Contents

Chapter	1	INTRODUCTION
Chapter	2	ISAM OVERVIEW
Chapter	3	DATA TYPES
	3.1	CHARTYPE
	3.2	INTTYPE AND LONGTYPE
	3.3	FLOATTYPE AND DOUBLETYPE
Chapter	4	INDEXING
	4.1	INDEX DEFINITION AND MANIPULATION
	4.2	INDEX COMPRESSION
Chapter	5	LOCKING
	5.1	EXCLUSIVE FILE LOCKING
	5.2	MANUAL FILE LOCKING
	5.3	RECORD LEVEL LOCKING
	5.3.1	Automatic Record Locking
	5.3.2	Manual Record Locking
Chapter	6	C PROGRAM EXAMPLES
	6.1	BUILDING A FILE
	6.2	ADDING SECONDARY INDEXES
	6.3	ADDING DATA
	6.4	SEQUENTIAL ACCESS
	6.5	RANDOM ACCESS
	6.6	CHAINING
Chapter	7	EXCEPTION HANDLING
Chapter	8	THE ISAM.H HEADER FILE

Chapter	9	CALL FORMATS AND DESCRIPTIONS	
	9.1	RETURN VALUE/EXCEPTION REPORTING	
	9.2	isam.h HEADER FILE	
	9.3	KEY STRUCTURE	
	9.4	RECORD NUMBER OF LAST CALL	
	9.5	CURRENT RECORD POSITION	
		isaddindex(ISAM)	
		isbuild(ISAM)	
		isclose(ISAM)	
		isdelcurr(ISAM)	
		isdelete(ISAM)	
		isdelindex(ISAM)	
		isdelrec(ISAM)	OPTIONAL
		iserase(ISAM)	
		isindexinfo(ISAM)	
		islock(ISAM)	
		isopen(ISAM)	
		isread(ISAM)	
		isrelease(ISAM)	
		isrename(ISAM)	
		isrewcurr(ISAM)	
		isrewrec(ISAM)	OPTIONAL
		isrewrite(ISAM)	
		isstart(ISAM)	
		isunlock(ISAM)	
		iswrcurr(ISAM)	OPTIONAL
		iswrite(ISAM)	

Chapter 1

Introduction

The input/output facilities supported by System V 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 Indexed Sequential Access Method (ISAM). The availability of this interface on X/OPEN systems will not only provide application portability, but will ease and encourage integration.

The X/OPEN ISAM definition is a major sub-set of the specification of the C-ISAM product, version 2.10, published by Relational Database Systems Inc. of 4100 Bohannon Drive, Menlo Park, California 94025.

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 systems.)

Version 2.10 of the C-ISAM product introduced four new functions, and a set of decimal data types. The new functions have been included in this first issue of the X/OPEN specification in the "optional" category. This means that although they are likely to appear on many X/OPEN systems, they are not guaranteed to be on all. Where they are supported, they will conform to the given definition. The decimal types have been excluded pending a general review of support of decimal types within X/OPEN systems.

This part of the guide 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 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 file containing definitions of various macros and symbolic constants.
Chapter 9	contains detailed specifications of the interfaces in the X/OPEN ISAM definition.

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
- 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.

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 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 is completed. Alternatively, it may be manual in that it is actioned as a result of a parameter of a read call.

The following functions are included in the X/OPEN ISAM definition.

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

⋆ These functions are optional in the X/OPEN ISAM definition.

ISAM Overview

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

FUNCTION NAME	PURPOSE
isaudit(ISAM)	performs operations on audit trail
isflush(ISAM)	flushes buffered index pages
issetuniq(ISAM)	set unique identifier
isuniqueid(ISAM)	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.

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 integer; *p* is a **char** pointer to the starting byte of format *INTTYPE*.
stint(i, p)	stores a machine-format 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 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 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 file <**isam.h**>, which must be included in any program which uses the function calls.

The structure *keydesc* contains the following members:

```
short k_flags;     /* compression and duplicates */
short k_nparts;    /* number of parts in this 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*(ISAM) 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= 0;     /* 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= 2; /* 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 initialize each of these structures to reflect the desired position, length, and data type for each index part.

The structure *keypart* contains the following members:

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

In the following example program, a 3-part index is defined. The index consists of a *CHARTYPE* field, a *LONGTYPE,* 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 {MAXKEYLEN}. There is no limit to the number of keys that can be added to a file.

```
#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    = 4;
        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 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 the X/OPEN 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. Both are built on the System V lock features. 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*(ISAM) or *isbuild*(ISAM), until the file is closed using *isclose*(ISAM), Exclusive file locking is specified by adding ISEXCLLOCK to the *mode* parameter of the *isopen*(ISAM) or *isbuild*(ISAM) function call.

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

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*(ISAM) and *isunlock*(ISAM) function calls. When a file is to be locked in this manner, ISMANULOCK must be added to the *mode* parameter of the *isopen*(ISAM) or *isbuild*(ISAM) call. Later in the program, when locking is desired, *islock*(ISAM) should be called to lock the file. When the file is to be unlocked, *isunlock*(ISAM) should be called. For example:

```
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*(ISAM) 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*(ISAM). It unlocks that record, and any other records that are currently locked, when *isrelease*(ISAM) 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, but they may not lock, re-write, or delete them.

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*(ISAM) or *isbuild*(ISAM) 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 is completed for the current file. Therefore, while using the automatic record locking mechanism, only one record per file may be locked at a given time.

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*(ISAM) or *isbuild*(ISAM) 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*(ISAM) 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*(ISAM) function. The number of records that may be locked in this manner at any one time is system dependent.

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);
```

Chapter 6

C Program Examples

This chapter discusses the creation and manipulation of ISAM files through C 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 **personnel** file there may be many records in the **performance** file. The field definitions for the records in both the **personnel** and **performance** files are shown below:

Employee File Definition

Field Name	*Location in Record*	
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*	
Employee number	0 — 3	*LONGTYPE*
Review date	4 — 9	*CHARTYPE*
Job rating	10 — 11	*CHARTYPE*
Salary after review	12 — 19	*DOUBLETYPE*
Title after review	20 — 50	*CHARTYPE*

6.1 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",
                49, &key, ISINOUT+ISEXCLLOCK);
        if (cc < SUCCESS)
        {
                printf("isbuild error %d for %s\n",
                        iserrno, "preformance file");
                exit(1);
        }
        isclose(fdperform);
}
```

```
getfirst()
{
        int cc;

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

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

        addpart(&key, 4, LONGTYPE);
}

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

        addpart(&key, 4, 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 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*(ISAM) call by initializing 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 iname key");
                    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(fdemploy, &key);
            if (cc != SUCCESS)
            {
                    printf("isaddindex error %d for %s \n",
                            iserrno, "perform sal key");
                    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          = 12;
        nparts          = 0;

        addpart(&key, sizeof(double), 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;
}
```

6.3 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 WHOLEKEY 0
#define SUCCESS 0
#define TRUE 1
#define FALSE 0

char emprec[85];
char perfrec[51];
char line[82];
long empnum;

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);
        ststring(line, perfrec+4, 6);

        ststring("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);
        ststring(line, perfrec+19, 30);

        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);
        }
}


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

```
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);
        ststring(line, emprec+4, 20);

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

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

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

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

```
/* move NUM sequential characters from SRC to DEST */
ststring(src, dest, num)
char *src;
char *dest;
int num;
{
        int i;

        /* don't move carriage returns or nulls    */
        for (i = 1; i <= num && *src != '\n' && src != 0; i++)
                *dest++ = *src++;

        /* pad remaining characters in blanks */
        while (i++ <= num)
                *dest++ = ' ';
}
```

6.4 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*(ISAM) call using ISNEXT returns the value [EENDFILE], which indicates that the end of file has been reached.

```
#include <isam.h>

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

char emprec[85];

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\",
                        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 = 0;
        key.k_nparts = 0;
        cstart = 0;
        nparts = 0;

        addpart(&key, 4, 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;
}
```

6.5 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*(ISAM) 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*(ISAM). 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*(ISAM) call will not have been changed (that is, no record will have been read).

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

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

char emprec[83];
char line[80];
long empnum;

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

/*
  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);
        }
        mkemplkey();
        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 EENDFILE:
                        eof = TRUE;
                        break;
                default:
                        printf("isread ISEQUAL error %d \n",
                                iserrno);
                        eof = TRUE;
                        return(1);
                }
        }
        return(0);
}

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

        addpart(&key, 4, 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;
}
```

6.6 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*.

Emp. No.	Review Date	Job New Rating	New Salary	Title
1	790501	g	20,000	PA
1	800106	g	23,000	PA
1	800505	f	24,725	PA
2	760301	g	18,000	JP
2	760904	g	20,700	PA
2	770305	g	23,805	PA
2	770902	g	27,376	SPA
3	800420	f	18,000	JP
4	800420	f	18,000	JP

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

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

char perfrec[51];
char operfrec[51];
char line[81];
long empnum;
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);
        ststring(line, perfrec+4, 6);

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

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

        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(operfrec));
                        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 *= 1.15;
                                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, 4, LONGTYPE);
        addpart(&key, 6, CHARTYPE);
}
```

```
/* move NUM sequential characters from SRC to DEST */
ststring(src, dest, num)
char *src;
char *dest;
int num;
{
        int i;

        /* don't move carriage returns or nulls */
        for (i = 1; i <= num && *src != '\n' && src != 0; i++)
                *dest++ = *src++;

        /* pad remaining characters in blanks */
        while (i++ <= num)
                *dest++ = ' ';
}

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;
}
```

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 integer *iserrno* and the global characters *isstat1* and *isstat2* are set to meaningful values 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.

ISAM codes indicate the following:

NAME	No.	TEXT	STATUS BYTE 1	STATUS BYTE 2
[EDUPL]	100	An attempt was made to add a duplicate value to an index via *iswrite*, *isrewrite*, *isrewcurr* or *isaddindex*.	2	2
[ENOTOPEN]	101	An attempt was made to perform some operation on a file that was not previously opened using the *isopen* call.	9	0
[EBADARG]	102	One of the arguments of the call is not within the range of acceptable values for that argument.	9	0
[EBADKEY]	103	One or more of the elements that make up the key description is outside of the range of acceptable values for that element.	9	0
[ETOOMANY]	104	The maximum number of files that may be open at one time would be exceeded if this request were processed.	9	0
[EBADFILE]	105	The format of the file has been corrupted.	9	0
[ENOTEXCL]	106	In order to add or delete an index, the file must have been opened with exclusive access.	9	0

NAME	No.	TEXT	STATUS BYTE 1	STATUS BYTE 2
[ELOCKED]	107	The record requested by this call cannot be accessed because it has been locked by another user.	9	0
[EKEXISTS]	108	An attempt was made to add an index that has been defined previously.	9	0
[EPRIMKEY]	109	An attempt was made to delete the primary key value. The primary key may not be deleted by the *isdelindex* call.	9	0
[EENDFILE]	110	The beginning or end of file was reached.	1	0
[ENOREC]	111	No record could be found that contained the requested value in the specified position.	2	3
[ENOCURR]	112	This call must operate on the current record. One has not been defined.	2	1
[EFLOCKED]	113	The file is exclusively locked by another user.	9	0
[EFNAME]	114	The file name is too long.	9	0

Two bytes are used to hold status information after calls. They are related in the following way. The first byte holds status information of a general nature, such as success or failure of a call. The second byte contains more specific information that has meaning based on the status code in byte one. The values of the status bytes are:

Byte One

0	Successful Completion
1	End of File
2	Invalid Key
3	System Error
9	User Defined Errors

Byte Two

When status key one is:	Status key two indicates:	
0 — 9	0	No further information is available
0	2	Duplicate key indicator — After a *isread*(ISAM) this indicates that the key value for the current key is equal to the value of that same key in the next record. — After a *isread*(ISAM) or *isrewrite*(ISAM) 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	The primary key value has been changed by the program between the successful execution of a *isread*(ISAM) statement and the execution of the next *isrewrite*(ISAM) statement.
	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.
	4	An attempt has been made to write beyond the externally defined boundaries of an indexed file.
9		The value of status key two is defined by the user.

Chapter 8

The isam.h Header File

This chapter defines the contents of the header file <isam.h>. The file contains definitions that are used for the mode arguments and also definitions of structures that are used in the calls.

Definitions that specify limits in the above table 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 file:

```
#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 */

#define ldint(p)  ((short)(((p)[0]<<BYTESHFT)+((p)[1]&BYTEMASK)))
#define stint(i,p) ((p)[0]=(i)>>BYTESHFT,(p)[1]=(i))
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        */
```

The isam.h Header File

```
/* isread lock modes */
#define ISLOCK    0x100    /* lock record before reading */


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


#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          */

/* the following is for internal use only */
                short k_len;         /* length of whole key  */
                long k_rootnode;     /* pointer to rootnode  */
};

#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 */
                short di_recsize;     /* data record size       */
                short di_idxsize;     /* index record size      */
                long di_nrecords;     /* number of records      */
};                                    /* in file */


#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    /* file name too long    */
#define EBADMEM      116    /* can't alloc memory    */
#define EBADCOLL     117    /* bad custom collating  */
```

The isam.h Header File

```
/*
 * For system call errors
 *   iserrno = errno (system error code 1-99)
 *   iserrio = IO_call + IO_file
 *     IO_call  = what system call
 *     IO_file  = which file caused error
 */

#define IO_OPEN         0x10    /* open()    */
#define IO_CREA         0x20    /* creat()   */
#define IO_SEEK         0x30    /* lseek()   */
#define IO_READ         0x40    /* read()    */
#define IO_WRIT         0x50    /* write()   */
#define IO_LOCK         0x60    /* locking() */
#define IO_IOCTL        0x70    /* ioctl()   */

extern int iserrno;        /* isam error return code      */
extern int iserrio;        /* system call error code      */
extern long isrecnum;      /* record number of last call  */
extern char isstat1;       /* cobol status characters     */
extern char isstat2;

/*  error message usage:
 *
 *  if (iserrno >= 100 && iserrno < is_nerr)
 *      printf("ISAM error %d: %s \n",
 *        iserrno, is_errlist[iserrno-100]);
 */
```

Chapter 9

Call Specifications

This chapter contains detailed descriptions of the X/OPEN ISAM functions. The following general notes apply throughout.

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*(ISAM) or *isopen*(ISAM), the return value will be a legal file descriptor or a -1. A -1 indicates that an error has occurred, and *iserrno* has been set. Also, the global characters *isstat1* and *isstat2* are set for the convenience of integration with COBOL. See Chapter 7, Exception Handling, for more information.

9.2 <isam.h> HEADER FILE

Some parameters in this chapter are declared to be structure types that are defined in the <isam.h> header file. Also defined are symbolic values.

9.3 Key Structure

The structures *keydesc* and *keypart*, also 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 ORed 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.

k_nparts is an integer that 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. *kp_start* indicates the starting byte of the key part that is being defined. *kp_leng* 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 file, <**isam.h**>, see Chapter 8, The isam.h Header File. If this part of the key is in descending order, the type constant should be ORed to the ISDESC constant (defined in <**isam.h**>). For more information about creating and manipulating indexes, see Chapter 4, Indexing.

9.4 RECORD NUMBER OF LAST CALL

Isrecnum is a 4-byte field that is set following the sucessful completion of all record-based calls. It identifies, in an implementation-dependent, shorthand way, the record just referenced. This returned value may be used in input to the *isdelrec*(ISAM), *isread*(ISAM), and *isrewrec*(ISAM) calls to perform optimised deletes, reads, and updates. 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(ISAM)	isdelete(ISAM)	isdelrec(ISAM)	isread(ISAM)
isrewcurr(ISAM)	isrewrec(ISAM)	isrewrite(ISAM)	isstart(ISAM)
iswrcurr(ISAM)	iswrite(ISAM)		

9.5 CURRENT RECORD POSITION

The current record position should not be confused with *isrecnum* (see above). The current record position 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. If the current record is deleted (by using *isdelcurr*(ISAM)), the current record position will not change and will continue to indicate the now deleted record. The current record may be rewritten directly using *isrewcurr*(ISAM).

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

isopen(ISAM) isread(ISAM) isstart(ISAM) iswrcur(ISAM)

and used in input to:

isdelcurr(ISAM) isread(ISAM) isrewcurr(ISAM)

NAME

isaddindex — add index to an ISAM file

SYNTAX

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

DESCRIPTION

Isaddindex is used to add an 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.

There is no limit to the number of indexes that may be added through the *isaddindex* call. However, 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 File'').

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

NAME

isbuild — create an ISAM file

SYNTAX

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

DESCRIPTION

Isbuild 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.

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

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 Program Examples'' for more details on key definition and use.

The *mode* parameter is 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*(ISAM) and *isunlock*(ISAM) calls or the ISLOCK mode flag of the *isread*(ISAM) call and the *isrelease*(ISAM) 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 file <**isam.h**>, for which a complete listing can be found in Chapter 8, ''The isam.h Header File''.

Modes that are used in the *isbuild* call are:

one of these,	added arithmetically to one of these:
ISEXCLLOCK	ISINPUT
ISMANULOCK	ISOUTPUT
ISAUTOLOCK	ISINOUT

NAME

isclose — close an ISAM file

SYNTAX

```
isclose(isfd)
int isfd;
```

DESCRIPTION

Isclose is used to close an ISAM file. Any locks that are held for the file by the process issuing the *isclose* call are released.

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

NAME

isdelcurr — delete current record

SYNTAX

```
isdelcurr (isfd)
int isfd;
```

DESCRIPTION

Isdelcurr differs from *isdelete*(ISAM) 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. This call is useful when the primary key is not unique and the record cannot be located and deleted in one call. *Isrecnum* is set to indicate the current record, (the record just deleted), whose position is left unchanged.

ISDELETE(ISAM)

NAME

isdelete — delete record specified by primary key

SYNTAX

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

DESCRIPTION

isdelete deletes the record specified by a unique primary key from the file indicated by *isfd*. The appropriate values will also be deleted from each index. If the primary index allows duplicates, then *isread*(ISAM) and *isdelcurr*(ISAM) should be used instead. *Isrecnum* is set to indicate the record just deleted, while the current record position is left unchanged.

NAME

isdelindex — remove index from an ISAM file

SYNTAX

```
isdelindex(isfd, keydesc)
int isfd;
struct keydesc *keydesc;
```

DESCRIPTION

Isdelindex 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.

NAME

isdelrec — delete record specified by record number (OPTIONAL)

SYNTAX

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

DESCRIPTION

Isdelrec differs from *isdelete*(ISAM) 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. *Recnum* must be a previously obtained *isrecnum* value. This call will set *isrecnum* to the value of *recnum*, while the current record position is left unchanged.

NAME

iserase — remove an ISAM file

SYNTAX

```
iserase (filename)
char *filename;
```

DESCRIPTION

iserase will remove the file specified by *filename*.

NAME

isindexinfo — access file information

SYNTAX

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

DESCRIPTION

Isindexinfo 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. General information such as 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 *buffer* parameter can contain information in the format of either *key-desc* 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 File'').

NAME

islock — lock an ISAM file

SYNTAX

```
islock(isfd)
int isfd;
```

DESCRIPTION

Islock will lock the entire file that is specified by *isfd*. More discussion of locking can be found in Chapter 5, ''Locking''.

ISOPEN(ISAM)

NAME

isopen — open an ISAM file

SYNTAX

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

DESCRIPTION

Isread 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 to the first record in order of the primary index. If another ordering is desired, the *isstart*(ISAM) call can be used to select another index.

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.

The mode is specified by using the define macros that are found in the header file <**isam.h**>, for which a complete listing can be found in Chapter 8, ''The isam.h Header File''. Modes that are used in the *isopen* command are:

One of these,	arithmetically added to one of these:
ISEXCLLOCK	ISINPUT
ISMANULOCK	ISOUTPUT
ISAUTOLOCK	ISINOUT

NAME

isread — read records

SYNTAX

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

DESCRIPTION

Isread 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.

Isread 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 file <**isam.h**>. Refer to Chapter 8, "The isam.h Header File" for the contents of this file.

Isread can also read records specified by a previously set *isrecnum*. First, call *isstart*(ISAM) 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 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*(ISAM) call. Entire files may be locked and unlocked by using the *islock*(ISAM) and *isunlock*(ISAM) calls.

Modes that are used in the *isread* call are:

One of these,	optionally arithmetically added to
ISCURR	ISLOCK
ISNEXT	
ISFIRST	
ISGREAT	
ISEQUAL	
ISPREV	
ISLAST	
ISGTEQ	

ISRELEASE(ISAM)

NAME

isrelease — unlock records

SYNTAX

isrelease (isfd)
int isfd;

DESCRIPTION

Isrelease unlocks records that have been locked using the ISMANULOCK mode in the *isread*(ISAM) call. 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''.

ISRENAME(ISAM)

NAME

isrename — rename an ISAM file

SYNTAX

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

DESCRIPTION

Isrename will rename the file specified by the *oldname* parameter to the name specified by the *newname* parameter.

NAME

isrewcurr — rewrite current record

SYNTAX

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

DESCRIPTION

Isrewcurr is used to rewrite the current record of the file indicated by *isfd* with the contents of the character array *record*. The appropriate values will be rewritten to each index that is defined. The primary key value may be changed. *isrewcurr* is useful when the primary key is not unique and the record cannot be located and rewritten in one call. *Isrecnum* is set to indicate the current record, whose position is left unchanged.

NAME

isrewrec — rewrite record specified by record number (OPTIONAL)

SYNTAX

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

DESCRIPTION

Isrewrec is used to rewrite the record specified by *recnum* in the file indicated by *isfd* with the contents of the character array *record*. *recnum* must be a previously obtained *isrecnum* value. Each index will be appropriately updated. This call will set *isrecnum* to the value of *recnum*, while the current record position will remain unchanged.

NAME

isrewrite — rewrite record specified by primary key

SYNTAX

```
isrewrite(isfd, record)
int isfd;
char *record;
```

DESCRIPTION

Isrewrite is used to change one or more values for a record that is already in the file identified by *isfd*. The primary key is used to determine which record should be changed, and the *record* parameter contains the changes. The primary key value must be unique and may not be changed. The whole record is written to the data file. Only the changed index values will be rewritten to each index that is defined.

This is consistent with COBOL requirements for maintaining the order of records in duplicate chains. *Isrewrite* does not change the position of the current record pointer, while *isrecnum* is set to indicate this record.

NAME

isstart — select an index

SYNTAX

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

DESCRIPTION

Isstart 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*(ISAM) 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.

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*(ISAM) 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*(ISAM) 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 next higher record whose key value is greater than that found in the *record* buffer parameter, or *isstart* will return an error condition (-1) and set *iserrno* to ENOREC.

The ISGTEQ mode parameter finds the record that has the next higher key value 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.

The above define macros, ISFIRST, ISLAST, ISEQUAL, ISGREAT, and ISGTEQ, can be found in the header file <**isam.h**>.

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

Isstart 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. *Isstart* need not be used to find each record before it is read using *isread*(ISAM).

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

ISUNLOCK(ISAM)

NAME

isunlock — unlock an ISAM file

SYNTAX

```
isunlock (isfd)
int isfd;
```

DESCRIPTION

Isunlock 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''.

NAME

iswrcurr — write record and set current position (OPTIONAL)

SYNTAX

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

DESCRIPTION

Iswrcurr 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.

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

NAME

iswrite — write record

SYNTAX

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

DESCRIPTION

Iswrite 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.

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