X/Open Developers' Specification

Indexed Sequential Access Method (ISAM)

X/Open Company, Ltd.

X/Open Developers' Specification

Indexed Sequential Access Method (ISAM) ISBN: 1 872630 03 0

Set in Palatino by X/Open Company Ltd., U.K.
Printed by Maple Press, Slough, U.K.
Published by X/Open Company Ltd., U.K.

Any comments relating to the material contained in this document may be submitted to X/Open at:

X/Open Company Limited
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

XoSpecs@xopen.co.uk

Contents

INDEXED SEQUENTIAL ACCESS METHOD (ISAM)

Preface

X/Open

X/Open is an independent, worldwide, open systems organisation supported by most of the world's largest information systems suppliers, user organisations and software companies. Its mission is to bring greater value to users through the practical implementation of open systems.

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable system environment, called the *Common Applications Environment (CAE)*. This environment covers all the standards, above the hardware level, that are needed to support open systems. It ensures portability and connectivity of applications, and allows users to move between systems without retraining.

The interfaces identified as components of the Common Applications Environment are defined in the *X/Open Portability Guide*. This guide contains an evolving portfolio of practical applications programming interface standards (APIs), which significantly enhance portability of application programs at the source code level. The interfaces defined in the X/Open Portability Guide are supported by an extensive set of conformance tests and a distinct trademark - the X/Open brand - that is carried only on products that comply with the X/Open definitions.

X/Open is thus primarily concerned with standards selection and adoption. The policy is to use formal approved *de jure* standards, where they exist, and to adopt widely supported *de facto* standards in other cases.

Where formal standards do not exist, it is X/Open policy to work closely with standards development organizations to encourage the creation of formal standards covering the needed functionalities, and to make its own work freely available to such organizations. Additionally, X/Open has a commitment to align its definitions with formal approved standards.

The X/Open Product Family - XPG

There is a single family of X/Open products, which has the generic name ''XPG''.

XPG Versions

There are different numbered versions of XPG within the XPG family (XPG1, XPG2, XPG3). Each XPG version is an integrated set of elements supporting the development, procurement and implementation of open systems products, and each comprises its own:

- XPG Specifications
- XPG Verification Suite
- XPG descriptive guides

- XPG trademark licensing materials

The XPG trademark (or ‘‘brand’’) licensed by X/Open always contains a particular XPG version number (e.g., ‘‘XPG3’’) and, when associated with a vendor’s system, communicates clearly and unambiguously to a procurer that the software bearing the trademark correctly implements the corresponding XPG specifications. Users specifying particular XPG versions in their procurements are therefore certain as to the XPG specifications to which vendors’ systems conform.

XPG Specifications

There are four types of XPG specification:

- **XPG*n* Formal Specifications**

  These are the long-life XPG specifications that form the basis for conformant/branded X/Open systems, and are the only type of XPG specification released with an XPG version number (e.g., ‘‘XPG3’’). They are intended to be used widely within the industry for product development and procurement purposes. Currently, all XPG Formal Specifications are included in Issue 3 of the X/Open Portability Guide.

  Individual XPG specifications are released as Formal Specifications only as part of the formal release of the complete XPG version to which they belong. However, prior to the launch of that XPG version, they may be made available as:

- **XPG Developers’ Specifications**

  These are specifically designed to allow developers to create X/Open-compliant products and applications in advance of the formal launch of a future version of the XPG.

  Developers’ Specifications may be relied on by product developers as the final, base specification that will appear in a future XPG. They are made available beforehand in order to meet the need of product developers for advance notification of the contents of XPG Formal Specifications, to assist in their product planning and development activities.

  By providing such advance notification, X/Open makes it possible for products conforming to future XPG Formal Specifications to be developed as soon as practicable, enhancing the value of XPG itself as a procurement aid to users.

- **XPG Preliminary Specifications**

  These are XPG specifications, usually addressing an emerging area of technology, and consequently not yet supported by a base of conformant product implementations, that are released in a controlled manner for validation purposes. A Preliminary Specification is not a ‘‘draft’’ specification. Indeed, it is as stable as X/Open can make it, and on publication will have gone through the same rigorous X/Open development and review procedures as XPG Formal and Developers’ Specifications.

  Preliminary Specifications are analogous with the ‘‘trial-use’’ standards issued by formal standards organizations, and product development teams are intended to develop product on the basis of them. Because of the nature of the technology they are addressing, they are untried in practice, and they may therefore change before being published as an XPG Formal or Developers’ Specification.

- **Snapshot Specifications**

  These are ‘‘draft’’ documents, that provide a mechanism for X/Open to disseminate information on its current direction and thinking to a limited audience, in advance of formal publication, with a view to soliciting feedback and comment.

This Document

This document is an XPG Developers’ Specification (see above for the implications of this).

Data management is a key element in the integration of applications. Applications, written in a variety of languages, must be able to work on the same basic data in the same form, and data must be passed easily and efficiently between applications.

As a first step towards addressing these issues, X/Open defines an interface for the creation, management and manipulation of indexed files, generally known as the **I**ndexed **S**equential **A**ccess **M**ethod (**ISAM**). The availability of this interface on X/Open compliant systems not only provides application portability, but eases and encourages integration.

The ISAM definition published in XPG3 was not fully compatible with the **IS 1989:1985** (identical to **ANS X3.23-1985)** COBOL standard.

This Developers’ Specification contains the necessary extensions. The variable length extension is marked as optional, i.e. it is not required in a C programming environment, but it is needed in a mixed C and COBOL environment.

It is intended to make the extension mandatory in the next issue of this specification.

Trademarks

Acknowledgements

X/Open gratefully acknowledges:

- **Informix Software Inc.** of Menlo Park, California for permission to use material from the specification of their C-ISAM product and for provision of that material in machine readable form.

Referenced Documents

The following documents are referenced in this guide:

- Informix Software Inc. C-ISAM Reference Manual (Version 2.10 - January 1985);
- Standard for COBOL (IS 1989:1974, identical to ANS X3.23-1974);
- Standard for COBOL (IS 1989:1985, identical to ANS X3.23-1985).

Chapter 1

Introduction

1.1 OVERVIEW

The input/output facilities supported by the operating system consist only of byte-stream read and write operations on files. No facilities are provided for operating on files as sets of records. This leads to application writers having to make their own arrangements for record handling, resulting in both a multiplication of effort and a proliferation of non-standard methods.

Data management is a key element in the integration of applications. Applications, written in a variety of languages, must be able to work on the same basic data in the same form, and data must be passed easily and efficiently between applications.

As a first step towards addressing these issues, X/Open defines an interface for the creation, management and manipulation of indexed files, generally known as the **I**ndexed **S**equential **A**ccess **M**ethod (**ISAM**). The availability of this interface on X/Open compliant systems will not only provide application portability, but will ease and encourage integration.

The X/Open ISAM Definition is structured as follows:

Chapter 2	gives an overview of ISAM.
Chapter 3	describes data types supported by the X/Open ISAM definition.
Chapter 4	describes the definition and manipulation of indexes and techniques for key compression.
Chapter 5	describes file and record locking techniques to ensure reliable updating in multi-user environments.
Chapter 6	contains a comprehensive set of C and COBOL program examples designed to illustrate all the facilities of the ISAM interface.
Chapter 7	describes the handling of exception conditions.
Chapter 8	describes the **<isam.h>** header containing definitions of various macros and symbolic constants.
Chapter 9	contains general notes on the interfaces in the X/Open ISAM definition, detailed in Chapter 10.
Chapter 10	contains detailed specifications of the interfaces in the X/Open ISAM definition.

1.2 DOCUMENT HISTORY

1.2.1 Notes on XPG1 and XPG2

The X/Open ISAM definition is a major subset of the specification of the C-ISAM product, version 2.10, published by Informix Software Inc. of Menlo Park, California.

The X/Open definition omits parts of the C-ISAM specification which are implementation-specific. An example is the audit trail facility which is defined in the C-ISAM document without any interfaces for recovery. Internal file formats are given and the user has to make direct use of these to effect recovery. As alternative implementations may exist, these internal file formats are not part of the X/Open standard, and neither, therefore, are the audit trail definitions. (Any use of these facilities on a system that includes them will imply that such applications are not totally portable across X/Open compliant systems.)

Version 2.10 of the C-ISAM product introduced new functions, and a set of decimal data types. The new functions were included in XPG1 in the ‘‘optional’’ category, but this limitation was removed in XPG2. The decimal types were excluded.

1.2.2 Notes on XPG3

The ISAM definition published in XPG2 did not include detailed descriptions of certain functions with respect to locking and error handling. In some cases the status of the current record pointer is not precisely defined.

The XPG3 specification contained more detailed descriptions of certain areas, like current record position, locking, access mode compatibility and physical order.

There is great interest among customers and vendors in making it possible to access the same ISAM file from both C and COBOL indexed I-O. To achieve this goal some changes in the behaviour of ISAM were made in XPG3. These changes are described in detail below.

The XPG3 ISAM definition is compatible with the **IS 1989:1974** (identical to **ANS X3.23-1974**) COBOL standard. It also conforms to the **IS 1989:1985** (identical to **ANS X3.23-1985**) standard with the exception that variable length records are not supported. Also, the values of the I-O status information variables, *isstat1* and *isstat2*, conform to **IS 1989:1974** (identical to **ANS X3.23-1974**), but compliance to **IS 1989:1985** (identical to **ANS X3.23-1985**) can be achieved by a COBOL run-time system that sets the appropriate value of the I-O status.

Changes

isopen( )

In previous issues of the Guide this function call positioned the current record pointer to the first record in the order of the primary index. In XPG3, the *isopen*( ) definition was changed to meet the **IS 1989:1985** (identical to **ANS X3.23-1985**) requirement that the current record pointer be positioned just before any possible first record in the order of the primary key. More detailed information on this topic can be found under **Section 9.4**, **Current Record Position**.

isrelease( ), *isunlock*( )

The previous issues stated that *isrelease*( ) unlocks records that have been previously locked with manual record locking while *isunlock*( ) unlocks a file that has been previously locked with manual file locking. Because the two levels of locking could be used together and because C-ISAM and other existing ISAM implementations work in a different way to the above, in XPG3 the behaviour has been changed so that both functions can be used as synonyms to unlock files and records. Further information on this topic can be found in **Chapter 5**, **Locking**.

isrewcurr( )

The previous issues stated that *isrewcurr*( ) rewrites the current record, whose position is left unchanged. In XPG3, the description was stated more precisely, so that the current record position always points to the rewritten record. In other words, if the key value of the current record is changed for the selected index, the current record position is changed to point to the record with the new key value. This behaviour is consistent with the current C-ISAM implementation. In order to meet the **IS 1989:1985** (identical to **ANS X3.23-1985**) requirement that the file position indicator should not be affected by a COBOL REWRITE statement, the *isrewrec*( ) or *isrewrite*( ) function calls must be used instead.

Additions

CHARTYPE

Data of type **CHARTYPE** are stored as non-terminated character strings padded with trailing blanks. The following two conversion routines for characters were added in XPG3:

ldchar( )	returns a null-terminated character array without trailing spaces from **CHARTYPE**.
stchar( )	stores a null-terminated character array into **CHARTYPE** by removing the null character and padding the string with trailing blanks.

isread( )

In default mode *isread*( ) does not change the current record pointer if the record to be read is locked by another process. An additional flag, ISSKIPLOCK, was been introduced, as an option of the mode parameter, to enable skipping a locked record.

isstart( )

In automatic record locking mode, *isstart*( ) by default unlocks the locked record as any other function call does. An additional flag, ISKEEPLOCK, was been introduced, as an option of the mode parameter, to enable the record lock to be kept. This is necessary to be consistent with current COBOL run-time systems that do not unlock the record after a START statement.

1.2.3 Notes on This Developers' Specification

The ISAM definition published in XPG3 was not fully compatible with the **IS 1989:1985** (identical to **ANS X3.23-1985)** COBOL standard.

This issue contains the necessary extensions. The variable length extension is marked as optional, i.e. it is not required in a C programming environment, but it is needed in a mixed C and COBOL environment.

It is intended to make the extension mandatory in the next issue of this Guide.

Additions

Two I-O status information variables, *isstat3* and *isstat4*, have been added whose values conform to the **IS 1989:1985** (identical to **ANS X3.23-1985)** COBOL standard.

The definition has been extended with the facility to support files having records of variable length. This extension is described in **Section 9.8**, **Records**.

Chapter 2

ISAM Overview

The X/Open ISAM definition specifies a set of C language functions that create and manipulate indexed files.

These functions provide for:

- the creation of files and associated primary indexes;
- the addition and deletion of further indexes;
- the opening, closing and deletion of existing files;
- the selection of the index to be used for subsequent reading and/or writing of records, and the start point within the file;
- the reading, writing and updating of data records, and
- the locking and unlocking of files and records.

When a file is created, two conceptual entities are formed, the container for holding data records and a primary index. The programmer can specify the field, or fields, of each record that is to be used as the primary key for distinguishing the records within the file. As each record is written to the file, an entry is made in the index which stores key value(s) together with the location of the data record in the file. For subsequent reads on the file, individual records are located by searching the index for the required key and using the location stored with it to go straight to the data. Access to a file can be sequential or random.

Indexes additional to the primary index can be created. These provide alternative access paths to the same data records by allowing different fields to be used as the keys. The definition puts no limit on the number of alternative indexes that can be created for a file. In an additional index, the same key value is allowed to occur in different records, ‘‘duplicates’’, although a facility is provided to inhibit this on any particular file. Duplicates are allowed for the primary key in ISAM. However, this feature should not be used if the file is ever to be accessed by a COBOL program.

The definition includes the facility to specify index key compression. This allows the density of key storage in an index to be increased by the use of such techniques as suppression of redundant spaces at the beginning and end of keys and by the elimination of duplicate entries. Only *no compression* and *maximum compression* are fully defined. However, it is recognised that intermediate levels may be provided on any particular member system, and mode values are defined to allow for this. All X/Open compliant systems will accept these values to ensure application portability, although the degree of resulting compression may vary.

Facilities are defined for the locking of files and records, to ensure reliable update and access in the multi-user environment. File locking locks out a whole file. It may be exclusive, in that all other accesses to the file are inhibited, or it may be write-only, allowing read accesses to continue. Record level locking may be automatic. In this case it is specified at file open time and a record is automatically locked before it is read, and remains locked until the next function call (except for *isstart*( ) with option ISKEEPLOCK), is executed successfully. It is implementor-defined whether or not an unsuccessful execution releases the lock. Alternatively, it may be manual in that it is actioned as a result of a parameter of a read call.

The definition includes the optional facility to support files having records of variable length. When a file is built it is declared to contain either fixed length or variable length records. Fixed length records must contain the same number of bytes for all the records in the file. Variable length records may contain differing numbers of bytes among the records on the file.

The following functions are included in the X/Open ISAM definition:

Function Name	Purpose
isaddindex( )	add index to an ISAM file
isbuild( )	create an ISAM file
isclose( )	close an ISAM file
isdelcurr( )	delete current record
isdelete( )	delete record specified by primary key
isdelindex( )	remove index from an ISAM file
isdelrec( )	delete record specified by record number
iserase( )	remove an ISAM file
isindexinfo( )	access file information
islock( )	lock an ISAM file
isopen( )	open an ISAM file
isread( )	read records
isrelease( )	unlock records
isrename( )	rename an ISAM file
isrewcurr( )	rewrite current record
isrewrec( )	rewrite record specified by record number
isrewrite( )	rewrite record specified by primary key
isstart( )	select an index
isunlock( )	unlock an ISAM file
iswrcurr( )	write record and set current position
iswrite( )	write record

The following C-ISAM facilities are not included within the X/Open ISAM definition and their use will impede portability:

Function Name	Purpose
isaudit( )	perform operations on audit trail
isflush( )	flushe buffered index pages
issetunique( )	set unique identifier
isuniqueid( )	return unique identifier

Also excluded are the decimal data types and associated manipulation routines.

Chapter 3

Data Types

The types of data that can be defined and manipulated are described in this chapter. Descriptions of how each data type is stored in files and how each data type must be treated are also included.

The data types for which properly ordered indexes are maintained are type character, 2-byte integers, 4-byte integers, machine float (floating point) and machine double (double precision floating point). The macro definitions used to describe these types are shown below. These definitions can also be found in **<isam.h>**.

CHARTYPE	character
INTTYPE	2-byte integer
LONGTYPE	4-byte integer
FLOATTYPE	machine float
DOUBLETYPE	machine double

3.1 CHARTYPE

The data type **CHARTYPE** comprises a string of characters, for example, a city name or an address.

Two routines are supplied for the conversion to and from ISAM character storage format. These routines are:

ldchar(*p, l, s*) transfers data of **CHARTYPE** to the C array of **char** *s*; *p* is a **char** pointer to the starting byte of format **CHARTYPE**. The transfer stops after *l* characters, trailing spaces are removed and the string is null-terminated.

stchar(*s, p, l*) stores a C array of **char** *s* at location *p*, where *p* is a **char** pointer to the starting byte of format **CHARTYPE**. The data starting at location *p* is padded with trailing spaces up to but not including position *p* + *l*.

3.2 INTTYPE AND LONGTYPE

The data types **INTTYPE** and **LONGTYPE** consist of 2 and 4-byte binary signed integer data. Integer data is always stored in files as high/low, most significant byte first, least significant byte last. This storage technique is independent of the form in which integers are stored in the machine on which the system is executing. Therefore, depending on the operating environment, the format of storage for integers in the files may differ from the format of storage for integers stored in executing programs. For this reason, four routines are supplied for the conversion to and from ISAM integer storage format.

The four format conversion routines for integers are:

ldint(*p*) returns a machine-format short integer; *p* is a **char** pointer to the starting byte of format **INTTYPE**.

stint(*i, p*) stores a machine-format short integer *i* as format **INTTYPE** at location *p*, where *p* is a **char** pointer to the first byte of format **INTTYPE**.

ldlong(*p*) returns a machine-format long integer; *p* is a **char** pointer to the first byte of format **LONGTYPE**.

stlong(*l, p*) stores a machine-format long integer *l* as format **LONGTYPE** at location *p*, where *p* is a **char** pointer to the first byte of format **LONGTYPE**.

These routines are either macros defined in **<isam.h>** or are in the ISAM library.

The typical use for the above routines occurs after a data record has been read into the user buffer. Integer values that are to be used by the user program first have to be converted to machine-usable format by using *ldint*( ) for type **INTTYPE** and *ldlong*( ) for **LONGTYPE**.

Similarly, storage of machine-format integer data requires the use of the *stint*( ) and *stlong*( ) routines.

Note that the formatted integers need not be aligned along word boundaries as do machine-formatted integers.

3.3 FLOATTYPE AND DOUBLETYPE

The data types **FLOATTYPE** and **DOUBLETYPE** are the two floating point data types. The data type **FLOATTYPE** is the same as the C data type **float**, while the data type **DOUBLETYPE** is the same as the C data type **double**. Both data types differ in length and format from machine to machine. There is no difference between the floating point format used and its counterpart in the C language except that floating point numbers may be placed on non-word boundaries. For this reason, four more routines allow the user to retrieve or replace these non-aligned floating point numbers from their positions in data records. These routines are:

ldfloat(*p*) returns a machine-format **float**; *p* is a **char** pointer to the starting (leftmost) byte of format **FLOATTYPE**.

stfloat(*f, p*) stores a machine-format **float** *f* at location *p*, where *p* is a **char** pointer to the starting (leftmost) byte of format **FLOATTYPE**.

lddbl(*p*) returns a machine-format **double**; *p* is a **char** pointer to the starting (leftmost) byte of format **DOUBLETYPE**.

stdbl(*d, p*) stores a machine-format **double** *d* as format **DOUBLETYPE** at location *p*, where *p* is a **char** pointer to the starting (leftmost) byte of format **DOUBLETYPE**.

The use of the floating point load and store routines is analogous to the use of the integer load and store routines.

Chapter 4

Indexing

4.1 INDEX DEFINITION AND MANIPULATION

The C language structures that describe an index to any given function call are the **keydesc** and **keypart** structures. These structures are shown below. They are defined in the header **<isam.h>**, which must be included in any program which uses the function calls.

The structure **keydesc** contains the following members:

```
short k_flags;                        /* flags */
short k_nparts;                       /* number of parts in key */
struct keypart k_part[NPARTS];        /* each key part */
```

The structure **keypart** contains the following members:

```
short kp_start;                       /* starting byte of key part */
short kp_leng;                        /* length in bytes of key part */
short kp_type;                        /* type of key part */
```

It is the purpose of this chapter to show how to initialise the **keydesc** structure for use with any of the functions that require it as a parameter.

The first sample index to be described here has one part which has the data type of **INTTYPE**. Integers are 2 bytes; therefore, the length of the index is 2 bytes. The index begins in the first byte of the record. No data compression is desired for keys stored in this index. The order of the index is to be ascending (lowest key value to highest key value). Finally, duplicate key values for this index are not to be allowed.

The C program to add the index described above is shown below. It is assumed that the file *myfile* has already been created using the *isbuild*( ) function call.

```
#include <isam.h>

struct keydesc first_key;
int fd;

main()
{
          /* In order to add an index to the file
            "myfile", the file must be opened with
            exclusive access.  Therefore, ISEXCLLOCK
            must be arithmetically added to the mode
            parameter. */

          if ((fd = isopen("myfile", ISINOUT+ISEXCLLOCK)) < 0)
          {
                    printf("Open error %d on myfile.\n", iserrno);
                    exit(1);
          }
          mkfirst_key();
          if (isclose(fd))
          {
                    printf("Close error %d on myfile.\n", iserrno);
                    exit(1);
          }
}

mkfirst_key()
{
          first_key.k_flags = ISNODUPS;          /* no dups, no compression */
          first_key.k_nparts = 1;                /* this index has one part */

          /*The starting byte of an index is always defined
            as the byte offset from the beginning of the
            record.  Since this index begins at the begin-
            ning of the record, its byte offset is zero. */

          first_key.k_part[0].kp_start = 0;               /* offset is zero */
          first_key.k_part[0].kp_type = INTTYPE;          /* data type is integer */
          first_key.k_part[0].kp_leng = INTSIZE;          /* 2 byte integer */

          if(isaddindex(fd, &first_key))                  /* add the index  */
          {
                    printf("Error %s iserrno = %d.\n",
                              "in adding first_key index: ", iserrno);
          }
}
```

Note that, in the above example, the structure element *k_flags* is initialised to zero. This indicates that no special characteristics are to be attributed to this index. Since *k_flags* is zero, duplicate key values will not be allowed, and no compression will be performed on key values as they are placed in the index.

If duplicate key values were to have been allowed, *k_flags* should have been initialised to ISDUPS as in the following statement:

```
/∗ allow duplicate key values ∗/
first_key.k_flags = ISDUPS;
```

If key value compression had been desired, *k_flags* should have been initialised to *ISDUPS+COMPRESS*. This would allow duplicate key values and would indicate that they be compressed in the index.

```
first_key.k_flags = ISDUPS+COMPRESS;
```

Note, also, that the index defined by the **keydesc** structure *first_key* has only one part. The number of key parts that make up the index is defined by the structure element *k_nparts,* which in the above example is initialised to one.

```
/∗ this index has one part ∗/
first_key.k_nparts = 1;
```

In the previous example, the index defined had only one part. That part had a data type of **INTTYPE**. However, a particular application could require that a multi-part index be used. Within the **keydesc** structure there exists an array of **keypart** structures. Each **keypart** structure defines one part of the index. It holds the starting byte offset from the beginning of the record, the part's length, and the part's data type. In order for a multi-part index to be described, the user's program must initialise each of these structures to reflect the desired position, length and data type for each index part.

In the following example program, a 3-part index is defined. The index consists of a **CHARTYPE** field, a **LONGTYPE** field and another **CHARTYPE** field. It is important to note that the parts of an index need not be contiguous within a record, nor do the parts of an index have to exist in any particular order within the record. However, the maximum number of key parts that can be defined for an index is {NPARTS}, and the total number of bytes within an index cannot exceed {MAXKEYSIZE}. The number of keys that can be added to a file is guaranteed to be not less than 15.

```
#include <isam.h>

struct keydesc second_key;
int fd;

main()
{
        if ((fd = isopen("myfile", ISINOUT+ISEXCLLOCK)) < 0)
        {
                printf("Open error %d on myfile.\n", iserrno);
                exit(1);
        }
        mksecond_key();
        if (isclose(fd))
        {
                printf("Close error %d on myfile.\n", iserrno);
                exit(1);
        }
}

mksecond_key()
{
        /* allow dups, full compression */
        second_key.k_flags      = ISDUPS+COMPRESS;

        /* this index has 3 parts */
        second_key.k_nparts     = 3;

        /* define the first index part */
        second_key.k_part[0].kp_start   = 15;
        second_key.k_part[0].kp_leng    = 8;
        second_key.k_part[0].kp_type    = CHARTYPE;

        /* define the second index part */
        second_key.k_part[1].kp_start   = 30;
        second_key.k_part[1].kp_leng    = LONGSIZE;
        second_key.k_part[1].kp_type    = LONGTYPE;

        /* define the third index part */
        second_key.k_part[2].kp_start   = 3;
        second_key.k_part[2].kp_leng    = 6;
        second_key.k_part[2].kp_type    = CHARTYPE+ISDESC;

        if (isaddindex(fd, &second_key))
        {
                printf("Error %s iserrno = %d.\n",
                        "in adding second_key index: ", iserrno);
        }
}
```

4.2 INDEX COMPRESSION

This section discusses key value compression. This allows the density of key storage in an index to be increased by the use of such techniques as suppression of redundant spaces at the beginning and end of keys and the elimination of duplicate entries.

Using these techniques, significant savings can be made in disc space, and substantial improvements obtained in response to random access requests.

Different levels of compression may be available on different machines. To allow for this, the X/Open definition is non-specific, but ensures that applications will run across X/Open compliant systems without change.

Two levels of space compression are defined: *no compression* and *maximum compression*. The latter calls for the maximum level of space compression available on the machine on which the application is running. The levels apply to each index individually.

In addition, an application can specify whether duplicates are to be allowed for each index.

Duplicates are allowed by setting the value ISDUPS into the *k_flags* field of the **keydesc** structure for a given index, and are inhibited by the value ISNODUPS. (As no default value is defined, either ISDUPS or ISNODUPS must be specified.) Space compression is specified by adding the value COMPRESS to ISDUPS or ISNODUPS. All other values in the *k_flags* field are implementation-defined, but an X/Open compliant system will accept such values as advisory (i.e., applications will not fail, but the level of compression obtained may vary from machine to machine).

Chapter 5

Locking

Two levels of locking are defined: file level locking and record level locking. Within these two levels the user can choose from among several methods the one which best suits application requirements.

5.1 EXCLUSIVE FILE LOCKING

File locking may be accomplished in two ways. One method prevents other processes from reading from or writing to a given file. This method is referred to as an exclusive lock and remains in effect from the moment the file is opened, using *isopen*( ) or *isbuild*( ), until the file is closed using *isclose*( ). Exclusive file locking is specified by adding ISEXCLLOCK to the *mode* parameter of the *isopen*( ) or *isbuild*( ) function call. If a file is opened in exclusive mode, any function calls for manual file or record locking (for this file) are treated as no operation.

Exclusive file level locking is not necessary for most situations, but it must be used when an index is being added using *isaddindex*( ) or when an index is being deleted using *isdelindex*( ).

The skeleton program shown below illustrates how exclusive file level locking is done:

```
myfd = isopen("myfile", ISEXCLLOCK+ISINOUT);
         .
         .
         .
isclose(myfd);
```

5.2 MANUAL FILE LOCKING

Manual file level locking prevents other processes from writing to a given file but allows them to read the locked file. This kind of file level locking is specified by use of the *islock*( ) and *isunlock*( ) function calls. When a file is to be locked in this manner, ISMANULOCK must be added to the *mode* parameter of the *isopen*( ) or *isbuild*( ) call. Later in the program, when locking is desired, *islock*( ) should be called to lock the file. When the file is to be unlocked, *isunlock*( ) should be called. If no previous lock occurred, *isunlock*( ) is treated as no operation, and thus returns no error.

Example for manual file locking:

```
myfd = isopen("myfile", ISMANULOCK+ISINOUT);
        .
        . /* "myfile" is unlocked here */
        .
islock(myfd);
        .
        . /* "myfile" is locked here */
        .
isunlock(myfd);
        .
        . /* "myfile" is unlocked here */
        .
isclose(myfd);
```

5.3 RECORD LEVEL LOCKING

There are two basic types of record level locking: automatic and manual.

Automatic record locking locks a record just before it is read using the *isread*( ) call. It unlocks the record after the next call has completed. Automatic record locking is used when the user wants to lock one record at a time and is unconcerned about when or for how long that record will be locked.

Manual record locking, on the other hand, can lock any number of records. Manual locking locks a record when that record is read using *isread*( ). It unlocks that record, and any other records that are currently locked, when *isrelease*( ) is called. Manual record locking is used when more control is required over when a record, or set of records, is to be locked and unlocked.

Both automatic and manual locking techniques allow other processes to read records locked by the current process as long as the other processes are not trying to lock those records (either manually or automatically). However, processes cannot lock, re-write or delete records locked by another process.

5.3.1 Automatic Record Locking

Automatic record locking must be specified when the file is opened. This is done by adding ISAUTOLOCK to the *mode* parameter of the *isopen*( ) or *isbuild*( ) function call. From when the file is opened until it is closed, every record will be locked automatically before it is read. Each record remains locked until the next function call (except *isstart*( ) with option ISKEEPLOCK) is completed without errors for the current file. It is implementor-defined whether an unsuccessful execution releases the lock. Therefore, while using the automatic record locking mechanism, only one record per file may be locked at a given time. If the option ISKEEPLOCK is used with *isstart*( ) the record will not be unlocked, i.e., it remains locked. Without this option *isstart*( ) will unlock the record as any other ISAM function call does (including *isindexinfo*( ) and *isrelease*( )). The function calls *islock*( ) and *isunlock*( ) should not be used in automatic record locking mode.

The following illustration shows how automatic record locking is used:

```
myfd = isopen("myfile", ISINOUT+ISAUTOLOCK);
        .
        .
        .
isread(myfd, myrecord, ISNEXT);
        .   /* record locked here */
        .   /* before record is read */
        .
isrewcurr(myfd, myrecord);
        .   /* record unlocked here */
        .   /* after completion */
        .
isclose(myfd);
```

5.3.2 Manual Record Locking

The user's intention to use manual record locking must be specified before any processing takes place. This is done by adding ISMANULOCK to the *mode* parameter of *isopen*( ) or *isbuild*( ) function calls when the file is opened. After the file is open, if the user wishes a record to be locked, ISLOCK must be added to the *mode* parameter of the *isread*( ) function call that is reading that record. Each and every record that is read in this manner remains locked until they are all unlocked by a call of the *isrelease*( ) function. The number of records that may be locked in this manner at any one time is system dependent. Manual record locking has no effect (is treated as no operation) if ISMANULOCK has not been specified at the *isopen*( ) or *isbuild*( ) call for that file descriptor. Calling *isread*( ) with the ISLOCK option has no locking effect if the file is exclusively locked. If no previous lock occurred *isrelease*( ) is treated as no operation, and thus returns no error.

The following illustration shows how a number of records in a particular file are locked and unlocked using manual record locking:

```
myfd = isopen("myfile", ISINOUT+ISMANULOCK);
        .
        .
        .
isread(myfd, first_record, ISEQUAL+ISLOCK);
        .
        .
        .
isread(myfd, second_record, ISEQUAL+ISLOCK);
        .
        .
        .
isread(myfd, third_record, ISEQUAL+ISLOCK);
        .
        .
        .
isrelease(myfd);
/* unlock all three records */
        .
        .
isclose(myfd);
```

5.4 COMBINING MANUAL FILE AND RECORD LOCKING

Manual file and record locking may be used together. In this case, the *isunlock*( ) and the *isrelease*( ) function calls both have the same behaviour: they unlock both the file and all records for that file.

5.5 LOCKING MATRIX

The following table shows the effect of several function calls in combination with the three different locking modes:

Function call	EXCLUSIVE	MANUAL	AUTOMATIC
islock( )	no operation	locks file	implementor defined
isunlock( )	no operation	unlocks file & records	implementor defined
isrelease( )	no operation	unlocks file & records	unlocks record
isread( )	no lock effect	no lock effect	unlocks previous, locks current record
isread (ISLOCK)	no lock effect	locks record	unlocks previous, locks current record
isstart( )	no lock effect	no lock effect	unlocks previous record
isstart (ISKEEPLOCK)	no lock effect	no lock effect	no lock effect
all others	no lock effect	no lock effect	unlocks previous record

Chapter 6

C and COBOL Examples

This chapter discusses the creation and manipulation of ISAM files through C and COBOL language examples. These examples are based on a very simple personnel system. The goal of the personnel system is to keep up to date information on employees. This information includes the names, addresses, job titles and salary histories for all employees.

The personnel system consists of two files, the *employee* file and the *performance* file. The *employee* file holds personal information about each employee. Each record holds the employee number, name and address for a single worker. The *performance* file holds information pertaining to each job performance review an employee has had. There is one record for each performance review, job title change or salary change an employee has had. For every employee record in the *employee* file there may be many records in the *performance* file. The field definitions for the records in both the *employee* and *performance* files are shown below:

EMPLOYEE FILE DEFINITION

FIELD NAME	LOCATION IN RECORD	TYPE
Employee number	0 - 3	**LONGTYPE**
Last name	4 - 23	**CHARTYPE**
First name	24 - 43	**CHARTYPE**
Address	44 - 63	**CHARTYPE**
City	64 - 83	**CHARTYPE**

PERFORMANCE FILE DEFINITION

FIELD NAME	LOCATION IN RECORD	TYPE
Employee number	0 - 3	**LONGTYPE**
Review date	4 - 9	**CHARTYPE**
Job rating	10	**CHARTYPE**
Salary after review	11 - 18	**DOUBLETYPE**
Title after review	19 - 48	**CHARTYPE**
Department number	49 - 50	**INTTYPE**
Job code	51 - 54	**LONGTYPE**

In COBOL, the file's definition including the definition of the primary and secondary indexes in the file-control-entry and FILE SECTION is:

```
SELECT EMP-FILE ASSIGN TO "employee"
    ORGANIZATION IS INDEXED
    ACCESS MODE IS DYNAMIC
    RECORD KEY IS EMP-NO
    ALTERNATE RECORD KEY IS LAST-NAME WITH DUPLICATES
    FILE STATUS IS EMP-FILE-STATUS.

SELECT PERF-FILE ASSIGN TO "perform"
    ORGANIZATION IS INDEXED
    ACCESS MODE IS DYNAMIC
    RECORD KEY IS EMP-REVIEW
    ALTERNATE RECORD KEY IS SALARY WITH DUPLICATES
    FILE STATUS IS PERF-FILE-STATUS.

FD  EMP-FILE.
01  EMP-RECORD.
    03 EMP-NO                               PIC X(4).
    03 LAST-NAME                            PIC X(20).
    03 FIRST-NAME                           PIC X(20).
    03 ADDRESS                              PIC X(20).
    03 CITY                                 PIC X(20).

FD  PERF-FILE.
01  PERF-RECORD.
    03 EMP-REVIEW.
       05 EMP-NR                            PIC X(4).
       05 REV-DATE                          PIC X(6).
    03 JOB-RATE                             PIC X.
    03 SALARY                               PIC X(8).
    03 TITLE                                PIC X(30).
    03 DEPARTMENT                           PIC S9(4) BINARY.
    03 JOB-CODE                             PIC S9(9) BINARY.
```

6.1 ACCESSING THE SAME FILE USING COBOL AND C (PROGRAMMING GUIDELINES)

The ISAM files can be handled both via the C calling interface and via COBOL, if it is possible to exchange the COBOL run-time for indexed file handling by an X/Open compatible ISAM. In order to access the files and the data via COBOL, the following requirements must be met that are imposed by the **ANS X3.23** standard:

1. COBOL keys are required to be of **CHARTYPE** and may not consist of more than one part. If the key-parts are adjacent, this condition is easily met. However, constructing a key that consists of two adjacent parts via C and using the key with one part defined via COBOL may result in errors. All keys created via ISAM function calls that are to be used via COBOL must meet these requirements.
2. The COBOL language requires an indexed file to have a primary key, and the key must be defined to have unique values. Hence the key defined as a parameter of the *isbuild*( ) call should not specify ISDUPS, and *k_nparts* should not be set to 0.
3. The ISAM **INTTYPE** corresponds to PIC S9(4) BINARY in COBOL and the ISAM **LONGTYPE** corresponds to PIC S9(9) BINARY in COBOL. In COBOL-74 either a non-portable type is provided or COMP is defined to be identical to BINARY of COBOL-85. These COBOL types should be used to define integer data in the record. All other positions in the record should be of **CHARTYPE** which corresponds to PIC X(n) in COBOL.

Note that the COBOL file definition has been changed to meet the above restrictions. The file definition used in **Section 6.2**, **C Program Examples** does not reflect the COBOL requirements in order to be able to illustrate additional ISAM features.

6.2 C PROGRAM EXAMPLES

6.2.1 Building a File

The following C language example creates both the *employee* and the *performance* files. It is important to note that the primary keys must be defined for every file created.

```
#include <isam.h>

#define SUCCESS 0

struct keydesc key;
int cstart, nparts;
int cc, fdemploy, fdperform;

/*
  This program builds the file systems for the
  data files employees and performance.
*/

main()
{
     mkemplkey();
     fdemploy = cc = isbuild("employee", 84, &key, ISINOUT+ISEXCLLOCK);
     if (cc < SUCCESS)
     {
          printf("isbuild error %d for %s\n",
                    iserrno, "employee file");
          exit(1);
     }
     isclose(fdemploy);

     mkperfkey();
     fdperform = cc = isbuild("perform", 55, &key, ISINOUT+ISEXCLLOCK);
     if (cc < SUCCESS)
     {
          printf("isbuild error %d for %s\n",
                    iserrno, "performance file");
          exit(1);
     }
     isclose(fdperform);
}
```

```
mkemplkey()
{
        key.k_flags = ISNODUPS;
        key.k_nparts = 0;
        cstart = 0;
        nparts = 0;

        addpart(&key, LONGSIZE, LONGTYPE);
}

mkperfkey()
{
        key.k_flags = COMPRESS;
        key.k_nparts = 0;
        cstart = 0;
        nparts = 0;

        addpart(&key, LONGSIZE, LONGTYPE);
        addpart(&key, 6, CHARTYPE);
}

addpart(keyp, len, type)
register struct keydesc *keyp;
int len;
int type;
{
        keyp-> k_part[nparts].kp_start = cstart;
        keyp-> k_part[nparts].kp_leng = len;
        keyp-> k_part[nparts].kp_type = type;
        keyp-> k_nparts = ++nparts;
        cstart += len;
}
```

6.2.2 Adding Secondary Indexes

Often the indexes defined to be primary indexes are not adequate for some applications. In the case of this application, two secondary indexes are desirable, an index on the *last name* field in the *employee* file and an index on the *salary* field in the *performance* file. The following program creates these two indexes. It is important to note that while adding indexes, the file must be opened with an exclusive lock. Exclusive file locks are specified in the *mode* parameter of the *isopen*() call by initialising that parameter to *ISINOUT+ISEXCLLOCK*. The ISINOUT specifies that the file is to be opened for both input and output, and the ISEXCLLOCK constant added to ISINOUT indicates that the file is to be exclusively locked for the current process and that no other process will be allowed to access this file. Note also that duplicates are to be allowed for both secondary indexes and that *last name* is to have full compression for its values stored in the index file.

```
#include <isam.h>

#define SUCCESS 0

struct keydesc key;
int cstart, nparts;
int fdemploy, fdperform;

/*
  This program adds secondary indexes for the last name
  field in the employee file, and the salary field in
  the performance file.
*/

main()
{
     int cc;

     fdemploy = cc = isopen("employee", ISINOUT+ISEXCLLOCK);
     if (cc < SUCCESS)
     {
          printf("isopen error %d %s\n",
                    iserrno, "for employee file");
          exit(1);
     }
```

```
        mklnamekey();
        cc = isaddindex(fdemploy, &key);
        if (cc != SUCCESS)
        {
            printf("isaddindex error %d for %s\n",
                    iserrno, "employee last-name key");
            isclose(fdemploy);
            exit(1);
        }
        isclose(fdemploy);

        fdperform = cc = isopen("perform", ISINOUT+ISEXCLLOCK);
        if (cc < SUCCESS)
        {
            printf("isopen error %d for %s\n",
                    iserrno, "performance file");
            exit(1);
        }

        mksalkey();
        cc = isaddindex(fdperform, &key);
        if (cc != SUCCESS)
        {
            printf("isaddindex error %d for %s\n",
                    iserrno, "perform salary key");
            isclose(fdperform);
            exit(1);
        }
        isclose(fdperform);
}
```

```
mklnamekey()
{
        key.k_flags = ISDUPS + COMPRESS;
        key.k_nparts = 0;
        cstart = 4;
        nparts = 0;

        addpart(&key, 20, CHARTYPE);
}

mksalkey()
{
        key.k_flags = ISDUPS;
        key.k_nparts = 0;
        cstart = 11;
        nparts = 0;

        addpart(&key, DOUBLESIZE, DOUBLETYPE);
}

addpart(keyp, len, type)
register struct keydesc *keyp;
int len;
int type;
{
        keyp-> k_part[nparts].kp_start = cstart;
        keyp-> k_part[nparts].kp_leng = len;
        keyp-> k_part[nparts].kp_type = type;
        keyp-> k_nparts = ++nparts;
        cstart += len;
}
```

6.2.3 Adding Data

The following program simply adds records to the *employee* file by prompting standard input for values of the fields in the data record. Note that the *employee* file is opened with the ISOUTPUT flag as its *mode* parameter.

```
#include <isam.h>
#include <stdio.h>

#define SUCCESS 0
#define TRUE 1
#define FALSE 0

char emprec[84];
char perfrec[55];
char line[80];
long empnum;
long jcod;
short dept;

struct keydesc key;
int cstart, nparts;
int fdemploy, fdperform;
int finished = FALSE;

/*
  This program adds a new employee record to the employee
  file.  It also adds that employee's first employee
  performance record to the performance file.
*/
```

```
main()
{
    int cc;

    fdemploy = cc = isopen("employee", ISMANULOCK + ISOUTPUT);
    if (cc < SUCCESS)
    {
        printf("isopen error %d %s\n",
                iserrno, "for employee file");
        exit(1);
    }
    fdperform = cc = isopen("perform", ISMANULOCK + ISOUTPUT);
    if (cc < SUCCESS)
    {
        printf("isopen error %d %s\n",
               iserrno, "for performance file");
        exit(1);
    }
    getemployee();

    while(!finished)
    {
        addemployee();
        getemployee();
    }
    isclose(fdemploy);
    isclose(fdperform);
}
```

```
getperform()
{
    double new_salary;

    if (empnum == 0)
    {
        finished = TRUE;
        return(0);
    }
    stlong(empnum, perfrec);

    printf("Start Date: ");
    fgets(line, 80, stdin);
    stchar(line, perfrec+4, 6);

    stchar("g", perfrec+10, 1);

    printf("Starting salary: ");
    fgets(line, 80, stdin);
    sscanf(line, "%lf", &new_salary);
    stdbl(new_salary, perfrec+11);

    printf("Title : ");
    fgets(line, 80, stdin);
    stchar(line, perfrec+19, 30);

    printf("Department number: ");
    fgets(line, 80, stdin);
    sscanf(line, "%d", &dept);
    stint(dept, perfrec+49);

    printf("Job code: ");
    fgets(line, 80, stdin);
    sscanf(line, "%ld", &jcod);
    stlong(jcod, perfrec+51);

    printf("\n\n\n");
}
```

```
addemployee()
{
    int cc;

    cc = iswrite(fdemploy, emprec);
    if (cc != SUCCESS)
    {
        printf("iswrite error %d %s\n",
                iserrno, "for employee");
        isclose(fdemploy);
        exit(1);
    }
}


addperform()
{
    int cc;

    cc = iswrite(fdperform, perfrec);
    if (cc != SUCCESS)
    {
        printf("iswrite error %d %s\n",
                iserrno, "for performance");
        isclose(fdperform);
        exit(1);
    }
}
```

```
getemployee()
{
    printf("Employee number (enter 0 to exit): ");
    fgets(line, 80, stdin);
    sscanf(line, "%ld", &empnum);
    if (empnum == 0)
    {
        finished = TRUE;
        return(0);
    }
    stlong(empnum, emprec);

    printf("Last name: ");
    fgets(line, 80, stdin);
    stchar(line, emprec+4, 20);

    printf("First name: ");
    fgets(line, 80, stdin);
    stchar(line, emprec+24, 20);

    printf("Address: ");
    fgets(line, 80, stdin);
    stchar(line, emprec+44, 20);

    printf("City: ");
    fgets(line, 80, stdin);
    stchar(line, emprec+64, 20);

    getperform();
    addperform();
    printf("\n\n\n");
}
```

6.2.4 Sequential Access

The next C language example shows how to read a file sequentially. In this particular case the *employee* file is being read in order of the primary key *employee number*. Since the *employee number* index is defined as ascending with no duplicate key values allowed, the sequence of records will print from the lowest value of *employee number* to the highest value of *employee number*. This will continue until the *isread*( ) call using ISNEXT returns the value [EENDFILE], which indicates that the end-of-file has been reached.

```
#include <isam.h>

#define SUCCESS 0
#define TRUE 1
#define FALSE 0

char emprec[84];

struct keydesc key;
int cstart, nparts;
int fdemploy, fdperform;
int eof = FALSE;

/*
   This program sequentially reads through the employee
   file by employee number, printing each record to
   stdout as it goes.
*/
```

```
main()
{
    int cc;

    fdemploy = cc = isopen("employee", ISINPUT+ISAUTOLOCK);
    if (cc < SUCCESS)
    {
        printf("isopen error %d %s\n",
                iserrno, "for employee file");
        exit(1);
    }
    mkemplkey();
    cc = isstart(fdemploy, &key, WHOLEKEY, emprec, ISFIRST);
    if (cc != SUCCESS)
    {
        printf("isstart error %d\n", iserrno);
        isclose(fdemploy);
        exit(1);
    }
    getfirst();
    while (!eof)
    {
        showemployee();
        getnext();
    }
    isclose(fdemploy);
}


showemployee()
{
    printf("Employee number: %ld", ldlong(emprec));
    printf("\nLast name: ");              putnc(emprec+4, 20);
    printf("\nFirst name: ");             putnc(emprec+24, 20);
    printf("\nAddress: ");                            putnc(emprec+44, 20);
    printf("\nCity: ");                               putnc(emprec+64, 20);
    printf("\n\n\n");
}


putnc(c, n)
char *c;
int n;
{
    while (n--) putchar(*(c++));
}
```

```
getfirst()
{
    int cc;

    if (cc = isread(fdemploy, emprec, ISFIRST))
    {
        switch(iserrno)
        {
            case EENDFILE:
                eof = TRUE;
                break;
            default:
                printf("isread ISFIRST error %d\n", iserrno);
                eof = TRUE;
                return(1);
        }
    }
    return(0);
}


getnext()
{
    int cc;

    if (cc = isread(fdemploy, emprec, ISNEXT))
    {
        switch(iserrno)
        {
            case EENDFILE:
                eof = TRUE;
                break;
            default:
                printf("isread ISNEXT error %d\n", iserrno);
                eof = TRUE;
                return(1);
        }
    }
    return(0);
}
```

```
mkemplkey()
{
        key.k_flags = ISNODUPS;
        key.k_nparts = 0;
        cstart = 0;
        nparts = 0;

        addpart(&key, LONGSIZE, LONGTYPE);
}


addpart(keyp, len, type)
register struct keydesc *keyp;
int len;
int type;
{
        keyp-> k_part[nparts].kp_start = cstart;
        keyp-> k_part[nparts].kp_leng = len;
        keyp-> k_part[nparts].kp_type = type;
        keyp-> k_nparts = ++nparts;
        cstart += len;
}
```

6.2.5 Random Access

The following program is an example of how random access to a file can be accomplished. This program interactively retrieves an employee number from standard input, searches for it in the *employee* file, and prints the results of its search to standard output.

Note that the ISEQUAL constant is used to specify the read mode to *isread*( ) in the C function called *reademp*( ). If no record corresponding to the value entered by the user is found for *employee number*, a condition code of [ENOREC] is returned by *isread*( ). It is the responsibility of the C programmer to handle that return code in an appropriate manner. If [ENOREC] is returned, the record buffer sent as the record parameter to the *isread*( ) call will not have been changed (i.e., no record will have been read).

```
#include <isam.h>
#include <stdio.h>

#define SUCCESS 0
#define TRUE 1
#define FALSE 0

char emprec[84];
char line[80];
long empnum;

struct keydesc key;
int cstart, nparts;
int fdemploy, fdperform;

/*
  This program interactively retrieves an employee's employee
  number from stdin, searches for it in the employee file,
  and prints the employee record that has that value as its
  employee number field.
*/
```

```
main()
{
    int cc;

    fdemploy = cc = isopen("employee", ISINPUT+ISAUTOLOCK);
    if (cc < SUCCESS)
    {
        printf("isopen error %d %s\n",
                iserrno, "for employee file");
        exit(1);
    }
    getempnum();
    while (empnum != 0)
    {
        if (reademp() == SUCCESS) showemployee();
        getempnum();
    }
    isclose(fdemploy);
}


getempnum()
{
    printf("Enter the employee number (0 to quit): ");
    fgets(line, 80, stdin);
    sscanf(line, "%ld", &empnum);
    stlong(empnum, emprec);
}


showemployee()
{
    printf("Employee number: %ld", ldlong(emprec));
    printf("\nLast name: ");            putnc(emprec+4, 20);
    printf("\nFirst name: ");           putnc(emprec+24, 20);
    printf("\nAddress: ");                          putnc(emprec+44, 20);
    printf("\nCity: ");                             putnc(emprec+64, 20);
    printf("\n\n\n");
}


putnc(c, n)
char *c;
int n;
{
    while (n--) putchar(*(c++));
}
```

```
reademp()
{
        int cc;

        cc = isread(fdemploy, emprec, ISEQUAL);
        if (cc != SUCCESS)
        {
            switch (iserrno)
            {
                case ENOREC:
                case EENDFILE:
                        printf("Employee doesn't exist; try again\n");
                        return(1);
                default:
                        printf("isread ISEQUAL error %d\n", iserrno);
                        exit(1);
            }
        }
        return(0);
}
```

6.2.6 Chaining

The following example shows how to chain to a record that is the last record in a chain of associated records, illustrating how the performance records appear logically by the primary key. The primary index is a composite index made up of the *employee number* and *review date.*

employee number	review date	job rating	new salary	new title
1	790501	g	20000	PA
1	800106	g	23000	PA
1	800505	f	24725	PA
2	760301	g	18000	JP
2	760904	g	20700	PA
2	770305	g	23805	PA
2	770902	g	27376	SPA
3	800420	f	18000	JP
4	800420	f	18000	JP

```
#include <isam.h>
#include <stdio.h>

#define SUCCESS 0
#define TRUE 1
#define FALSE 0

char perfrec[55];
char operfrec[55];
char line[80];
long empnum;
long jcod;
short dept;
double new_salary, old_salary;

struct keydesc key;
int cstart, nparts;
int fdemploy, fdperform;
int finished = FALSE;

/*
  This program interactively reads data from stdin and adds
  performance records to the "perform" file.  Depending on
  the rating given the employee on job performance, the
  following salary increases are placed in the salary field
  of the performance file.

                        rating        percent increase
                        p  (poor)        0.0 %
                        f  (fair)       7.5 %
                        g  (good)       13.5 %
*/
```

```
main()
{
    int cc;

    fdperform = cc = isopen("perform", ISINOUT+ISAUTOLOCK);
    if (cc < SUCCESS)
    {
        printf("isopen error %d %s\n",
                iserrno, "for performance file");
        exit(1);
    }
    mkperfkey();
    getperformance();
    while (!finished)
    {
        if (get_old_salary())
        {
            finished = TRUE;
        }
        else
        {
            addperformance();
            getperformance();
        }
    }
    isclose(fdperform);
}

addperformance()
{
    int cc;

    cc = iswrite(fdperform, perfrec);
    if (cc != SUCCESS)
    {
        printf("iswrite error %d\n", iserrno);
        isclose(fdperform);
        exit(1);
    }
}
```

```
getperformance()
{
    printf("Employee number (enter 0 to exit): ");
    fgets(line, 80, stdin);
    sscanf(line, "%ld", &empnum);
    if (empnum == 0)
    {
        finished = TRUE;
        return(0);
    }
    stlong(empnum, perfrec);

    printf("Review Date: ");
    fgets(line, 80, stdin);
    stchar(line, perfrec+4, 6);

    printf("Job rating (p = poor, f = fair, g = good): ");
    fgets(line, 80, stdin);
    stchar(line, perfrec+10, 1);

    printf("Salary After Review: ");
    printf("(Sorry, you do not get to add this)\n");
    new_salary = 0.0;
    stdbl(new_salary, perfrec+11);
    printf("Title After Review: ");
    fgets(line, 80, stdin);
    stchar(line, perfrec+19, 30);

    printf ("Department number: ");
    fgets(line, 80, stdin);
    sscanf(line, "%d", &dept);
    stint(dept, perfrec+49);

    printf("Job code: ");
    fgets(line, 80, stdin);
    sscanf(line, "%ld", &jcod);
    stlong(jcod, perfrec+51);

    printf("\n\n\n");
}
```

```
get_old_salary()
{
        int mode, cc;

        /* get employee id no. */
        bytecpy(perfrec, operfrec, 4);

        /* largest possible date */
        bytecpy("999999", operfrec+4, 6);

        cc = isstart(fdperform, &key,
        WHOLEKEY, operfrec, ISGTEQ);
        if (cc != SUCCESS)
        {
                switch(iserrno)
                {
                      case ENOREC:
                      case EENDFILE:
                             mode = ISLAST;
                             break;
                      default:
                             printf("isstart error %d\n", iserrno);
                             return(1);
                }
        }
        else
        {
                mode = ISPREV;
        }
        cc = isread(fdperform, operfrec, mode);
        if (cc != SUCCESS)
        {
                printf("isread error %d %s\n",
                            iserrno, "in get_old_salary");
                return(1);
        }
        if (cmpnbytes(perfrec, operfrec, 4))
        {
                printf("%s for employee number %ld\n",
                        "No performance record", ldlong(perfrec));
                return(1);
        }
        else
        {
                printf("\nPerformance record found.\n\n");
                old_salary = new_salary = lddbl(operfrec+11);
                printf("Rating: ");
                switch(*(perfrec+10))
                {
                    case 'p':
                        printf("poor\n");
                        break;
                    case 'f':
                        printf("fair\n");
                        new_salary *= 1.075;
                        break;
                    case 'g':
                        printf("good\n");
                        new_salary *= 1.135;
                        break;
                    default:
                        printf("no rating available\n");
                        break;
                }
                stdbl(new_salary, perfrec+11);
                printf("Old salary was %f\n", old_salary);
                printf("New salary is %f\n", new_salary);
                return(0);
        }
}

bytecpy(src,dest,n)
register char *src;
register char *dest;
register int n;
{
        while (n-- > 0)
        {
            *dest++ = *src++;
        }
}
```

```
cmpnbytes(byte1, byte2, n)
register char *byte1, *byte2;
register int n;
{
        if (n <= 0) return(0);
        while (*byte1 == *byte2)
        {
              if (--n == 0) return(0);
              ++byte1;
              ++byte2;
        }
        return(((*byte1 & BYTEMASK) <
                     (*byte2 & BYTEMASK)) ? -1 : 1);
}


mkperfkey()
{
        key.k_flags = COMPRESS;
        key.k_nparts = 0;
        cstart = 0;
        nparts = 0;

        addpart(&key, LONGSIZE, LONGTYPE);
        addpart(&key, 6, CHARTYPE);
}


addpart(keyp, len, type)
register struct keydesc *keyp;
int len;
int type;
{
        keyp->k_part[nparts].kp_start = cstart;
        keyp->k_part[nparts].kp_leng = len;
        keyp->k_part[nparts].kp_type = type;
        keyp->k_nparts = ++nparts;
        cstart += len;
}
```

6.3 COBOL PROGRAM EXAMPLES

For COBOL, two of the preceding examples will be given: Adding Data (see example in **Section 6.2.3** for analogy) and Sequential Access (see example in **Section 6.2.4** for analogy). Only the relevant parts of the programs are shown.

Building the file and creating the primary and alternate record keys is done automatically by opening a non-existent file (see examples in **Section 6.2.1**, **Building a File** and **Section 6.2.2**, **Adding Secondary Indexes**).

The error handling in COBOL would make use of the following declarations and piece of code:

```
01  EMP-FILE-STATUS                          PIC XX.
    88 EOF                                   VALUE "10".
01  PERF-FILE-STATUS                         PIC XX.
    88 EOF                                   VALUE "10".
01  iserrno                                  PIC S9(9) COMP-5 EXTERNAL.

DECLARATIVES.
EMP-DECL SECTION.
    USE AFTER EXCEPTION PROCEDURE ON EMP-FILE.
EMP-PAR.
    DISPLAY "Error " EMP-FILE-STATUS
           " on employee file".
    IF EMP-FILE-STATUS (1:1) EQUAL "9"
      THEN DISPLAY "isam code " iserrno
    END-IF.
    MOVE 1 TO RETURN-CODE.
    STOP RUN.
PERF-DECL SECTION.
    USE AFTER EXCEPTION PROCEDURE ON PERF-FILE.
PERF-PAR.
    DISPLAY "Error " PERF-FILE-STATUS
           " on performance file".
    IF PERF-FILE-STATUS (1:1) EQUAL "9"
       THEN DISPLAY "isam code " iserrno
    END-IF.
    MOVE 1 TO RETURN-CODE.
    STOP RUN.
END DECLARATIVES.
```

The following local data is used in the examples:

```
01  LOC-EMP-NO
    88 FINISHED
01  EMP-NO-EDITED                                              PIC Z(4).
01  LOC-SALARY
```

6.3.1 Adding Data

```
    OPEN I-O EMP-FILE, PERF-FILE.
    PERFORM ADD-EMPLOYEE WITH TEST AFTER UNTIL FINISHED.
    CLOSE EMP-FILE, PERF-FILE.
    STOP RUN.

ADD-EMPLOYEE SECTION.
ADD-PAR.
    DISPLAY "Employee number (enter 0 to exit): "
            WITH NO ADVANCING.
    ACCEPT LOC-EMP-NO.
    IF FINISHED GO TO ADD-EXIT.
    MOVE LOC-EMP-NO TO EMP-NO.
    DISPLAY "Last name: " WITH NO ADVANCING.
    ACCEPT LAST-NAME.
    DISPLAY "First name: " WITH NO ADVANCING.
    ACCEPT FIRST-NAME.
    DISPLAY "Address: " WITH NO ADVANCING.
    ACCEPT ADDRESS.
    DISPLAY "City: " WITH NO ADVANCING.
    ACCEPT CITY.
    WRITE EMP-RECORD.
    PERFORM ADD-PERFORM.
ADD-EXIT.  EXIT.

ADD-PERFORM SECTION.
ADD-PAR.
    MOVE LOC-EMP-NO TO EMP-NR.
    DISPLAY "Start date: " WITH NO ADVANCING.
    ACCEPT REV-DATE.
    MOVE "g" TO JOB-RATE.
    DISPLAY "Starting Salary: " WITH NO ADVANCING.
    ACCEPT LOC-SALARY.
    MOVE LOC-SALARY TO SALARY
    DISPLAY "Title: " WITH NO ADVANCING.
    ACCEPT TITLE.
    DISPLAY "Department Number: " WITH NO ADVANCING.
    ACCEPT DEPARTMENT.
    DISPLAY "Job Code: " WITH NO ADVANCING.
    ACCEPT JOB-CODE.
    WRITE PERF-RECORD.
```

6.3.2 Sequential Access

```
OPEN INPUT EMP-FILE.
MOVE LOW-VALUE TO EMP-NO.
START EMP-FILE KEY IS NOT LESS EMP-NO.
PERFORM WITH TEST AFTER UNTIL EOF OF EMP-FILE-STATUS
  READ EMP-FILE NEXT RECORD
    AT END CONTINUE;
    NOT AT END MOVE EMP-NO TO EMP-NO-EDITED
      DISPLAY "Employee number: " EMP-NO-EDITED
      DISPLAY "Last name: " LAST-NAME
      DISPLAY "First name: " FIRST-NAME
      DISPLAY "Address: " ADDRESS
      DISPLAY "City: " CITY
  END-READ
END-PERFORM.
CLOSE EMP-FILE.
STOP RUN.
```

Chapter 7

Exception Handling

Calls to ISAM functions generally return a value of 0 to indicate success or -1 to indicate some kind of exception. In the latter case, the global **int** *iserrno* is set to a meaningful value to define the nature of the condition. When testing return values in *iserrno*, it is recommended that the symbolic names defined in **<isam.h>** be used, rather than absolute values. The global **char**s *isstat1*, *isstat2*, *isstat3* and *isstat4*, that are characters, are always set to meaningful values.

7.1 ISAM CODES

ISAM codes indicate the following:

NAME	TEXT	COBOL I-O STATUS			
		ISSTAT1	ISSTAT2	ISSTAT3	ISSTAT4
[EDUPL]	An attempt was made to add a duplicate value (via *isrewcurr*( ), *isrewrec*( ), *isrewrite*( ), *iswrcurr*( ) or *iswrite*( )) to an index with no duplicates allowed, or to add an index (via *isaddindex*( )) with no duplicates allowed, while there are duplicate values for that key in the file.	2	2	2	2
[ENOTOPEN]	An attempt was made to perform some operation on a file that was not previously opened using the *isopen*( ) or *isbuild*( ) call or that was not opened with the appropriate access mode for this function call.	9		4 4 4 4 9	2 7 8 9
[EBADARG]	One of the arguments of the call is not within the range of acceptable values for that argument, or the value of the global integer *isreclen* is outside the range of acceptable values.	9		3 3 4 9	7 9 4
[EBADKEY]	One or more of the elements that make up the key description is outside the range of acceptable values for that element.	9		9	

NAME	TEXT	COBOL I-O STATUS			
		ISSTAT1	ISSTAT2	ISSTAT3	ISSTAT4
[ETOOMANY]	The maximum number of files that may be open at one time would be exceeded if this request were processed.	9		9	
[EBADFILE]	The format of the file has been corrupted.	9		9	
[ENOTEXCL]	In order to add or delete an index, the file must have been opened with exclusive access.	9		9	
[ELOCKED]	The record requested by this call cannot be accessed because either it or the entire file has been locked by another user.	9		9	
[EKEXISTS]	An attempt was made to add an index that has been defined previously.	9		9	
[EPRIMKEY]	An attempt was made to delete the primary key value. The primary key may not be deleted by the *isdelindex*( ) call.	9		9	
[EENDFILE]	The beginning or end-of-file was reached.	1	0	1 4	0 6
[ENOREC]	No record could be found that contained the requested value in the specified position or identified by *isrecnum*.	2	3	2	3

NAME	TEXT	COBOL I-O STATUS			
		ISSTAT1	ISSTAT2	ISSTAT3	ISSTAT4
[ENOCURR]	This call must operate on the current record. One has not been defined or the current record has been deleted.	2	1	4 4	3 6
[EFLOCKED]	The file is exclusively locked by another user.	9		9	
[EFNAME]	The filename is too long or otherwise invalid.	9		9	
[EBADMEM]	Adequate memory cannot be allocated for execution of the function call.	9		9	

For the meaning of the values in the columns of the COBOL I-O Status see **Section 7.2**, **Isstat1 and Isstat2 Codes**, and **Section 7.3**, **Isstat3 and Isstat4 Codes**.

For system call errors *iserrno* is set to the error code returned by the system and *isstat1* is set to 9. In this case *isstat3* is set to 9 except for the error conditions that are described by the entries 3 5 and 3 7 in **Section 7.3**, **Isstat3 and Isstat4 Codes**, that should set *isstat3* and *isstat4* to 3 and 5, or 3 and 7, respectively.

7.2 ISSTAT1 AND ISSTAT2 CODES

Two global **char**s *isstat1* and *isstat2* are used to hold status information after calls. They are related in the following way. The first one holds status information of a general nature, such as success or failure of a call. The second one contains more specific information that has meaning based on the status code in *isstat1*. The values conform to the I-O status values defined for COBOL by the **IS 1989:1974** (identical to **ANS X3.23-1974**) standard. The values are:

isstat1

0	Successful Completion
1	End of File
2	Invalid Key
9	Implementor-defined Errors

isstat2

When *isstat1* is:	*isstat2* indicates:	
0 - 9	0	No further information is available
0	2	Duplicate key indicator: — After an *isread*( ) this indicates that the key value for the current key is equal to the value of that same key in the next record. If the ISPREV mode is specified, it applies to the next record to be read with that mode. — After an *iswrite*( ) or *isrewrite*( ) this indicates that the record just written created a duplicate key value for at least one alternate record key for which duplicates are allowed.
2	1	This call must operate on the current record. One has not been defined, or the current record has been deleted.
	2	An attempt has been made to write or rewrite a record that would create a duplicate key in an indexed file.
	3	No record with the specified key can be found.
9		The value of *isstat2* is implementor-defined.

The relation between *isstat1* and *isstat2* and the ISAM codes is given in the table in **Section 7.1**, **ISAM Codes**.

7.3 ISSTAT3 AND ISSTAT4 CODES

Two global **char**s *isstat3* and *isstat4* are used to hold status information after calls. The values conform to the I-O status values defined for COBOL by the **IS 1989:1985** (identical to **ANS X3.23-1985)** standard. The values are:

isstat3	*isstat4*	
0	0	Successful completion and no further information is available.
0	2	Successful completion, but a duplicate key is detected: — After an *isread*() this indicates that the key value for the current key is equal to the value of that same key in the next record. If the ISPREV mode is specified, it applies to the next record to be read with that mode. — After an *iswrite*() or *isrewrite*() this indicates that the record just written created a duplicate key value for at least one alternate record key for which duplicates are allowed.
1	0	At end condition with unsuccessful completion.
2	2	An attempt has been made to write or rewrite a record that would create a duplicate key for a key that does not allow duplicate values.
2	3	No record with the specified key can be found.
3	5	A permanent error exists because the *filename* specified in the *isopen*() function is not present.
3	7	A permanent error exists because the *mode* parameter specified in the *isopen*() function is not allowed for the file.
3	9	A permanent error exists because a conflict has been detected between the fixed file attributes and the *mode* parameter specified in the *isopen*() function.
4	2	The *isclose*() function is attempted for a file that is not open.
4	3	This call must operate on the current record. One has not been defined, or the current record has been deleted.
4	4	An attempt is made to write or rewrite a record that is larger or smaller than is allowed on the file.
4	6	The *isread*() function with option ISNEXT is attempted and no valid next record has been established because there is no current record defined, or the previous *isread*() caused an at end condition.

isstat3	*isstat4*	
4	7	The *isread*() or *isstart*() function is attempted on a file not opened with *mode* ISINPUT or ISINOUT.
4	8	The *iswrite*() or *iswrcurr*() function is attempted on a file not opened with *mode* ISOUTPUT or ISINOUT.
4	9	The *isdelete*(), *isdelrec*(), *isdelcurr*(), *isrewrite*(), *isrewrec*() or *isrewcurr*() function is attempted on a file not opened with *mode* ISINOUT.
9		Implementor-defined errors: the value of *isstat4* is implementor-defined.

The relation between *isstat3* and *isstat4* and the ISAM codes is given in **Section 7.1**, **ISAM Codes**.

Chapter 8

The <isam.h> Header

This chapter defines the minimum contents of the header **<isam.h>**. The header contains definitions that are used for the mode arguments and also definitions of structures that are used in the calls. The structures may contain additional fields.

The numeric values shown in the definitions are purely illustrative; actual values are implementor-defined. Other additional options may be provided by an implementation. If additional bits are set this will have no effect to a portable application, i.e., will not be treated as an error. However, the access and lock modes used by *isopen*( ) and *isbuild*( ), and the position and lock modes used by *isread*( ) and *isstart*( ), should not overlap. Only the names and types of the functions are defined. They might be replaced by macro definitions. Definitions that specify limits ({NPARTS} and {MAXKEYSIZE}) give the limit that can be assumed by applications for full portability across X/Open machines. There will be at least that number on a given system, although there may in fact be more.

For example, {NPARTS} gives the maximum number of key parts, and it is set to 8. This means that all X/Open systems will allow at least 8 key parts. It also means that, for full portability, an application should not require more than this number. A particular X/Open machine may allow more than 8 and, on that system, the definition will be set to a higher value. However, applications relying on this higher value are not guaranteed to be portable.

The **<isam.h>** header:

```
#define CHARTYPE        0
#define CHARSIZE        1

#define INTTYPE         1
#define INTSIZE         2

#define LONGTYPE        2
#define LONGSIZE        4

#define DOUBLETYPE      3
#define DOUBLESIZE      (sizeof(double))

#define FLOATTYPE       4
#define FLOATSIZE       (sizeof(float))

#define MAXTYPE         5
#define ISDESC          0x80            /* add to make */
                                        /* descending type */
#define TYPEMASK        0x7F            /* type mask */

#define BYTEMASK        0xFF            /* mask for one byte */
#define BYTESHFT        8               /* shift for one byte */

/* the following might also be macro definitions */
short ldint();
long ldlong();
double ldfloat();
double lddbl();

#define ISFIRST         0               /* first record */
#define ISLAST          1               /* last record */
#define ISNEXT          2               /* next record */
#define ISPREV          3               /* previous record */
#define ISCURR          4               /* current record */
#define ISEQUAL         5               /* equal value */
#define ISGREAT         6               /* greater value */
#define ISGTEQ          7               /* >= value */


/* isread lock modes */
#define ISLOCK         0x100     /* lock record before reading */
#define ISSKIPLOCK     0x1000    /* advance record pointer to locked record */
```

```
/* isopen, isbuild lock modes */
#define ISAUTOLOCK      0x200       /* automatic record lock */
#define ISMANULOCK      0x400       /* manual record lock */
#define ISEXCLLOCK      0x800       /* exclusive isam file lock */

/* isopen, isbuild file types */
#define ISVARLEN        0x10000     /* variable length records */
#define ISFIXLEN        000         /* fixed length records only */

/* isstart lock mode for automatic record locking */
#define ISKEEPLOCK      0x2000      /* keep record locked */

#define    ISINPUT        0      /* open for input only */
#define    ISOUTPUT       1      /* open for output only */
#define    ISINOUT        2      /* open for input and output */

#define    MAXKEYSIZE     120    /* max number of bytes in key */
#define    NPARTS         8      /* max number of key parts */


struct keypart
{
                short kp_start;        /* starting byte of key part */
                short kp_leng;         /* length in bytes */
                short kp_type;         /* type of key part */
};


struct keydesc
{
                short k_flags;         /* flags */
                short k_nparts;        /* number of parts in key */
                struct keypart
                k_part[NPARTS];        /* each key part */
};


#define k_start     k_part[0].kp_start
#define k_leng      k_part[0].kp_leng
#define k_type      k_part[0].kp_type

#define ISNODUPS      000     /* no duplicates and no */
                              /* compression allowed */
#define ISDUPS        001     /* duplicates allowed */
#define COMPRESS      016     /* full compression */
```

```
struct dictinfo
{
                short di_nkeys;         /* number of keys defined */
                                        /* msb set for variable length files */
                short di_recsize;       /* (maximum) data record size */
                short di_idxsize;       /* index record size */
                long di_nrecords;       /* number of records */


/* ISAM errors must not overlap with system call error numbers */
#define EDUPL           100     /* duplicate record */
#define ENOTOPEN        101     /* file not open */
#define EBADARG         102     /* illegal argument */
#define EBADKEY         103     /* illegal key desc */
#define ETOOMANY        104     /* too many files open */
#define EBADFILE        105     /* bad ISAM file format */
#define ENOTEXCL        106     /* non-exclusive access */
#define ELOCKED         107     /* record locked */
#define EKEXISTS        108     /* key already exists */
#define EPRIMKEY        109     /* is primary key */
#define EENDFILE        110     /* end/begin of file */
#define ENOREC          111     /* no record found */
#define ENOCURR         112     /* no current record */
#define EFLOCKED        113     /* file locked */
#define EFNAME          114     /* filename too long */
#define EBADMEM         116     /* cannot allocate memory */

/*
 * For system call errors
 *   iserrno = errno (system error code 1-99)
 */

extern int iserrno;             /* isam error return code */
extern long isrecnum;           /* record number of last call */
extern int isreclen;            /* actual record length or */
                                /* minimum (isbuild, isindexinfo) */
                                /* or maximum (isopen) */
extern char isstat1;            /* COBOL-74 status characters */
extern char isstat2;
extern char isstat3;            /* COBOL-85 status characters */
extern char isstat4;
```

Chapter 9

General Information

This chapter contains general information that is relevant for the detailed description of the X/Open ISAM functions in **Chapter 10**, **ISAM Function Definitions**.

9.1 RETURN VALUE/EXCEPTION REPORTING

Most calls return either a 0 or a -1 as the value of the function and set the global integer *iserrno* either to 0 or to an error indicator. In the case of *isbuild*( ) or *isopen*( ), the return value will be a legal file descriptor (which is not an XSI file descriptor) or a -1. A -1 indicates that an error has occurred and *iserrno* has been set. The *iserrno* variable is not cleared on successful calls, so it should only be tested after an error has been indicated. Also, the global characters *isstat1*, *isstat2*, *isstat3* and *isstat4* are set for the convenience of integration with COBOL. See **Chapter 7**, **Exception Handling**, for more information.

9.2 KEY STRUCTURE

The structures **keydesc** and **keypart**, defined in **<isam.h>**, are used for index definition and are further explained below:

The structure **keydesc** contains the following members:

```
short k_flags;                          /* flags */
short k_nparts;                         /* number of parts in key */
struct keypart k_part[NPARTS];          /* each key part */
```

The structure **keypart** contains the following members:

```
short kp_start;                         /* starting byte of key part */
short kp_leng;                          /* length in bytes */
short kp_type;                          /* type of key part */
```

In the **keydesc** structure, the integer *k_flags* is used to hold duplicate and compression information for the index that is being added, deleted or selected. The symbolic values that are defined in **<isam.h>** should be used to indicate the compression techniques that are desired. If more than one feature is specified, the values are logically-or'ed together. The meaning of these symbolic values is:

ISDUPS	Duplicate values are allowed for this index.
ISNODUPS	No duplicates.
COMPRESS	Full compression for this index.

One of ISDUPS and ISNODUPS must be specified. Compression is requested by the addition of COMPRESS.

The *k_nparts* integer indicates how many parts make up the index. These parts must be described in the *k_part* array of **keypart** structures. A **keypart** structure defines each part of the index individually. The number of elements in the *k_part* array should be equal to the integer value in *k_nparts*.

The elements in the **keypart** structure are used as follows. The *kp_start* field indicates the starting byte of the key part that is being defined. The *kp_leng* field is a count of the number of bytes in the part, and *kp_type* designates the data type of the part. The types allowed are defined in the header, **<isam.h>**, see **Chapter 8**, **The <isam.h> Header**. If this part of the key is in descending order, the type constant should be logically-or'ed to the ISDESC constant (defined in **<isam.h>**). For more information about creating and manipulating indexes, see **Chapter 4**, **Indexing**.

9.3 RECORD NUMBER OF LAST CALL

The record number is a unique identification of a record. It is guaranteed that a record keeps the same record number until the record is deleted or the ISAM file is closed. If the file allows variable length records, it is implementor-defined whether a record keeps the same record number after it is rewritten using a different length than the original record had. The *isrecnum* variable is a **long** integer global variable that is set to this identification following the successful completion of all record based calls. It identifies, in an implementation-dependent, shorthand way, the record just referenced. This returned value may be used as an input parameter to the *isdelrec*() and *isrewrec*() calls, and as an implicit input parameter to the *isread*() call to perform optimised deletes, updates and reads. If used to perform sequential processing, the records will be read according to their physical layout on disc, and not according to any logical key order. Note that as the actual value returned is implementation-dependent, the user should not attempt to interpret its actual value, as this could compromise portability.

The following calls set *isrecnum*:

isdelcurr( )	*isdelete*( )	*isdelrec*( )	*isread*( )
isrewcurr( )	*isrewrec*( )	*isrewrite*( )	*isstart*( )
iswrcurr( )	*iswrite*( )		

9.4 CURRENT RECORD POSITION

The current record position should not be confused with *isrecnum* (see above). The current record position is a logical pointer that allows sequential processing to be performed according to a logical key order. The mode parameters ISNEXT and ISPREV are thus always relative to this value, while ISCURR indicates that this (the current) record should be read.

The current record position is:

- either directly on a record;
- just before a record;
- just after a record, or
- undefined.

If the pointer is directly on a certain record, that record is the current record and may be accessed or manipulated with the function calls *isdelcurr*( ), *isrewcurr*( ) or *isread*( ) using mode ISCURR.

If the pointer is just before a record, the designated record cannot be manipulated directly but may be accessed with *isread*( ) modes ISCURR or ISNEXT.

If the pointer is just after a record, the designated record cannot be manipulated directly but may be accessed with *isread*( ) mode ISPREV.

If the current record position is undefined, a function call using it as input returns an error [ENOCURR].

The *isread*( ) and the *iswrcurr*( ) function calls position the current record pointer directly on a certain record. The *isstart*( ) function call positions the current record pointer just before (with mode ISFIRST, ISEQUAL, ISGREAT or ISGTEQ) or just after (with mode ISLAST) the selected record. If no record matches the selection the position of the current record pointer is undefined and the function call returns an error [ENOREC]. The *isopen*( ) function call positions the current record pointer just before any possible first record in the order of the primary index.

The difference between the positioning via *isstart*( ) and *isopen*( ) is: if using *isstart*( ), the position is set to point just before the first record at the time this function call is executed; if using *isopen*( ), the position is set to point just before any first record. So, if a new record is added (using the *iswrite*( ) function call) with a lower value for the currently selected index than the old first record, the record position remains unchanged following an *isstart*( ), but is pointing just before this new record following an *isopen*( ) call. A call to *isread*( ) (with mode ISCURR or ISNEXT) will always read the record selected at the time *isstart*( ) was called, but will always read the newest first record at read time if *isopen*( ) was called before. If *isstart*( ) is called after *isopen*( ) the rules for *isstart*( ) will be applied.

If the key value of the current record is changed for the selected index by an *isrewrec*( ) or *isrewrite*( ) function call, the current record position remains unchanged. A read next or read previous record (*isread*( ), ISNEXT or ISPREV) results in reading the next or previous record in relation to the old key value.

However *isrewcurr*( ) sets the current record position on the record with the new key value. In other words, if the key value for the selected index is changed, the current record position is set differently than for the other rewrite functions.

If the current record is deleted (by using any delete record function), the current record position remains unchanged. The current record is no longer accessible. The subsequent function calls *isdelcurr*( ), *isrewcurr*( ) or *isread*( ) mode ISCURR will return an error [ENOCURR]. A read next or read previous record (*isread*( ), ISNEXT or ISPREV) results in reading the next or previous record in relation to the deleted key value.

If the current record position points just before or after a record and that record is locked by another process, trying to read that record (without ISSKIPLOCK flag) with manual or automatic record locking results in an error [ELOCKED] and the current record position remains unchanged. If the flag ISSKIPLOCK is specified both *isrecnum* and the current record position will point to the locked record. The parameter *record* does not point to any valid data.

If the current record position is directly on a record and the next/previous record is locked by another process, trying to read that next/previous record (without ISSKIPLOCK flag) with manual or automatic record locking results in an error [ELOCKED]. Both *isrecnum* and the current record position remain unchanged. If the flag ISSKIPLOCK is specified both *isrecnum* and the current record position will point to the locked record. The parameter *record* does not point to any valid data.

The current record position is set after successful completion of the following calls:

isopen( )	*isstart*( )	(pointing just before/after a record)
isread( )	*iswrcurr*( )	(pointing directly on the record)

and used in input to:

isdelcurr( )	*isrewcurr*( )	(if pointing directly on a record)
isread( ) with modes ISCURR, ISNEXT or ISPREV		(if pointing directly on or just before/after a record)

The current record position is undefined in the following situations:

- using *isstart*( ) with modes ISEQUAL, ISGREAT or ISGTEQ and no record is found whose key value satisfies the condition;
- using *isread*( ) with modes ISEQUAL, ISGREAT or ISGTEQ and no record is found whose key value satisfies the condition;
- after an unsuccessful call to *iswrcurr*( ) returning an error different from [EDUPL],and
- using *isstart*( ) with *k_nparts=0* (physical order) with modes ISEQUAL, ISGREAT or ISGTEQ and no record is found satisfying the condition with *isrecnum*.

9.5 PHYSICAL ORDER

If building a file, setting *k_nparts=0* in the *keydesc* parameter results in a file having no primary key. If such a file is opened, or, for any file *isstart*( ) is called setting *k_nparts=0*, the file is accessed in physical order. The record number defines the key order in this case; it may be regarded as a kind of physical index. Using *isstart*( ) with modes ISFIRST/ISLAST results in setting the current record position to point just before/after the first / last valid record in physical order. With modes ISEQUAL, ISGREAT or ISGTEQ the current record position is set to point just before the record satisfying the condition for *isrecnum*.

Using modes ISEQUAL, ISGREAT or ISGTEQ causes *isread*( ) to look at *isrecnum* for determining the desired record. If using *isread*( ) with modes ISFIRST/ISLAST the corresponding record is read and the current record position is set directly on the first / last valid record in physical order. With modes ISNEXT/ISPREV the next / previous valid record in physical order will be read.

The current record position is undefined if no record is found satisfying the condition with *isrecnum* (using *isread*( ) or *isstart*( )).

9.6 ACCESS MODE

While building or opening a file with *isbuild*( ) or *isopen*( ), a *mode* parameter is used to specify the locking and the access mode for that file. See **Chapter 5**, **Locking**, for more information about the locking mode. The access modes ISINPUT, ISOUTPUT or ISINOUT are used to open the file for reading, writing or both kinds of operations. Not all ISAM function calls can be used with each access mode. The following table shows which access mode is compatible with which function call. The ISINOUT access mode implies both the ISINPUT and the ISOUTPUT mode. If a function is called while the file had been opened with an incompatible access mode the error [ENOTOPEN] is returned.

Access Mode	Any Mode	ISINPUT	ISOUTPUT	ISINOUT
Functions:	*isbuild*( )	*isread*( )	*iswrcurr*( )	*isaddindex*( )
	isclose( )	*isstart*( )	*iswrite*( )	*isdelindex*( )
	iserase( )			*isdelcurr*( )
	isindexinfo( )			*isdelete*( )
	islock( )			*isdelrec*( )
	isopen( )			*isrewcurr*( )
	isrelease( )			*isrewrec*( )
	isrename( )			*isrewrite*( )
	isunlock( )			

9.7 FILES

The two conceptual entities, data records and indexes, are stored in an implementor-defined way in file(s). The user gets transparent access to these entities by handling one logical ISAM file that is identified by a filename.

A file name must be a legal operating system filename. The maximum length of a filename is four characters shorter than the operating system limit for characters in a filename ({NAME_MAX}, see **<limits.h>**). A pathname may be used instead of a filename. It is implementor-defined whether the indexes are stored in a separate file and/or directory.

ISAM must allow an application program to have at least ten ISAM files open at one time. However, this does not guarantee that an application program can open this number of ISAM files at any one time. If the program has opened a number of other files or the operating system limit of files per process ({OPEN_MAX}, see **<limits.h>**) is exceeded, no more ISAM files can be opened.

The effects of opening a currently open file again by the same process are implementor-defined.

Access to a file is granted via *isbuild*( ) or *isopen*( ) and remains valid until *isclose*( ). It is mandatory to close ISAM files after processing has finished. Failure to do so causes unpredictable results. Closing a file releases all locks for that file and its records that are set by the current process.

9.8 RECORDS

The definition includes the optional facility to support files having records of variable length. When a file is built it is declared to contain either fixed length or variable length records. Fixed length records must contain the same number of bytes for all the records in the file. Variable length records may contain differing numbers of bytes among the records on the file. To create a file that allows variable length records, the ISVARLEN option must be added to the *mode* parameter of the *isbuild*() function call.

The maximum number of bytes is specified by the *recordlength* parameter of the *isbuild*() function call, whereas the minimum number of bytes is specified by the global integer *isreclen*.

When opening such files, the ISVARLEN option must be added to the *mode* parameter of the *isopen*() function call. If neither ISFIXLEN nor ISVARLEN is specified, ISFIXLEN is assumed.

The minimum length defines the part of the record, that must be present for all records of the file. All indexes defined on the file must occupy character position(s) in the fixed part of the record and will be defined with the same attributes for all record types in that file. This is checked by the *isaddindex*() function call.

The global integer *isreclen* is used as an implicit input parameter to the following calls:

isbuild()	(indicating the minimum record length of the file)

iswrite() iswrcurr()
isrewrite() isrewcurr()
isrewrec()
	(indicating the actual number of bytes to be written)

and is set after successful completion of:

isread()	(indicating the actual number of bytes in the record returned)
isopen()	(indicating the maximum record length supported by the file)
isindexinfo()	(indicating the minimum record length supported by the file)

The contents of *isreclen* are implementor-defined if the file does not allow variable length records.

NAME

isaddindex - add a secondary index to an ISAM file

SYNOPSIS

int isaddindex (isfd, keydesc)
int isfd;
struct keydesc ∗**keydesc;**

DESCRIPTION

The *isaddindex*( ) function is used to add a secondary index to an ISAM file. The index will be built for the file indicated by the *isfd* parameter and will be defined according to the information in the **keydesc** structure. This call will execute only if the file has been opened for exclusive access.

The number of indexes that can be added to a file is guaranteed to be not less than 15. The maximum number of parts that may be defined for an index is {NPARTS}, and the maximum number of bytes that can exist in an index is {MAXKEYSIZE} (see **Chapter 8**, **The <isam.h> Header**).

Use of this call and index use in general are explained in **Chapter 4**, **Indexing**.

RETURN VALUE

A value of -1 is returned if an error was detected and *iserrno* is set to indicate the error. Otherwise a value of 0 is returned.

ERRORS

The following ISAM errors are possible, under the conditions given:

[EBADKEY]	Error in key description; one of the elements of the key description has an unacceptable value or, in the case of a file allowing variable length records, the character positions specified in the key description conflict with the minimum record size defined for that file.
[EDUPL]	The key description specifies that no duplicates are allowed; however, there are duplicate key values in the file.
[EKEXISTS]	The index corresponding to the specified key already exists.
[ENOTEXCL]	The file was not opened in the exclusive mode, which is required for this routine.
[ENOTOPEN]	The ISAM file descriptor *isfd* does not correspond to an open ISAM file or the file has not been opened with access mode ISINOUT.

NAME

isbuild - create an ISAM file

SYNOPSIS

int isbuild (filename, recordlength, keydesc, mode)
char *filename;
int recordlength;
struct keydesc *keydesc;
int mode;

DESCRIPTION

The *isbuild*( ) function is used to create an ISAM file. Depending on the particular implementation, this call will create and initialise appropriate disc structures to contain data and indexes.

After *isbuild*( ) has completed successfully, the file will remain open for further processing. The *isbuild*( ) function returns a file descriptor that should be used in subsequent accesses to the file.

The *filename* parameter should contain a null-terminated character string which is at least four characters shorter than the longest legal operating system filename.

The *recordlength* parameter is the length of the record. Its value is the sum of the number of bytes in each field of the record. See **Chapter 3**, **Data Types**, for the length of each data type.

All ISAM files are required formally to have a primary index. The *keydesc* parameter of this call is used to specify the structure of the primary index. However, setting *k_nparts=0* means that there is actually no primary key. Additional indexes may be added later using *isaddindex*( ). See **Chapter 4**, **Indexing**, and **Chapter 6**, **C and COBOL Examples**, for more details on key definition and use. The *mode* parameter is used to indicate whether the file allows variable length records. Selecting variable length records indicates that the user wishes to indicate at each write or rewrite operation the actual number of bytes to be stored, and wishes to be informed after a successful read operation about the actual number of bytes returned. If neither ISFIXLEN nor ISVARLEN is specified, ISFIXLEN is assumed.

For a file allowing variable length records, *recordlength* indicates the maximum length of the record, whereas the minimum length is specified in the global integer *isreclen*.

The *mode* parameter is also used to specify locking information. The user has three options: manual, automatic or exclusive. Selecting the manual option indicates that the user wishes to be responsible for locking records at the appropriate times using either the *islock*( ) and *isunlock*( ) calls or the ISLOCK mode flag of the *isread*( ) call and the *isrelease*( ) function call. Selecting automatic locking indicates that the user wishes to lock each record at the time it is read and unlock each record after the next function call is made. Selection of exclusive locking will deny file access to anyone other than this process. More information about locking can be found in **Chapter 5**, **Locking**. The *mode* is specified by using the define macros that are found in the header **<isam.h>**, for which a complete listing can be found in **Chapter 8**, **The <isam.h> Header**.

Modes that are used in the *isbuild*( ) call are the arithmetic sum of a lock mode, an access mode and optionally a record type:

Lock Modes	Access Modes	Record Types
ISEXCLLOCK	ISINPUT	ISFIXLEN
ISMANULOCK	ISOUTPUT	ISVARLEN
ISAUTOLOCK	ISINOUT	

RETURN VALUE

Upon successful completion, the file descriptor is returned. Otherwise a value of -1 is returned and *iserrno* is set to indicate the error.

ERRORS

The following ISAM errors are possible, under the conditions given:

[EBADARG]	One of the parameter values specified is illegal. In particular, this error will occur if an improper value is used for the *mode* parameter.
[EBADKEY]	Error in key description; one of the elements of the key description has an unacceptable value or, in the case of a file allowing variable length records, the character positions specified in the key description conflict with the minimum record size defined for that file.
[EFNAME]	The ISAM filename is too long or otherwise invalid.
[ETOOMANY]	The maximum number of open files would be exceeded if the ISAM file were created and opened.

WARNING

If ISVARLEN is specified and the implementation does not support files having variable length records, no error might be returned, and applications depending on the setting of *isreclen* will not function correctly.

NAME

isclose - close an ISAM file

SYNOPSIS

int isclose (isfd)
int isfd;

DESCRIPTION

The *isclose*( ) function is used to close an ISAM file. Any locks that are held for the file by this file descriptor are released.

NOTE: it is mandatory to close ISAM files after processing has finished. Failure to do so could cause unpredictable results.

RETURN VALUE

A value of -1 is returned if an error was detected and *iserrno* is set to indicate the error. Otherwise a value of 0 is returned.

ERRORS

The following ISAM error is possible, under the condition given:

[ENOTOPEN] The ISAM file descriptor *isfd* does not correspond to an open ISAM file.

NAME

isdelcurr - delete current record

SYNOPSIS

int isdelcurr (isfd)
int isfd;

DESCRIPTION

The *isdelcurr*( ) function differs from *isdelete*( ) in that it deletes the current record from the file, rather than the record indicated by the primary key. The appropriate values will be deleted from each index that is defined. The current record position must be directly on a valid record for a successful execution of the *isdelcurr*( ) function call. After a record is deleted successfully it cannot be accessed again. The *isdelcurr*( ) function is useful when the primary key is not unique and the record cannot be located and deleted in one call.

The *isrecnum* variable is set to indicate the current record, the record just deleted. A read next / previous record (*isread*( ), ISNEXT or ISPREV) results in reading the next / previous record in relation to the old deleted key value. This is consistent with COBOL requirements that the file position indicator is not affected.

RETURN VALUE

A value of -1 is returned if an error was detected and *iserrno* is set to indicate the error. Otherwise a value of 0 is returned.

ERRORS

The following ISAM errors are possible, under the conditions given:

[ELOCKED]	The record or the entire file is locked by another process.
[ENOCURR]	The current record pointer has not been set, or is invalid, or the current record has been deleted by another process.
[ENOTOPEN]	The ISAM file descriptor *isfd* does not correspond to an open ISAM file or the file has not been opened with access mode ISINOUT.

NAME

isdelete - delete record specified by primary key

SYNOPSIS

int isdelete (isfd, record)
int isfd;
char *record;

DESCRIPTION

The *isdelete*( ) function deletes a record specified by a primary key from the file indicated by *isfd*. The appropriate values will also be deleted from each index that is defined. After a successful execution the deleted record can no longer be accessed. The primary index definition for the file must not allow duplicates. The primary key field in the *record* parameter is used to identify the record to be deleted, while the other fields have no significance. If the primary key index allows duplicates, then the deletion fails. In that case the deletion can be done using *isread*( ) and *isdelcurr*( ) instead.

The *isrecnum* variable is set to indicate the record just deleted, while the current record position remains unchanged. If the current record is deleted, a read next / previous record (*isread*( ), ISNEXT or ISPREV) results in reading the next / previous record in relation to the old deleted key value. This is consistent with COBOL requirements that the file position indicator is not affected.

RETURN VALUE

A value of -1 is returned if an error was detected and *iserrno* is set to indicate the error. Otherwise a value of 0 is returned.

ERRORS

The following ISAM errors are possible, under the conditions given:

[ELOCKED]	The record or the entire file has been locked by another process.
[ENOREC]	The specified record cannot be found or the file has been created without a primary key.
[ENOTOPEN]	The ISAM file descriptor *isfd* does not correspond to an open ISAM file or the file has not been opened with access mode ISINOUT.

NAME

isdelindex - remove index from an ISAM file

SYNOPSIS

int isdelindex (isfd, keydesc)
int isfd;
struct keydesc ∗keydesc;

DESCRIPTION

The *isdelindex*( ) function is used to remove an existing index. The index will be removed from the file indicated by *isfd*. The index to be removed will be defined by the information in the **keydesc** structure. All indexes may be deleted except the primary index. Attempts to delete the primary index will cause an error code -1 to be returned and the *iserrno* global integer to be set. This call will execute only if the file has been opened for exclusive access.

RETURN VALUE

A value of -1 is returned if an error was detected and *iserrno* is set to indicate the error. Otherwise a value of 0 is returned.

ERRORS

The following ISAM errors are possible, under the conditions given:

[EBADKEY]	Error in key description; one of the elements of the key description has an unacceptable value, or the index to be deleted is the current index, or no index (neither primary nor secondary) is defined for the file.
[ENOTEXCL]	The file was not opened in the exclusive mode, which is required for this routine.
[ENOTOPEN]	The ISAM file descriptor *isfd* does not correspond to an open ISAM file, or the file has not been opened with access mode ISINOUT.
[EPRIMKEY]	The primary index is specified (for deletion); it may not be deleted.

NAME

isdelrec - delete record specified by record number

SYNOPSIS

int isdelrec (isfd, recnum)
int isfd;
long recnum;

DESCRIPTION

The *isdelrec*( ) function differs from *isdelete*( ) in that it deletes the record specified by *recnum* from the file indicated by *isfd*, rather than the record indicated by the primary key. The appropriate values will be deleted from each index that is defined. The parameter *recnum* must be a previously obtained *isrecnum* value that identifies an existing record.

This call will set *isrecnum* to the value of *recnum*, while the current record position will remain unchanged. If the current record is deleted, a read next/previous record (*isread*( ), ISNEXT or ISPREV) results in reading the next/previous record in relation to the old deleted key value. This is consistent with COBOL requirements that the file position indicator is not affected.

RETURN VALUE

A value of -1 is returned if an error was detected and *iserrno* is set to indicate the error. Otherwise a value of 0 is returned.

ERRORS

The following ISAM errors are possible, under the conditions given:

[ELOCKED] The record or the entire file has been locked by another process.

[ENOREC] No record could be found using the specified *recnum*.

[ENOTOPEN] The ISAM file descriptor *isfd* does not correspond to an open ISAM file or the file has not been opened with access mode ISINOUT.

NAME

iserase - remove an ISAM file

SYNOPSIS

int iserase (filename)
char *filename;

DESCRIPTION

The *iserase*( ) function will remove the file specified by *filename*.

RETURN VALUE

A value of -1 is returned if an error was detected and *iserrno* is set to indicate the error. Otherwise a value of 0 is returned.

ERRORS

The following ISAM errors are possible, under the conditions given:

[EFNAME]	The ISAM filename is too long or otherwise invalid.
[EFLOCKED]	The ISAM file is locked by another process.

NAME

isindexinfo - access file information

SYNOPSIS

int isindexinfo (isfd, buffer, number)
int isfd;
struct keydesc ∗buffer;
/∗ buffer may be a pointer to ∗/
/∗ a **dictinfo** structure instead. ∗/
int number;

DESCRIPTION

The *isindexinfo*( ) function gives the caller access to information about the file, such as information about the defined indexes, their location within the record, their length, and whether duplicate values are allowed.

Information about a particular index is obtained by specifying the number of the index using the *number* parameter. For a file without primary index, *number* set to 1 identifies the physical index; in the information that is returned, *k_nparts* is set to 0. General information such as (maximum) data record size, indication of variable length records allowance (the most significant bit of *di_nkeys* being set), the number of indexes, index record size, and data record size is obtained by calling *isindexinfo*( ) with the *number* parameter set to 0 and reading the *buffer* into a structure of type **dictinfo**. The minimum record length is returned in the global integer *isreclen*. If the file was built without specification of a primary index, the physical index is included in the **dictinfo** information.

The *buffer* parameter can contain information in the format of either **keydesc** or **dictinfo** depending on whether the *number* parameter is positive or 0, respectively. As indexes are added and deleted, the number of a particular index may vary. To ensure review of all indexes, loop over the number of indexes indicated in **dictinfo** (see structure definitions in **Chapter 8**, **The <isam.h> Header**).

RETURN VALUE

A value of -1 is returned if an error was detected and *iserrno* is set to indicate the error. Otherwise a value of 0 is returned.

ERRORS

The following ISAM errors are possible, under the conditions given:

[EBADARG] This error will occur if an improper value is used for the *number* parameter.

[ENOTOPEN] The ISAM file descriptor *isfd* does not correspond to an open ISAM file.

NAME

islock - lock an ISAM file

SYNOPSIS

int islock (isfd)
int isfd;

DESCRIPTION

The *islock*( ) function will lock the entire file that is specified by *isfd*. More discussion of locking can be found in **Chapter 5**, **Locking**.

RETURN VALUE

A value of -1 is returned if an error was detected and *iserrno* is set to indicate the error. Otherwise a value of 0 is returned.

ERRORS

The following ISAM errors are possible, under the conditions given:

[ELOCKED]	The entire file or at least one record of the file has been locked by another process.
[ENOTOPEN]	The ISAM file descriptor *isfd* does not correspond to an open ISAM file.

NAME

isopen - open an ISAM file

SYNOPSIS

int isopen (filename, mode)
char *filename;
int mode;

DESCRIPTION

The *isopen*( ) function is used to open an ISAM file for processing. The function will return the file descriptor that should be used in subsequent accesses to the file.

This call will automatically position the current record pointer just before any possible first record in the order of the primary index. If another ordering is desired, the *isstart*( ) call can be used to select another index. More information about the difference between positioning with *isopen*( ) and *isstart*( ) can be found in **Section 9.4**, **Current Record Position**.

The *filename* parameter must contain a null-terminated string, which is the name of the file to be processed.

The *mode* parameter determines the locking information. The user has three options: manual, automatic or exclusive. Selecting the manual option indicates that the user wishes to be responsible for locking records at the appropriate times. Selecting automatic locking indicates that the user wishes to lock each record as it is read and unlock it after any subsequent function calls. Selection of exclusive locking will deny file access to anyone other than this process. More information about locking can be found in **Chapter 5**, **Locking**. The *mode* parameter also specifies whether the file is to be opened for read, write or read / write access.

If the file allows variable length records, the ISVARLEN option must be added to the *mode* parameter. In this case the global integer *isreclen* is set to indicate the maximum record length allowed for the file. If neither ISFIXLEN nor ISVARLEN is specified, ISFIXLEN is assumed.

The mode is specified by using the define macros that are found in the header **<isam.h>**, for which a complete listing can be found in **Chapter 8**, **The <isam.h> Header**. Modes that are used in the *isopen*( ) function are the arithmetic sum of a lock mode, an access mode and optionally a record type:

Lock Modes	Access Modes	Record Types
ISEXCLLOCK	ISINPUT	ISFIXLEN
ISMANULOCK	ISOUTPUT	ISVARLEN
ISAUTOLOCK	ISINOUT	

RETURN VALUE

Upon successful completion, the file descriptor is returned. Otherwise a value of -1 is returned and *iserrno* is set to indicate the error.

ERRORS

The following ISAM errors are possible, under the conditions given:

[EBADARG]	One of the parameter values specified is illegal. In particular, this error will occur if an improper value is used for the *mode* parameter or if the file type indication stored in the file conflicts with the *mode* parameter.
[EFLOCKED]	The file has been exclusively locked by another process.
[EFNAME]	The ISAM filename is too long or otherwise invalid.
[ETOOMANY]	The maximum number of open files would be exceeded if the ISAM file were opened.

WARNING

If ISVARLEN is specified and the implementation does not support files having variable length records, no error might be returned, and applications depending on the setting of *isreclen* will not function correctly.

NAME

isread - read records

SYNOPSIS

int isread (isfd, record, mode)
int isfd;
char *record;
int mode;

DESCRIPTION

The *isread*( ) function is used to read records sequentially or randomly as indicated by the *mode* parameter.

When sequential processing is desired, *mode* must specify which record is to be read. It may take one of the following values:

ISCURR	current
ISFIRST	first record
ISLAST	last record
ISNEXT	next record
ISPREV	previous record

When random selection is desired, *mode* must specify the value of the record to be returned relative to the specified search value. This value may be one of:

ISEQUAL	equal to
ISGREAT	greater than
ISGTEQ	greater than or equal to

The search value is placed in the *record* buffer in the correct byte positions.

The *isread*( ) function will fill in the *record* with the results of the search. The *mode* is specified by using the define macros that are found in the header **<isam.h>**. Refer to **Chapter 8**, **The <isam.h> Header**, for the contents of this file.

The *isread*( ) function can also read records specified by a previously set *isrecnum*. First, call *isstart*( ) with *k_nparts=0* so that the file is set to read in physical order. Then call *isread*( ) with *mode=ISEQUAL*. This will cause *isread*( ) to look at *isrecnum* for the desired record.

Following the successful execution of this call, the current record position and *isrecnum* will both be set to indicate the record just read.

If the file allows variable length records, the global integer *isreclen* is set after a successful call to indicate the number of bytes in the *record* buffer that define the actual record returned.

The contents of the bytes in the *record* buffer beyond the value of *isreclen* are implementor-defined.

If manual locking was specified when the file was opened and the record is to be locked before being read, the ISLOCK flag may be arithmetically added to one of the above macros. The record will then remain locked until unlocked with the *isrelease*( ) or *isunlock*( ) function call. Entire files may be locked and unlocked by using the *islock*( ) and *isunlock*( ) or *isrelease*( ) calls.

If *isread*( ), with flag ISLOCK and modes ISCURR, ISNEXT or ISPREV, tries to access a record locked by another process both *isrecnum* and the current record pointer remains unchanged. An error [ELOCKED] will be returned.

If *isread*( ), with flag ISLOCK and modes ISFIRST, ISLAST, ISEQUAL, ISGREAT or ISGTEQ, tries to access a record locked by another process both *isrecnum* and the current record position will point to the locked record. The *iserrno* variable will be set to [ELOCKED] and *record* will not point to any valid data.

If the ISLOCK flag is used with modes ISNEXT or ISPREV, an ISSKIPLOCK option is provided giving the capability to read past/before a locked record by two successive *isread*( ) calls. Using this option both *isrecnum* and the current record position will point to the locked record. The *iserrno* variable will be set to [ELOCKED] and *record* will not point to any valid data. Using automatic lockmode, the option ISSKIPLOCK may be used (without the ISLOCK flag being set) to achieve the same behaviour as just described for manual locking mode.

Using manual lockmode it is always possible to read a record locked by another process if no ISLOCK flag is specified. But it is not possible to lock, rewrite or delete such a record. Thus locking is an advisory and not a mandatory feature for read access within ISAM.

Modes that are used in the *isread*( ) call are the arithmetic sum of a read mode and one or both of the optional lock modes:

Read Modes	Lock Modes
ISCURR	ISLOCK
ISNEXT	ISSKIPLOCK (with ISNEXT or ISPREV)
ISPREV	
ISFIRST	
ISLAST	
ISEQUAL	
ISGREAT	
ISGTEQ	

RETURN VALUE

A value of -1 is returned if an error was detected and *iserrno* is set to indicate the error. Otherwise a value of 0 is returned.

ERRORS

The following ISAM errors are possible, under the conditions given:

[EBADARG]	One of the parameter values specified is illegal. In particular, this error will occur if an improper value is used for the *mode* parameter.
[EENDFILE]	The end of the file has been reached. This error can occur if the sequential processing modes ISNEXT or ISPREV are used; or if the modes ISFIRST or ISLAST are used while accessing an empty file.
[ELOCKED]	The record or the entire file has been locked by another process. This error can occur only if the read is done with locking; a read without locking will

not be blocked. (A read is done with locking if the file has been opened in the ISAUTOLOCK mode, or if it has been opened in the ISMANULOCK mode, and ISLOCK is used in the mode specification of *isread*( ).)

[ENOCURR] The current record pointer has not been set, or is invalid, or the current record has been deleted by another process. In the last case this error can occur only if the mode ISCURR is used.

[ENOREC] The specified record cannot be found. This error can occur only if the random access modes ISEQUAL, ISGREAT or ISGTEQ are used.

[ENOTOPEN] The ISAM file descriptor *isfd* does not correspond to an open ISAM file, or the file has been opened with access mode ISOUTPUT.

NAME

isrelease - unlock records

SYNOPSIS

int isrelease (isfd)
int isfd;

DESCRIPTION

The *isrelease*( ) function unlocks records that have been locked using the ISMANULOCK mode in the *isread*( ) call (when in manual locking mode) or the currently locked records, if any (when in automatic locking mode). All locked records in the file indicated by *isfd* will be unlocked. More information, including examples of how to use *isrelease*( ), can be found in **Chapter 5**, **Locking**.

RETURN VALUE

A value of -1 is returned if an error was detected and *iserrno* is set to indicate the error. Otherwise a value of 0 is returned.

ERRORS

The following ISAM error is possible, under the condition given:

[ENOTOPEN] The ISAM file descriptor *isfd* does not correspond to an open ISAM file.

NAME

isrename - rename an ISAM file

SYNOPSIS

int isrename (oldname, newname)
char *oldname;
char *newname;

DESCRIPTION

The *isrename*( ) function will rename the file specified by the *oldname* parameter to the name specified by the *newname* parameter.

RETURN VALUE

A value of -1 is returned if an error was detected and *iserrno* is set to indicate the error. Otherwise a value of 0 is returned.

ERRORS

The following ISAM errors are possible, under the conditions given:

[EFNAME]	The ISAM filename *oldname* or *newname* is too long, or otherwise invalid.
[EFLOCKED]	The ISAM file is locked by another process.

NAME

isrewcurr - rewrite current record

SYNOPSIS

int isrewcurr (isfd, record)
int isfd;
char *record;

DESCRIPTION

The *isrewcurr*( ) function is used to rewrite the current record of the file indicated by *isfd* with the contents of the character array *record*. All fields, including primary key fields, can be changed. Each index (primary inclusive) will be appropriately updated. The current record position must be directly on a valid record for a successful execution of the *isrewcurr*( ) function call. It is not possible to rewrite a previously deleted record.

If the file allows variable length records, the global integer *isreclen* must be set prior to the call to indicate the number of bytes in the *record* buffer that define the actual record to be written.

The *isrewcurr*( ) function is useful when the primary key is not unique and the record cannot be located and rewritten in one call. The current record position points to the new position of the record after the successful execution of this function call.

The *isrecnum* variable is set to indicate the current record. Its position is left unchanged unless the key value of the current record is changed for the selected index. In the latter case the current record position is changed to point to the record with the new key value. A read next/previous record (*isread*( ), ISNEXT or ISPREV) results in reading the next/previous record in relation to the new key value. This is not consistent with COBOL requirements that the file position indicator should not be affected. The *isrewrec*( ) or *isrewrite*( ) function calls can be used instead to fulfil that requirement.

When the value of a specific index that allows duplicates is changed, the record is logically positioned last within the set of duplicate records containing that value.

RETURN VALUE

A value of -1 is returned if an error was detected and *iserrno* is set to indicate the error. Otherwise a value of 0 is returned.

ERRORS

The following ISAM errors are possible, under the conditions given:

[EBADARG]	The value of the global integer *isreclen* is unacceptable.
[EDUPL]	The rewrite would add a duplicate value in an index, in which duplicates are not allowed.
[ELOCKED]	The record or the entire file has been locked by another process.
[ENOCURR]	The current record pointer has not been set, or is invalid, or the current record has been deleted by another process.
[ENOTOPEN]	The ISAM file descriptor *isfd* does not correspond to an open ISAM file, or the file has not been opened with access mode ISINOUT.

NAME

isrewrec - rewrite record specified by record number

SYNOPSIS

int isrewrec (isfd, recnum, record)
int isfd;
long recnum;
char *record;

DESCRIPTION

The *isrewrec*( ) function is used to rewrite the record specified by *recnum* in the file indicated by *isfd* with the contents of the character array *record*.

If the file allows variable length records, the global integer *isreclen* must be set prior to the call to indicate the number of bytes in the *record* buffer that define the actual record to be written.

The *recnum* argument must be a previously obtained *isrecnum* value that identifies an existing record. Each index (primary inclusive) will be appropriately updated.

The *isrewrec*( ) function is useful when the primary key is not unique and the record cannot be located and rewritten in one call while the current record position should remain unchanged.

The *isrecnum* variable will be set to the value of *recnum*, while the current record position will remain unchanged.

If the key value of the current record is changed for the selected index, a read next/previous record (*isread*( ), ISNEXT or ISPREV) results in reading the next/previous record in relation to the old key value. This is consistent with COBOL requirements that the file position indicator is not affected.

When the value of a specific index that allows duplicates is changed, the record is logically positioned last within the set of duplicate records containing that value. This is consistent with COBOL requirements for maintaining the order of records in duplicate chains.

RETURN VALUE

A value of -1 is returned if an error was detected and *iserrno* is set to indicate the error. Otherwise a value of 0 is returned.

ERRORS

The following ISAM errors are possible, under the conditions given:

[EBADARG]	The value of the global integer *isreclen* is unacceptable.
[EDUPL]	The rewrite would add a duplicate value in an index, in which duplicates are not allowed.
[ELOCKED]	The record or the entire file has been locked by another process.
[ENOREC]	No record could be found using the specified *recnum*.
[ENOTOPEN]	The ISAM file descriptor *isfd* does not correspond to an open ISAM file, or the file has not been opened with access mode ISINOUT.

NAME

isrewrite - rewrite record specified by primary key

SYNOPSIS

int isrewrite (isfd, record)
int isfd;
char ∗record;

DESCRIPTION

The *isrewrite*( ) function is used to change one or more values for a record that is already in a file identified by *isfd*. All fields but the primary key field can be changed. The primary index definition for the file must not allow duplicates. The *record* parameter contains the changes. The primary key field is used to identify the record to be changed, while the other fields contain the changes.

If the file allows variable length records, the global integer *isreclen* must be set prior to the call to indicate the number of bytes in the *record* buffer that define the actual record to be written.

The *isrewrite*( ) function does not change the position of the current record pointer, while *isrecnum* is set to indicate this record.

If the key value of the current record is changed for the selected index, a read next/previous record (*isread*( ), ISNEXT or ISPREV) results in reading the next/previous record in relation to the old key value. This is consistent with COBOL requirements that the file position indicator is not affected.

When the value of a specific alternate index that allows duplicates is changed, the record is logically positioned last within the set of duplicate records containing that value. This is consistent with COBOL requirements for maintaining the order of records in duplicate chains.

RETURN VALUE

A value of -1 is returned if an error was detected and *iserrno* is set to indicate the error. Otherwise a value of 0 is returned.

ERRORS

The following ISAM errors are possible, under the conditions given:

[EBADARG]	The value of the global integer *isreclen* is unacceptable.
[EDUPL]	The rewrite would add a duplicate value in an index, in which duplicates are not allowed.
[ELOCKED]	The record or the entire file has been locked by another process.
[ENOREC]	The specified record cannot be found or no primary key is defined for this file.
[ENOTOPEN]	The ISAM file descriptor *isfd* does not correspond to an open ISAM file, or the file has not been opened with access mode ISINOUT.

NAME

isstart - select an index

SYNOPSIS

int isstart (isfd, keydesc, length, record, mode)
int isfd;
struct keydesc *keydesc;
int length;
char *record;
int mode;

DESCRIPTION

The *isstart*( ) function selects the index to be used in subsequent operations. The key value to be sought should be placed in the *record* parameter, in the positions described by the *keydesc* parameter. The **keydesc** structure must describe an index that has been added previously using the *isaddindex*( ) call.

The *length* parameter is used to specify the part of the key to be considered significant when doing the search. A zero indicates that the whole key is significant; a positive value is used to indicate a shorter length. If *length* is greater than zero, the response during searches will be as if the index were originally defined to have that shorter length.

The *mode* parameter may be ISFIRST, ISLAST, ISEQUAL, ISGREAT or ISGTEQ. It is used to position the user in the file in association with the index selected by the *keydesc* argument. An additional flag ISKEEPLOCK may be arithmetically added to the *mode* macros. With automatic record locking this flag is used to enable the record lock to be kept.

ISFIRST positions the user's program in the file just before the first record in the ordering of the index specified in the *keydesc* parameter. A subsequent call to *isread*( ) using the ISNEXT *mode* parameter reads the first record in the current ordering.

ISLAST positions the user's program just after the last record in that ordering. A subsequent call to *isread*( ) using the ISPREV *mode* parameter reads the last record in the current ordering.

Note that if *mode* is ISFIRST or ISLAST, the parameters *length* and *record* are not needed and are not used by the *isstart*( ) call.

Use of the ISEQUAL, ISGREAT or ISGTEQ modes is different from the use of the ISFIRST or ISLAST modes. When using the former modes, the user's program must place the key value to be searched for in the *record* buffer before calling *isstart*( ). The value to be searched for must be placed in the location in the *record* buffer where the *keydesc* parameter claims the index exists.

ISEQUAL will give one of two possible results. It will either find a record whose key value is equal to that found in the appropriate positions of the *record* buffer parameter, or it will return an error code -1 and set *iserrno* to [ENOREC]. The error code [ENOREC] indicates that no record with the key value specified in the *record* buffer parameter exists in the file.

ISGREAT will also give one of two responses. It will either find the first record whose key value is greater than the key value specified in the appropriate positions of the *record* buffer parameter, or *isstart*( ) will return an error condition -1 and set *iserrno* to [ENOREC].

The ISGTEQ *mode* parameter finds the first record whose key value is greater than or equal to the key value specified in the appropriate positions of the *record* buffer parameter. If no such record is found, *isstart*( ) returns an error code -1 and sets *iserrno* to [ENOREC]. If *isstart*( ) returns an error code the current record position is undefined.

If ISKEEPLOCK is specified *isstart*( ) will not unlock any record lock using automatic record locking. Without this option *isstart*( ) will unlock any record if executed successfully.

The above macros, ISFIRST, ISLAST, ISEQUAL, ISGREAT and ISGTEQ and the lock option ISKEEPLOCK are defined in the header **<isam.h>**.

The *isstart*( ) function can also be used for sequential access in physical order by specifying a previously defined key that has zero parts; i.e., give a value to *keydesc* to designate a structure in which *k_nparts=0* (see *isread*( )).

The *isstart*( ) function performs two basic functions. It selects the index that is to be used for subsequent reads, and it finds (but does not read) a record in the file. The *isstart*( ) function need not be used to find each record before it is read using *isread*( ).

Following the successful execution of this call, the current record position and *isrecnum* will both be set to indicate this record. The current record position is set just before (using modes ISFIRST, ISEQUAL, ISGREAT and ISGTEQ) or just after (mode ISLAST) the record found. This means that the selected record cannot directly be manipulated by an *isdelcurr*( ) or *isrewcurr*( ) function call, but it can be read with *isread*( ) to set the current record pointer directly on the record for further manipulation.

More information about positioning the current record pointer with *isstart*( ) can be found in **Section 9.4**, **Current Record Position**.

RETURN VALUE

A value of -1 is returned if an error was detected and *iserrno* is set to indicate the error. Otherwise a value of 0 is returned.

ERRORS

The following ISAM errors are possible, under the conditions given:

[EBADARG]	One of the parameter values specified is illegal. In particular, this error will occur if an improper value is used for the *mode* parameter.
[EBADKEY]	Error in key description; one of the elements of the key description has an unacceptable value.
[ENOREC]	The specified record cannot be found. This error can occur only if the random access modes ISEQUAL, ISGREAT or ISGTEQ are used.
[ENOTOPEN]	The ISAM file descriptor *isfd* does not correspond to an open ISAM file, or the file has been opened with access mode ISOUTPUT.

NAME

isunlock - unlock an ISAM file

SYNOPSIS

int isunlock (isfd)
int isfd;

DESCRIPTION

The *isunlock*( ) function is used to release an existing file-level lock for the file specified by the file descriptor *isfd*. Further discussion of locking can be found in **Chapter 5**, **Locking**.

RETURN VALUE

A value of -1 is returned if an error was detected and *iserrno* is set to indicate the error. Otherwise a value of 0 is returned.

ERRORS

The following ISAM error is possible, under the condition given:

[ENOTOPEN] The ISAM file descriptor *isfd* does not correspond to an open ISAM file.

NAME

iswrcurr - write record and set current position

SYNOPSIS

int iswrcurr (isfd, record)
int isfd;
char *record;

DESCRIPTION

The *iswrcurr*( ) function writes the record passed to it in the *record* parameter to the data file identified by *isfd*. The appropriate values will be inserted into each index that is defined.

If the file allows variable length records, the global integer *isreclen* must be set prior to the call to indicate the number of bytes in the *record* buffer that define the actual record to be written.

Following the successful execution of this call, the current record position and *isrecnum* will both be set to indicate this record.

If the error [EDUPL] is returned the current record position and *isrecnum* remain unchanged. If other kinds of errors are returned the current record position is undefined.

RETURN VALUE

A value of -1 is returned if an error was detected and *iserrno* is set to indicate the error. Otherwise a value of 0 is returned.

ERRORS

The following ISAM errors are possible, under the conditions given:

[EBADARG]	The value of the global integer *isreclen* is unacceptable.
[EDUPL]	The write would add a duplicate value in an index, in which duplicates are not allowed.
[ELOCKED]	The file has been locked by another process.
[ENOTOPEN]	The ISAM file descriptor *isfd* does not correspond to an open ISAM file, or the file has been opened with access mode ISINPUT.

NAME

iswrite - write record

SYNOPSIS

int iswrite (isfd, record)
int isfd; char *record;

DESCRIPTION

The *iswrite*( ) function writes the record passed to it in the *record* parameter to the file. The appropriate values will be inserted into each index that is defined.

If the file allows variable length records, the global integer *isreclen* must be set prior to the call to indicate the number of bytes in the *record* buffer that define the actual record to be written.

The *iswrite*( ) function does not change the position of the current record pointer, but *isrecnum* is set to indicate this record.

RETURN VALUE

A value of -1 is returned if an error was detected and *iserrno* is set to indicate the error. Otherwise a value of 0 is returned.

ERRORS

The following ISAM errors are possible, under the conditions given:

[EBADARG]	The value of the global integer *isreclen* is unacceptable.
[EDUPL]	The write would add a duplicate value in an index, in which duplicates are not allowed.
[ELOCKED]	The file has been locked by another process.
[ENOTOPEN]	The ISAM file descriptor *isfd* does not correspond to an open ISAM file, or the file has been opened with access mode ISINPUT.

Index