IXS330; /* LESS THAN */
        /* SKIP TO NEXT ENTRY */
        FOURPTR = ADDR(NRECA);
        CALL ZDREA(FOUR,FOURCHAR);
        OFFST = OFFST + 4 * NRECA;
        GO TO IXS315;
        /* DETERMINE IF SEARCH TYPE CODE IS 4 - GREATER THAN EQUAL.
           IF SO SET ZPOSN TO POINT TO IT, OTHERWISE SET ZPOSN TO
           POINT TO NEXT ENTRY */
IXS320: IF STYPE(TABNO) = 4 THEN GO TO IXS330;
        /* FIND OUT IF FIELD VALUE IS HIGHEST IN INDEX. IF SO, BRANCH
           TO END OF SEARCH */
        CALL ZPCMP(CVALUE,FVALUE,HWORK);
        IF HWORK = 0 THEN GO TO IXS335;
        /* SKIP TO NEXT ENTRY */
        FOURPTR = ADDR(NRECA);
        CALL ZDREA(FOUR,FOURCHAR);
        OFFST = OFFST + 4 * NRECA;
        /* GET A FIELD VALUE */
IXS325: CALL ZDREA(IWLEN,CVALUE);
        /* DETERMINE NUMBER OF RECORDS ON THE LIST */
IXS330: FOURPTR = ADDR(NRECA);
        CALL ZDREA(FOUR,FOURCHAR);
        /* INCREMENT TOTAL NUMBER OF RECORD ADDRESSES FOUND */
        ZTREC = ZTREC + NRECA;
        /* START A LOOP TO GET AND STORE RECORD NUMBERS */
        FOURPTR = ADDR(FWORK);
```

```
        DO I = 1 TO NRECA;
              CALL ZDREA(FOUR,FOURCHAR);
              /* STORE RECORD NUMBER ON ADDRESS WORK FILE */
              WRITE FILE(ZWADD) FROM(FOURCHAR);
        END;
        /* FIND OUT IF CURRENT FIELD VALUE IS HIGHEST ON INDEX. IF
           NOT BRANCH TO GET ANOTHER FIELD VALUE */
        CALL ZPCMP(CVALUE,FVALUE,HWORK);
        IF HWORK = 0 THEN;
                                   ELSE GO TO IXS325;
        /* FREE PROCESSING AREAS AND BRANCH TO SORT AND DISPLAY
           RECORD NUMBERS */
IXS335: FREE SVALUE,FVALUE,PVALUE,CVALUE;
        FREE SCHAREA;
        GO TO IXSRET;
        /* ***** END OF SELECT PROCESSING TYPES 3 AND 4 -
           GREATER THAN AND GREATER THAN OR EQUAL ***** */
        /* ***** START OF SELECT PROCESSING TYPES 5 AND 6 -
           LESS THAN AND LESS THAN OR EQUAL ***** */
        /* DETERMINE IF THE FIELD VALUE ON WHICH THE SEARCH IS
           DEFINED IS LESS THAN THE FIRST VALUE ON THE INDEX. IF SO,
           DISPLAY A MESSAGE AND EXIT */
IXS500: CALL ZPCMP(SCHAREA,SVALUE,HWORK);
        IF HWORK = -1 THEN;
                                        ELSE GO TO IXS505;
        PUT FILE(REPORT) SKIP(2) EDIT
              ('***** SELECT CONDITION NO. ',SCNUM(TABNO))(A,F(2));
        PUT FILE(REPORT) SKIP EDIT
('FIELD VALUE SPECIFIED IS LESS THAN LOWEST VALUE ON INDEX')(A);
        GO TO IXSRT2;
        /* SET ZPOSN TO START OF VALUE AND ADDRESS SUB-FILE */
IXS505: ALLOCATE CVALUE;
        BLOCK = SBLOCK;
        OFFST = 1;
        /* GET A FIELD VALUE */
IXS510: CALL ZDREA(INLEN,CVALUE);
        /* COMPARE THE SEARCH VALUE WITH THE CURRENT ENTRY VALUE */
        CALL ZPCMP(SCHAREA,CVALUE,HWORK);
        IF HWORK = 0 THEN GO TO IXS515; /* EQUAL */
        IF HWORK = 1 THEN GO TO IXS520; /* GREATER THEN */
        GO TO IXS525; /* LESS THAN - END OF SEARCH */
        /* DETERMINE IF SEARCH TYPE CODE IS 5 - LESS THAN. IF SO,
           GO TO END OF SEARCH, OTHERWISE DISPLAY IT */
IXS515: IF STYPE(TABNO) = 5 THEN GO TO IXS525;
        /* DETERMINE NO. OF RECORDS ON LIST */
IXS520: FOURPTR = ADDR(NRECA);
        CALL ZDREA(FOUR,FOURCHAR);
        /* INCREMENT TOTAL NUMBER OF RECORD ADDRESSES FOUND */
        ZTREC = ZTREC + NRECA;
        /* START A LOOP TO GET AND STORE RECORD NOS. */
        FOURPTR = ADDR(FWORK);
        DO I = 1 TO NRECA;
              CALL ZDREA(FOUR,FOURCHAR);
              /* STORE RECORD NUMBER ON ADDRESS WORK FILE */
              WRITE FILE(ZWADD) FROM(FOURCHAR);
        END;
        /* FIND OUT IF CURRENT FIELD VALUE IS HIGHEST ON INDEX. IF
           NOT BRANCH TO GET ANOTHER FIELD VALUE */
        CALL ZPCMP(CVALUE,FVALUE,HWORK);
        IF HWORK = 0 THEN;
```

```
                                        ELSE GO TO IXS510;
        /* FREE PROCESSING AREAS AND BRANCH TO SORT AND DISPLAY
           RECORD NUMBERS */
IXS525: FREE SVALUE,FVALUE,CVALUE;
        FREE SCHAREA;
        /* ***** END OF SELECT PROCESSING TYPES 5 AND 6 -
           LESS THAN AND LESS THAN OR EQUAL ***** */
        /* IF NUMBER OF RECORD ADDRESSES EQUALS ZERO, PRINT A
           MESSAGE AND EXIT */
IXSRET: IF ZTREC = 0 THEN DO;
            PUT FILE(REPORT) SKIP(2) EDIT
            ('***** SELECT CONDITION NO. ',SCNUM(TABNO))(A,F(2));
            PUT FILE(REPORT) SKIP EDIT
            ('NO RECORDS SATISFYING SEARCH PARAMETERS FOUND')(A);
        END;
        /* RETURN TO MAIN PROGRAM */
IXSRT2: RETURN;
        END ZPSEL;
```

```
£COPY *SOURCE* *SINK*
        /* IXPIXS - SELECT RECORDS USING INVERTED INDEX */
ZPIXS:  PROC OPTIONS(MAIN);
        /* THIS PROGRAM DETERMINES ALL THE RECORDS SATISFYING A
           NUMBER OF INPUT SELECTION CONDITIONS INPUT FOR SPECIFIED
           FILE AND FIELD NAMES */
        /* DEFINE INPUT FILE NAME, FIELD NAME, SELECTION OPERATOR
           AND FIELD VALUE AREAS */
        DCL (FILENAME,FIELDNAME) CHAR(20),
            OPERATOR CHAR(3) VAR,
            FIELDVALUE CHAR(255) VAR;
        /* CPU TIME PROCESSING AREAS */
        DCL TIME ENTRY,
            P3 FIXED BIN(31) INIT(3) STATIC,
            CKEY FIXED BIN(31) INIT(1),
            CPR FIXED BIN(31) INIT(0),
            CCRES BIN FLOAT(21),
            CRES BIN FIXED(31),
            CRESA(5) FIXED BIN(31);
        /* DEFINE INPUT AND OUTPUT FILES */
        DCL INFILE FILE INPUT,
            REPORT FILE OUTPUT;
        /* DEFINE TABLE FOR STORING SELECT CONDITIONS ENTERED */
        DCL 1 ZSTAB(20) EXT, /* MAXIMUM 20 CONDITIONS PER QUERY */
              2 SCNUM BIN FIXED, /* SEARCH CONDITION NUMBER */
              2 LVNUM BIN FIXED, /* LEVEL NUMBER */
              2 IBADDR BIN FIXED(31,0), /* INDEX BLOCK ADDRESS */
              2 STYPE BIN FIXED, /* SEARCH TYPE */
              2 FTYPE BIN FIXED, /* FIELD TYPE */
              2 FVLEN BIN FIXED, /* FIELD VALUE LENGTH */
              2 FVPTR PTR; /* POINTER TO FIELD VALUE STORAGE AREA */
        /* DEFINE NUMBER OF ENTRIES IN TABLE */
        DCL ZSINC BIN FIXED EXT INIT(0);
        /* DEFINE ERROR RETURN VARIABLE */
        DCL ZERTF BIN FIXED EXT INIT(0);
        /* DEFINE COMPARISON COUNT FIELD */
        DCL ZCOMP BIN FIXED(31,0) EXT INIT(0);
        /* DEFINE SEARCH CONDITION DELIMITER */
        DCL DELIMITER CHAR(3) VAR;
        /* DEFINE FIRST TIME SWITCH */
        DCL FSWCH BIN FIXED;
        /* DEFINE SELECT PROCESSING FILES */
        DCL (ZWADD,ZAADD,ZCADD) FILE RECORD SEQUENTIAL ENV(F(240,4));
        /* DEFINE TOTALS FOR AND/OR PROCESSING */
        DCL (ANDTOT,ORTOT) BIN FIXED(31,0);
        /* DEFINE A WORK AREA TO PROCESS RECORD ADDRESSES */
        DCL FWORK BIN FIXED(31,0),
            FOURCHAR CHAR(4) BASED(FOURPTR),
            FOURPTR PTR;
        /* DEFINE PREVIOUS LEVEL NUMBER STORAGE */
        DCL PLNUM BIN FIXED;
        /* DEFINE TABLE ENTRY NUMBER PASSED AS PARAMETER */
        DCL TABNO BIN FIXED;
        /* DEFINE TOTAL NUMBER OF RECORDS FOUND IN SELECTION */
        DCL ZTREC BIN FIXED(31,0) EXT;
        /* SORT PARAMETERS */
        DCL FI FIXED BIN(31) INIT(1) STATIC,
            SORT ENTRY,
            PLIRC RETURNS (FIXED BIN(31));
```

```
DCL 1 CSA STATIC,
      2 A1 CHAR(13) INIT('S=BI,,1,4 I=-'),
      2 A2 CHAR(1),
      2 A3 CHAR(19) INIT('@¬TRIM,FB,4,240 C=-'),
      2 A4 CHAR(1),
      2 A5 CHAR(18) INIT('@¬TRIM,FB,4,240 R='),
      2 A6 PIC'(6)9',
      2 A7 CHAR(5) INIT(',END ');
/* PARAMETERS TO EMPTY SELECT WORK FILE */
DCL F2 FIXED BIN(31) INIT(2) STATIC,
   CMD ENTRY,
    WMPTCMD CHAR(10) INIT('£EMPTY -W '),
    OMPTCMD CHAR(10) INIT('£EMPTY -O '),
    EMPTLEN BIN FIXED INIT(10);
/* DEFINE PREVIOUS RECORD ADDRESS AREA */
DCL PWORK BIN FIXED(31,0);
/* PRINT STARTUP MESSAGE */
PUT FILE(REPORT) SKIP(2) EDIT
      ('SELECT RECORDS USING INVERTED INDEX')(A);
PUT FILE(REPORT) SKIP EDIT
      ('==================================')(A);
/* INITIALIZE RUN STATISTICS FIELDS */
DO I = 1 TO 5;
     CRESA(I) = 0;
END;
/* GET AND STORE CURRENT CPU TIME STATUS */
CALL PLCALL(TIME,P3,ADDR(CKEY),ADDR(CPR),ADDR(CRES));
CRESA(1) = CRES;
/* GET CONTROL BLOCKS INTO STORAGE */
CALL ZDOPN;
/* ***** START OF INPUT DATA EDIT PHASE ***** */
/* PRINT START OF EDIT PHASE MESSAGE */
PUT FILE(REPORT) SKIP(2) EDIT
      ('INPUT EDIT PHASE')(A);
PUT FILE(REPORT) SKIP EDIT
      ('----------------')(A);
/* THE SELECT INPUT DATA IS ENTERED BY A FILE NAME FOLLOWED BY
   A NUMBER OF SELECT CONDITIONS SEPARATED BY LOGICAL
   DELIMITERS. THE FORMAT OF A SELECT CONDITION IS:
            'FIELDNAME' 'OPERATOR' 'FIELDVALUE'
    WHERE:
            FIELDNAME IS THE NAME OF THE FIELD WITHIN THE
                FILE PREVIOUSLY NAMED.
            OPERATOR IS THE TYPE OF SEARCH TO BE CARRIED
                OUT. ALLOWED OPERATORS ARE EQUAL(EQ), NOT
                EQUAL(NEQ), GREATER THAN(GT), GREATER THAN
                OR EQUAL(GTE), LESS THAN(LT), LESS THAN OR
                EQUAL(LTE).
            FIELDVALUE IS THE VALUE ON WHICH THE SEARCH IS
                TO BE CARRIED OUT.
    THE ALLOWABLE LOGICAL DELIMITERS BETWEEN SELECT CONDITIONS
    ARE:
            AND - PERFORM AN 'AND' OPERATION.
            OR - PERFORM AN 'OR' OPERATION.
            END - END OF ALL SELECT CONDITIONS ENTERED.
    A MAXIMUM OF 20 SEARCH CONDITIONS ARE ALLOWED PER QUERY */
/* GET THE FILE NAME TO BE PROCESSED */
GET FILE(INFILE) LIST(FILENAME);
/* SET CONDITION AND LEVEL NUMBERS TO INITIAL VALUES */
ISNUM = 0;
```

```
          ILNUM = 1;
          /* INCREMENT TABLE AND CONDITION NUMBERS. THEN LOAD INTO
             TABLE */
IXS100:   ISNUM = ISNUM + 1;
          ZSTNO = ZSTNO + 1;
          SCNUM(ZSTNO) = ISNUM;
          LVNUM(ZSTNO) = ILNUM;
          /* CHECK TO SEE IF NO. OF SEARCH CONDITIONS EXCEEDS 20 */
          IF ZSTNO > 20 THEN DO;
                /* PRINT AN ERROR MESSAGE, SET ERROR FLAG AND BRANCH
                   TO END OF INPUT EDIT PHASE */
                PUT FILE(REPORT) SKIP(2) EDIT
                ('MAXIMUM 20 SEARCH CONDITIONS ALLOWED PER QUERY')(A);
                ZEDTF = 1;
                GO TO IXS105;
          END;
          /* GET THE INPUT SELECT CONDITION */
          GET FILE(INFILE) LIST(FIELDNAME,OPERATOR,FIELDVALUE);
          /* CALL ZPEDT WITH INPUT PARAMETERS */
          CALL ZPEDT(FILENAME,FIELDNAME,OPERATOR,FIELDVALUE);
          /* READ IN A SEARCH CONDITION DELIMITER */
          GET FILE(INFILE) LIST(DELIMITER);
          /* DISPLAY LOGICAL DELIMITER */
          PUT FILE(REPORT) SKIP(2) EDIT
                ('***** LOGICAL DELIMITER ',DELIMITER,' ENTERED')(A,A,A);
          /* PROCESS DELIMITER AND TAKE APPROPRIATE ACTION */
          IF DELIMITER = 'AND' THEN GO TO IXS100; /* GET MORE DATA */
          IF DELIMITER = 'OR' THEN DO;
                ILNUM = ILNUM + 1; /* CHANGE LEVEL NUMBER */
                GO TO IXS100; /* GET MORE DATA */
          END;
          IF DELIMITER = 'END' THEN; /* END OF INPUT */
          ELSE DO; /* ERROR IN DELIMITER INPUT */
                PUT FILE(REPORT) SKIP(2) EDIT
                      ('INVALID DELIMITER ENTERED')(A);
                ZEDTF = 1;
          END;
          /* CHECK FOR ERRORS IN EDITING. IF SO, STOP RUN */
IXS105:   IF ZEDTF = 1 THEN DO;
                PUT FILE(REPORT) SKIP(2) EDIT
          ('EXCEPTION CONDITION IN EDIT PHASE - STOP RUN')(A);
                IERRSW = 1;
                GO TO IXS220;
          END;
          /* ***** END OF INPUT DATA EDIT PHASE ***** */
          /* GET AND STORE CURRENT CPU TIME STATUS */
          CALL PLCALL(TIME,P3,ADDR(CKEY),ADDR(CPR),ADDR(CRES));
          CRESA(2) = CRES;
          /* ***** START OF 'AND' PROCESSING PHASE ***** */
          /* PRINT START OF SELECT PHASE MESSAGE */
          PUT FILE(REPORT) SKIP(2) EDIT
                ('SELECT PROCESSING PHASE')(A);
          PUT FILE(REPORT) SKIP EDIT
                ('----------------------')(A);
          /* THE TABLE OF SELECT CONDITIONS GENERATED IN THE INPUT
             EDIT PHASE IS NOW PROCESSED TO ARRIVE AT A FINAL LIST
             OF RECORD ADDRESSES SATISFYING THE SELECT CONDITIONS.
             'AND' PROCESSING IS CARRIED OUT FIRST, FOLLOWED BY 'OR'
             PROCESSING AND PRINTING OF SEARCH RESULTS. */
          /* SET FIRST TIME SWITCH TO ZERO */
```

```
          FSWCH = 0;
          /* SET ERROR SWITCHES TO ZERO */
          IERRSW = 0;
          NORECS = 0;
          /* OPEN 'OR' FILE FOR OUTPUT */
          OPEN FILE(ZOADD) OUTPUT;
          /* SET ANDTOT AND ORTOT TO ZERO */
          ANDTOT = 0;
          ORTOT = 0;
          /* MAKE RECORD ADDRESS WORK AREA ADDRESSABLE */
          FCURPTR = ADDR(FWORK);
          /* START A LOOP TO PROCESS TABLE ENTRIES */
          /* ********** */ DO I = 1 TO ZSTNO;
          /* CHECK IF FIRST TIME SWITCH IS ZERO. IF SO SET IT TO
             1 AND BRANCH TO PROCESS FIRST TABLE ENTRY */
          IF FSWCH = 0 THEN DO;
                FSWCH = 1;
                GO TO IXS200;
          END;
          /* CHECK TO SEE IF THE PREVIOUS AND CURRENT LEVEL NUMBERS
             ARE THE SAME. IF SO BRANCH TO 'AND' RECORD ADDRESSES */
          IF PLNUM = LVNUM(I) THEN GO TO IXS205;
          /* A CHANGE OF LEVEL NUMBER IMPLIES THAT AN 'OR' HAS BEEN
             SPECIFIED. THE RESULTS FROM PREVIOUS 'AND' PROCESSING
             ARE DUMPED TO THE 'OR' FILE  BEFORE FURTHER 'AND' PROCESSING
             STARTS (IF THERE ARE NOT ZERO RECORD ADDRESSES). */
          IF ANDTOT = 0 THEN GO TO IXS200;
          /* CLOSE WORK FILE FOR OUTPUT */
          CLOSE FILE(ZWADD);
          /* OPEN WORK FILE FOR INPUT */
          OPEN FILE(ZWADD) INPUT;
          /* TRANSFER CONTENTS OF WORK FILE TO 'OR' FILE */
          DO J = 1 TO ANDTOT;
                READ FILE(ZWADD) INTO(FCURCHAR);
                WRITE FILE(ZOADD) FROM(FCURCHAR);
          END;
          /* CLOSE WORK FILE FOR INPUT */
          CLOSE FILE(ZWADD);
          /* ADD 'AND' TOTAL TO 'OR' TOTAL */
          ORTOT = ORTOT + ANDTOT;
          /* OPEN WORK FILE FOR OUTPUT */
IXS200:   OPEN FILE(ZWADD) OUTPUT;
          /* SET TABLE ENTRY NUMBER AND PASS TO ZPSEL TO PERFORM
             SELECT AND PLACE RESULTS IN WORK FILE */
          TABNO = I;
          CALL ZPSEL(TABNO);
          /* CHECK TO SEE IF NO. OF RECORDS SELECTED IS ZERO. IF SO
             SET ANDTOT TO ZERO, CLOSE WORK FILE AND BRANCH TO END
             OF TABLE ENTRY PROCESSING LOOP. OTHERWISE SET 'AND' TOTAL
             TO NUMBER OF RECORDS SELECTED AND BRANCH TO END OF TABLE
             ENTRY PROCESSING LOOP */
          IF ZTREC = 0 THEN DO;
                ANDTOT = 0;
                CLOSE FILE(ZWADD);
                GO TO IXS210;
          END;
          ANDTOT = ZTREC;
          GO TO IXS210;
          /* NO CHANGE IN LEVEL NUMBERS. IF 'AND' TOTAL IS ZERO BRANCH
             ROUND 'AND' PROCESSING */
```

```
IXS2C5: IF ANDTCT = C THEN GC TO IXS210;
        /* SET TABLE ENTRY NUMBER AND PASS TC ZPSEL TC PERFCRM
           SELECT ANC PLACE RESULTS IN WORK FILE */
        TABNO = I;
        CALL ZPSEL(TABNO);
        /* CHECK TO SEE IF NUMBER CF RECCRDS SELECTED IS ZERO. IF SO
           SET ANDTOT TC ZERC, CLCSE WCRK FILE AND BRANCH TC END OF
           TABLE ENTRY PROCESSING */
        IF ZTREC = 0 THEN DO;
              ANDTCT = 0;
              CLCSE FILE(ZWACD);
              GC TC IXS210;
        END;
        /* INCREMENT NUMBER CF RECCRDS IN WORK FILE ANC CLCSE */
        ANDTOT = ANDTOT + ZTREC;
        CLCSE FILE(ZWACD);
        /* SET UP SORT PARAMETERS ANC EXECUTE */
        CSA.A2 = 'W'; /* INPUT IS WORK FILE */
        CSA.A4 = 'A'; /* OUTPUT IS 'ANC' FILE */
        CSA.A6 = ANDTCT; /* NUMBER OF ADDRESSES TO BE SCRTED */
        CALL PLCALL(SCRT,F1,CSA);
        IF PL1RC = 0 THEN;
                     ELSE DO;
              PUT FILE(REPORT) SKIP(2) EDIT
                    ('ERROR RETURN FRCM SORT A')(A);
              IERRSW = 1;
              GC TC IXS220;
        END;
        /* EMPTY PREVICUS SELECT WCRK FILE */
        CALL PLCALL(CMD,F2,WMPTCMD,ADDR(EMPTLEN));
        /* CPEN 'AND' AND WCRK FILES FCR 'ANC' PROCESSING */
        CPEN FILE(ZAACD) INPUT;
        CPEN FILE(ZWACD) OUTPUT;
        /* SET PREVICUS RECCRC ADDRESS TC ZERO */
        PWCRK = C;
        /* 'ANC' PROCESSING - THE 'ANC' FILE (IN SEQUENCE) IS READ
           AND IF TWC SUCCEEDING RECORC ADCRESSES ARE THE SAME THAT
           ADDRESS IS TRANSFERRED TC THE WCRK FILE */
        JWCRK = ANDTCT;
        ANDTOT = C;
        DO J = 1 TC JWORK;
              READ FILE(ZAACD) INTC(FCURCHAR);
              IF FWORK = PWCRK THEN DC;
                    WRITE FILE(ZWACD) FRCM(FCURCHAR);
                    ANDTCT = ANDTCT + 1;
              ENC;
              PWCRK = FWORK;
        END;
        /* CLOSE 'ANC' FILE FCR INPUT */
        CLCSE FILE(ZAACD);
        /* END CF TABLE ENTRY PROCESSING  MOVE CURRENT LEVEL INTC
           PREVICUS AND PROCESS ANCTHER ENTRY (IF ANY) */
IXS210: PLNUM = LVNUM(I);
        /* END CF I LCCP */
        /* ********** */ ENC;
        /* END CF PRCCESSING SELECT CCNDITIONS. CUMP THE RECORD
           ADDRESSES FRCM PREVICUS 'ANC' PROCESSING TC THE 'OR'
           FILE (IF NUMBER CF RECCRC ADCRESSES NCT ECUAL TC ZERO */
        IF ANDTCT = C THEN GC TC IXS215;
        /* CLOSE WORK FILE FCR CUTPUT */
```

```
          CLOSE FILE(ZWADD);
          /* OPEN WORK FILE FOR INPUT */
          OPEN FILE(ZWADD) INPUT;
          /* TRANSFER CONTENTS OF WORK FILE TO 'OR' FILE */
          DO J = 1 TO ANDTOT;
                READ FILE(ZWADD) INTO(FOURCHAR);
                WRITE FILE(ZOADD) FROM(FOURCHAR);
          END;
          /* CLOSE WORK FILE FOR INPUT */
          CLOSE FILE(ZWADD);
          /* ADD 'AND' TOTAL TO 'OR' TOTAL */
          ORTOT = ORTOT + ANDTOT;
          /* CLOSE 'OR' FILE FOR OUTPUT */
IXS215:   CLOSE FILE(ZOADD);
          /* ***** END OF 'AND' PROCESSING PHASE ***** */
          /* GET AND STORE CURRENT CPU TIME STATUS */
          CALL PLICALL(TIME,P3,ADDR(CKEY),ADDR(CPR),ADDR(CRES));
          CRESA(3) = CRES;
          /* ***** START OF 'OR' PROCESSING PHASE ***** */
          /* CHECK TO SEE IF ORTOT IS ZERO. IF SO PRINT A MESSAGE AND
             EXIT */
          IF ORTOT = 0 THEN DO;
                PUT FILE(REPORT) SKIP(2) EDIT
               ('NO RECORDS SATISFY SEARCH PARAMETERS')(A);
                NORECS = 1;
                GO TO IXS220;
          END;
          /* SET UP SORT PARAMETERS AND EXECUTE */
          CSA.A2 = 'O'; /* INPUT IS 'OR FILE' */
          CSA.A4 = 'W'; /* OUTPUT IS WORK FILE */
          CSA.A6 = ORTOT; /* NUMBER OF ADDRESSES TO BE SORTED */
          CALL PLICALL(SORT,F1,CSA);
          IF PLIRC = 0 THEN;
                       ELSE DO;
                PUT FILE(REPORT) SKIP(2) EDIT
                      ('ERROR RETURN FROM SORT B')(A);
                IERRSW = 1;
                GO TO IXS220;
          END;
          /* OPEN WORK FILE FOR 'OR' PROCESSING */
          OPEN FILE(ZWADD) INPUT;
          /* OPEN 'OR' FILE FOR 'OR' PROCESSING */
          OPEN FILE(ZOADD) OUTPUT;
          /* SET PREVIOUS RECORD ADDRESS TO ZERO */
          PWORK = 0;
          /* 'OR' PROCESSING - THE WORK FILE (IN SEQUENCE) IS READ
             AND EACH RECORD ADDRESS CONTAINED IS WRITTEN TO THE
             'OR' FILE (DUPLICATES IGNORED) */
          JWORK = ORTOT;
          ORTOT = 0;
          DO J = 1 TO JWORK;
                READ FILE(ZWADD) INTO(FOURCHAR);
                IF FWORK = PWORK THEN;
                                 ELSE DO;
                      WRITE FILE(ZOADD) FROM(FOURCHAR);
                      ORTOT = ORTOT + 1;
                END;
                PWORK = FWORK;
          END;
          /* CLOSE WORK AND 'OR' FILES */
```

```
        CLOSE FILE(ZWACD);
        CLOSE FILE(ZCACD);
        /* ***** END OF 'OR' PROCESSING PHASE ***** */
        /* GET AND STORE CURRENT CPU TIME STATUS */
        CALL PLCALL(TIME,P3,ADDR(CKEY),ADDR(CPR),ADDR(CRES));
        CRESA(4) = CRES;
        /* ***** START OF ADDRESS DISPLAY PHASE ***** */
        PUT FILE(REPORT) SKIP(2) EDIT
        /* PRINT START OF DISPLAY PHASE MESSAGE */
              ('ADDRESS DISPLAY PHASE')(A);
        PUT FILE(REPORT) SKIP EDIT
              ('--------------------')(A);
        /* DISPLAY FINAL LIST OF RECORD ADDRESSES */
        OPEN FILE(ZOACD) INPUT;
        PUT FILE(REPORT) SKIP(2) EDIT
              ('NO. OF RECORD ADDRESSES = ',CRTOT,' - LIST FOLLOWS')
              (A,F(6),A);
        PUT FILE(REPORT) SKIP;
        DO J = 1 TO CRTOT;
              READ FILE(ZOACD) INTO(FOURCHAR);
              PUT FILE(REPORT) EDIT(FWORK)(F(8));
        END;
        /* ***** END OF ADDRESS DISPLAY PHASE ***** */
        /* GET AND STORE CURRENT CPU TIME STATUS */
IXS220: CALL PLCALL(TIME,P3,ADDR(CKEY),ADDR(CPR),ADDR(CRES));
        CRESA(5) = CRES;
        /* EMPTY PREVIOUS SELECT WORK FILE */
        CALL PLCALL(CMD,F2,WMPTCMD,ADDR(EMPTLEN));
        /* EMPTY PREVIOUS SELECT 'CR' FILE */
        CALL PLCALL(CMD,F2,CMPTCMD,ADDR(EMPTLEN));
        /* IF ERROR SWITCH IS SET, BRANCH ROUND STATISTICS */
        IF IERRSW = 1 THEN GO TO IXS225;
        /* CALCULATE AND DISPLAY CPU TIME USED IN SELECTION PHASES */
        PUT FILE(REPORT) SKIP(2) EDIT
              ('INDEX SELECTION STATISTICS')(A);
        PUT FILE(REPORT) SKIP EDIT
              ('-------------------------')(A);
        PUT FILE(REPORT) SKIP(2) EDIT
              ('NO. OF COMPARISONS = ',ZCCMP)(A,F(8));
        CCRES = CRESA(2) - CRESA(1);
        CCRES = CCRES / 1000.0;
        PUT FILE(REPORT) SKIP(2) EDIT
              ('EDIT CPU TIME(SECONDS) =  ',CCRES)(A,F(10,3));
        CCRES = CRESA(3) - CRESA(2);
        CCRES = CCRES / 1000.0;
        PUT FILE(REPORT) SKIP(2) EDIT
              ('AND CPU TIME(SECONDS) =   ',CCRES)(A,F(10,3));
        IF NORECS = 1 THEN CCRES = 0;
                      ELSE CCRES = CRESA(4) - CRESA(3);
        CCRES = CCRES / 1000.0;
        PUT FILE(REPORT) SKIP(2) EDIT
              ('CR CPU TIME(SECONDS) =    ',CCRES)(A,F(10,3));
        IF NORECS = 1 THEN CCRES = 0;
                      ELSE CCRES = CRESA(5) - CRESA(4);
        CCRES = CCRES / 1000.0;
        PUT FILE(REPORT) SKIP(2) EDIT
              ('LIST CPU TIME(SECONDS) =  ',CCRES)(A,F(10,3));
        CCRES = CRESA(5) - CRESA(1);
        CCRES = CCRES / 1000.0;
IXS225: PUT FILE(REPORT) SKIP(2) EDIT
```

```
      ('TOTAL CPU TIME(SECONDS) = ',CORES)(A,F(10,3));
/* END OF PROGRAM - PRINT MESSAGE AND EXIT */
CLOSE FILE(ZCADD);
PUT FILE(REPORT) SKIP(2) EDIT('END OF RUN')(A);
CALL ZOCLS;
/* END OF PROGRAM */
END ZPIXS;
```

```
&COPY *SOURCE* *SINK*
        /* IXUIXL - DISPLAY INVERTED INDEX STATUS */
ZUIXL:  PROC OPTIONS(MAIN);
        /* THIS PROGRAM DISPLAYS AN INVERTED INDEX FOR A SPECIFIED
           FILE AND FIELD NAME */
        /* DEFINE INPUT AND OUTPUT FILES */
        DCL INFILE FILE INPUT,
            REPORT FILE OUTPUT;
        %INCLUDE IXMSPM;
        %INCLUDE IXMDPM;
        /* DEFINE INPUT FILE AND FIELD NAME STORAGE */
        DCL (FILENAME,FIELDNAME) CHAR(20);
        /* DEFINE PROCESSING OPTION AREA */
        DCL POPT CHAR(3) VAR;
        /* DEFINE ROUTINE FOR SEARCHING DIRECTORY */
        DCL ZPFND ENTRY RETURNS (BIN FIXED(15,0));
        /* DEFINE STORAGE FOR FIELD BEING DISPLAYED */
        DCL DISPAREA CHAR(FDLEN) CTL,
            DISPPTR PTR,
            HBINARY BIN FIXED(15,0) UNALIGNED BASED(DISPPTR),
            FBINARY BIN FIXED(31,0) UNALIGNED BASED(DISPPTR),
            SFLOAT BIN FLOAT(21) UNALIGNED BASED(DISPPTR),
            DFLOAT BIN FLOAT(53) UNALIGNED BASED(DISPPTR);
        /* DEFINE CONTROL PROCESSING AREAS AND VARIABLES */
        DCL NVAB BIN FIXED, HWORK BIN FIXED,
            NWORK BIN FIXED(31,0), MWORK BIN FIXED(31,0),
            FOUR BIN FIXED INIT(4), TWO BIN FIXED INIT(2);
        DCL FOURCHAR CHAR(4) BASED(FOURPTR),
            TWOCHAR CHAR(2) BASED(TWOPTR);
        DCL (FOURPTR,TWOPTR) PTR;
        /* DEFINE LABEL FOR DISPLAY OF RESULTS CODING */
        DCL LABDS LABEL;
        /* DEFINE FIELD TYPE CHARACTER STRING */
        DCL FIELDTYPE CHAR(20) VAR;
        /* DEFINE STORE AREA FOR HIGHEST KEY */
        DCL STORAREA CHAR(FDLEN) CTL;
        /* PRINT STARTUP MESSAGE */
        PUT FILE(REPORT) SKIP(2) EDIT
              ('DISPLAY AN INVERTED INDEX')(A);
        PUT FILE(REPORT) SKIP EDIT
              ('========================')(A);
        /* BRING CONTROL BLOCKS INTO STORAGE */
        CALL ZDCFN;
        /* GET THE FILE AND FIELD NAMES INTO STORAGE */
        GET FILE(INFILE) LIST(FILENAME,FIELDNAME,POPT);
        /* DISPLAY INPUT DATA */
        PUT FILE(REPORT) SKIP(2) EDIT
              ('DISPLAY INDEX ON ',FILENAME,'(',FIELDNAME,')')
              (A,A(20),A,A(20),A);
        /* SEARCH DIRECTORY FOR FILE NAME. IF VALUE RETURNED BY
           ZPFND IS ZERO, THEN NO MATCH HAS BEEN FOUND - PRINT
           AN ERROR MESSAGE AND EXIT */
        IF ZPFND(FILENAME) = 0 THEN DO;
              PUT FILE(REPORT) SKIP(2) EDIT
                    ('FILE ',FILENAME,' IS NOT IN THE DIRECTORY')
                    (A,A(20),A);
              CALL ZDCLS;
        END;
        /* ZPOSN POINTS TO THE APPROPRIATE FILE NAME ENTRY. GET IT
```

```
        INTO STORAGE */
     CALL ZCREA(FILEN,FLENT);
     /* SET ZPOSN TO POINT AT THE START OF THE FIELD NAME ENTRIES */
     BLOCK = FDPTR;
     OFFST = 1;
     /* START A LOOP TO PROCESS FIELD DESCRIPTIONS */
     DO I = 1 TO NOFLDS;
     /* GET A FIELD NAME ENTRY */
     CALL ZCREA(FLDLN,FDENT);
     /* CHECK FOR A MATCH ON FIELD NAME */
     IF FDNAM = FIELDNAME THEN;
                          ELSE GO TO IXL1;
     /* IF THE FIELD IS NOT INDEXED THEN CANCEL RUN */
     IF FDIXPTR = -1 THEN DO;
           PUT FILE(REPORT) SKIP(2) EDIT
                 ('FIELD ',FIELDNAME,' IS NOT INDEXED')
                 (A,A(20),A);
           CALL ZDCLS;
     END;
     /* WE HAVE A VALID FIELD NAME. BRANCH TO DISPLAY INDEX */
     GO TO IXL2;
     /* END OF FIELD NAME PROCESSING LOOP */
IXL1: END;
     /* NO MATCH ON FIELD NAME. PRINT AN ERROR MESSAGE AND EXIT */
     PUT FILE(REPORT) SKIP(2) EDIT
           ('FIELD NAME ',FIELDNAME,' NOT FOUND IN FILE ',FILENAME)
           (A,A(20),A,A(20));
     CALL ZDCLS;
     /* SET ZPOSN TO DISPLAY INDEX CONTROL INFORMATION BLOCKS */
IXL2: BLOCK = FDIXPTR;
     OFFST = 1;
     /* SET UP FIELD RECEIVING AREA AND MAKE BINARY AND FLOAT
        MASKS ADDRESSABLE */
     ALLOCATE DISPAREA;
     DISPTR = ADDR(DISPAREA);
     /* SET UP FIELD TYPE */
     IF FDTYP = 2 THEN DO;
           FIELDTYPE = 'BINARY FIXED(15,0)';
           GO TO IXL3;
     END;
     IF FDTYP = 3 THEN DO;
           FIELDTYPE = 'BINARY FIXED(31,0)';
           GO TO IXL3;
     END;
     IF FDTYP = 4 THEN DO;
           FIELDTYPE = 'BINARY FLOAT(21)';
           GO TO IXL3;
     END;
     IF FDTYP = 5 THEN DO;
           FIELDTYPE = 'BINARY FLOAT(53)';
           GO TO IXL3;
     END;
     FIELDTYPE = 'CHARACTER STRING';
     /* DISPLAY FIELD TYPE AND LENGTH */
IXL3: PUT FILE(REPORT) SKIP(2) EDIT
     ('FIELD TYPE ',FIELDTYPE,', LENGTH ',FDLEN)(A,A,A,F(3));
     /* DISPLAY INDEX CONTROL BLOCK INFORMATION */
     PUT FILE(REPORT) SKIP(2) EDIT
           ('INDEX CONTROL BLOCK INFORMATION')(A);
     PUT FILE(REPORT) SKIP EDIT
```

```
            ('--------------------------------')(A);
        /* GET NUMBER OF VALUE AND ADDRESS BLOCKS AND DISPLAY */
        TWOPTR = ADDR(NVAB);
        CALL ZDREA(TWO,TWOCHAR);
        PUT FILE(REPORT) SKIP(2) EDIT
            ('NO. OF VALUE AND ADDRESS BLOCKS = ',NVAB)(A,F(3));
        /* START A LOOP TO PROCESS INDEX CONTROL BLOCKS */
        FOURPTR = ADDR(NWORK);
        TWOPTR = ADDR(HWORK);
        DO I = 1 TO NVAB;
            CALL ZDREA(FDLEN,DISPAREA);
            CALL ZDREA(FOUR,FOURCHAR);
            CALL ZDREA(TWO,TWOCHAR);
            /* DISPLAY FIELD VALUE */
            LABDS = LABDS1;
            GO TO DISP;
LABDS1:     /* DISPLAY BLOCK ADDRESS */
            PUT FILE(REPORT) SKIP EDIT
('IS HIGHEST KEY ON BLOCK ',NWORK,', OFFSET ',HWORK)(A,F(6),A,F(6));
        END;
        /* SET UP ZPOSN TO DISPLAY VALUE AND ADDRESS BLOCKS */
        BLOCK = FDIXPTR;
        OFFST = 3 + FDLEN;
        CALL ZDREA(FOUR,FOURCHAR);
        BLOCK = NWORK;
        OFFST = 1;
        /* STORE HIGHEST KEY ON FILE IN STORE AREA */
        ALLOCATE STORAREA;
        STORAREA = DISPAREA;
        /* BRANCH TO END IF COMPLETE DISPLAY NOT REQUESTED */
        IF FOPT = 'ALL' THEN;
                        ELSE GO TO IXL5;
        /* DISPLAY VALUE AND ADDRESS BLOCK INFORMATION */
        PUT FILE(REPORT) SKIP(2) EDIT
            ('VALUE AND ADDRESS BLOCK INFORMATION')(A);
        PUT FILE(REPORT) SKIP EDIT
            ('----------------------------------')(A);
        /* GET AND DISPLAY FIELD VALUE */
IXL4:   CALL ZDREA(FDLEN,DISPAREA);
        LABDS = LABDS2;
        GO TO DISP;
        /* GET AND DISPLAY NO. OF RECORD ADDRESSES */
LABDS2: FOURPTR = ADDR(MWORK);
        CALL ZDREA(FOUR,FOURCHAR);
        PUT FILE(REPORT) SKIP EDIT
            ('NO. OF RECORD ADDRESSES = ',MWORK,' - LIST FOLLOWS')
            (A,F(6),A);
        /* GET AND DISPLAY RECORD NUMBERS */
        FOURPTR = ADDR(NWORK);
            PUT FILE(REPORT) SKIP;
        DO I = 1 TO MWORK;
            CALL ZDREA(FOUR,FOURCHAR);
            PUT FILE(REPORT) EDIT(NWORK)(F(8));
        END;
        /* DETERMINE IF THIS IS LAST FIELD VALUE */
        IF DISPAREA = STORAREA THEN;
                               ELSE GO TO IXL4;
        /* FREE STORAGE AREAS, CLOSE INDEX, PRINT END OF RUN MESSAGE
           AND EXIT */
IXL5:   FREE DISPAREA,STORAREA;
```

```
        PUT FILE(REPORT) SKIP(2) EDIT('END OF RUN')(A);
        CALL ZDCLS;
        RETURN;
        /* DISP - IN-LINE CODING TO DISPLAY FIELD VALUES */
DISP:   IF FDTYP = 2 THEN DO;
              PUT FILE(REPORT) SKIP(2) EDIT
                    ('FIELD VALUE ',HBINARY)(A,F(6));
              GO TO DISP1;
        END;
        IF FDTYP = 3 THEN DO;
              PUT FILE(REPORT) SKIP(2) EDIT
                    ('FIELD VALUE ',FBINARY)(A,F(10));
              GO TO DISP1;
        END;
        IF FDTYP = 4 THEN DO;
              PUT FILE(REPORT) SKIP(2) EDIT
                    ('FIELD VALUE ',SFLOAT)(A,F(20,7));
              GO TO DISP1;
        END;
        IF FDTYP = 5 THEN DO;
              PUT FILE(REPORT) SKIP(2) EDIT
                    ('FIELD VALUE ',DFLOAT)(A,F(20,7));
              GO TO DISP1;
        END;
        PUT FILE(REPORT) SKIP(2) EDIT
                    ('FIELD VALUE ',DISPAREA)(A,A);
DISP1:  GO TO LABDS;
        END ZUIXL;
```

```
£COPY *SOURCE* *SINK*
        /* IXCPHY - CONVERT PHYSICS DATA TO TABULAR FORM */
ZCPHY:  PROC OPTIONS(MAIN);
        /* DEFINE FILES */
        DCL REPORT FILE OUTPUT,
            INFILE FILE INPUT,
            IPHYS FILE RECORD SEQUENTIAL ENV(F(80)),
            ZDATA FILE RECORD SEQUENTIAL ENV(F(99));
        /* DEFINE I/O PROCESSING AREAS */
        DCL INAREA CHAR(80),
            ZIOAREA CHAR(99);
        /* DEFINE POINTER AND FILE STRUCTURE FOR OUTPUT RECORDS */
        DCL PHYSPTR PTR;
        DCL 1 PHYSICS BASED(PHYSPTR) UNALIGNED,
              2 EXPNO BIN FIXED(31,0),
              2 ISYMB CHAR(1),
              2 REFER CHAR(29),
              2 PCODE CHAR(13),
              2 ICODE CHAR(10),
              2 ECODE CHAR(10),
              2 ANGLE CHAR(10),
              2 ENRGY BIN FLOAT(21),
              2 ANGVR BIN FLOAT(21),
              2 OBSRV BIN FLOAT(21),
              2 ERROR BIN FLOAT(21),
              2 PNTNO BIN FIXED,
              2 WIDTH BIN FLOAT(21);
        /* DEFINE END EXPERIMENT NUMBER */
        DCL IXPNO BIN FIXED(31,0);
        /* DEFINE CURRENT EXPERIMENT NUMBER */
        DCL CXPNO BIN FIXED(31,0);
        /* DEFINE RECORD COUNTER */
        DCL NRECS BIN FIXED(31,0);
        /* DEFINE PROCESSING SWITCHES */
        DCL (LSWCH,DSWCH) BIN FIXED;
        /* DEFINE DECAY/COMMENT DECISION AREA */
        DCL DCDEC CHAR(1);
        /* ***** DEFINE AREAS TO CONVERT PHYSICS FLOATING POINT
           INPUT ***** */
        DCL (S1,S2,S3,S4) BIN FLOAT(21),
            (E1,E2,E3,E4) BIN FIXED;
        /* CONVERSION ERROR PROCESSING - STORE A NULL DATA RECORD */
        ON CONVERSION BEGIN;
              PUT FILE(REPORT) SKIP(2) EDIT
                    ('CONVERSION ERROR IN RECORD NO. ',NRECS)(A,F(6));
              REVERT CONVERSION;
        ENRGY = 0.0;
        ANGVR = 0.0;
        OBSRV = 0.0;
        ERROR = 0.0;
        PNTNO = 0;
        WIDTH = 0;
        GO TO PHY45;
        END;
        /* MAKE FILE STRUCTURE ADDRESSABLE */
        PHYSPTR = ADDR(ZIOAREA);
        /* SET LSWCH TO ZERO */
        LSWCH = 0;
        /* OPEN FILES AND PRINT A STARTUP MESSAGE */
```

```
        OPEN FILE(IPHYS) INPUT;
        OPEN FILE(ZDATA) OUTPUT;
        PUT FILE(REPORT) SKIP(2) EDIT
              ('CONVERT PHYSICS DATA TO TABULAR FORM')(A);
        /* READ IN AND EXPERIMENT NUMBER AND DISPLAY */
        GET FILE(INFILE) LIST(IXPNO);
        PUT FILE(REPORT) SKIP(2) EDIT
              ('PROCESS UP TO EXPERIMENT NUMBER ',IXPNO)(A,F(6));
        /* SET NUMBER OF RECORDS STORED TO ONE */
        NRECS = 1;
        /* READ IN THE FIRST CARD (MASTER) */
        READ FILE(IPHYS) INTO(INAREA);
        /* GET MASTER DATA AND STORE IN FILE STRUCTURE */
PHY1:   GET STRING(INAREA) EDIT
              (EXPNO,SSYMB,REFER,PCODE,ICODE,ECODE,ANGLE,DCDEC)
              (F(5),A(1),A(29),A(13),A(10),A(10),A(10),X(1),A(1));
        /* END OF INPUT FILE PROCESSING */
        IF EXPNO > IXPNO THEN GO TO PHY7;
        /* TO SAVE SPACE ON THE TABULAR DATA FILE, DECAY AND COMMENT
           INFORMATION CARDS ON THE ORIGINAL DATA FILE ARE NOT
           STORED */
        /* DETERMINE IF A DECAY OR COMMENT CARD FOLLOWS THE CARD JUST
           READ. IF SO, READ IT AND BRANCH BACK TO TEST FOR A FURTHER
           DECAY OR COMMENT CARD. IF NOT, BRANCH TO PROCESS FIRST
           CARD OR NEXT MASTER CARD */
PHY2:   IF DCDEC = ' ' THEN GO TO PHY3;
        READ FILE(IPHYS) INTO(INAREA);
        GET STRING(INAREA) EDIT
              (DCDEC)(X(79),A(1));
        GO TO PHY2;
        /* SET DSWCH TO ZERO */
PHY3:   DSWCH = 0;
        /* READ A DATA CARD */
PHY4:   READ FILE(IPHYS) INTO(INAREA);
        /* GET EXPERIMENT NO. AND COMPARE IT WITH CURRENT EXPERIMENT
           NO. */
        GET STRING(INAREA) EDIT(CXPNO)(F(5),X(75));
        IF EXPNO = CXPNO THEN;
                         ELSE GO TO PHY5;
        /* WE HAVE A DATA CARD. SET DSWCH TO 1, STORE DATA INFO,
           WRITE RECORD TO DISK, INCREMENT RECORD COUNT AND BRANCH
           TO READ ANOTHER DATA CARD */
        DSWCH = 1;
        /* ***** NECESSARY TO CONVERT PHYSICS FLOATING POINT INPUT
           BECAUSE PL/1 WILL NOT ACCEPT A BLANK IN THE EXPONENT
           WHERE A PLUS SIGN IS EXPECTED ***** */
        GET STRING(INAREA) EDIT
              (S1,E1,S2,E2,S3,E3,S4,E4,PNTNO,WIDTH)
              (X(28),4 (F(6,3),X(1),F(3)),F(5),F(6,3),X(1));
        ENRGY = S1 * 10. ** E1;
        ANGVR = S2 * 10. ** E2;
        OBSRV = S3 * 10. ** E3;
        ERROR = S4 * 10. ** E4;
PHY45:  WRITE FILE(ZDATA) FROM(ZIOAREA);
        NRECS = NRECS + 1;
        GO TO PHY4;
        /* CHECK TO SEE IF THERE IS NO DATA. IF NOT, BLANK OUT DATA
           AREAS AND WRITE TO DISK */
PHY5:   IF DSWCH = 0 THEN;
                     ELSE GO TO PHY1;
```

```
PHY6:   ENRGY = 0.0;
        ANGVR = 0.0;
        OBSRV = 0.0;
        ERROR = 0.0;
        PNTNO = 0;
        WIDTH = 0;
        WRITE FILE(ZDATA) FROM(ZICAREA);
        /* ADD 1 TO NO. OF RECORDS STORED */
        NRECS = NRECS + 1;
        /* IF LSWCH IS 1 GO TO END OF PROCESSING */
        IF LSWCH = 1 THEN GO TO PHY8;
                     ELSE GO TO PHY1;
        /* END OF INPUT FILE. SET LSWCH TO 1 AND CHECK FOR DSWCH
           BEING ZERO */
PHY7:   LSWCH = 1;
        IF DSWCH = 0 THEN GO TO PHY6;
        /* DISPLAY NO. OF RECORDS STORED, CLOSE FILES AND EXIT */
PHY8:   NRECS = NRECS - 1;
        PUT FILE(REPORT) SKIP(2) EDIT
              ('NO. OF RECORDS STORED = ',NRECS)(A,F(6));
        CLOSE FILE(IPHYS);
        CLOSE FILE(ZDATA);
        PUT FILE(REPORT) SKIP(2) EDIT('END OF RUN')(A);
        RETURN;
        END ZOPHY;
```