

C-ISAM

Indexed Sequential Access Method

Programmer's Manual

Version 5.0

December 1991

Part No. 000-7115

THE INFORMIX SOFTWARE AND USER MANUAL ARE PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE INFORMIX SOFTWARE AND USER MANUAL IS WITH YOU. SHOULD THE INFORMIX SOFTWARE AND USER MANUAL PROVE DEFECTIVE, YOU (AND NOT INFORMIX OR ANY AUTHORIZED REPRESENTATIVE OF INFORMIX) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR, OR CORRECTION. IN NO EVENT WILL INFORMIX BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS, OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE SUCH INFORMIX SOFTWARE OR USER MANUAL, EVEN IF INFORMIX OR AN AUTHORIZED REPRESENTATIVE OF INFORMIX HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY. IN ADDITION, INFORMIX SHALL NOT BE LIABLE FOR ANY CLAIM ARISING OUT OF THE USE OF OR INABILITY TO USE SUCH INFORMIX SOFTWARE OR USER MANUAL BASED UPON STRICT LIABILITY OR INFORMIX’S NEGLIGENCE. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS, WHICH VARY FROM STATE TO STATE.

All rights reserved. No part of this work covered by the copyright hereon may be reproduced or used in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems—without permission of the publisher.

Published by: Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025

INFORMIX and **C-ISAM** are registered trademarks of Informix Software, Inc.

UNIX is a registered trademark of UNIX System Laboratories, Inc.
PostScript is a registered trademark of Adobe Systems Incorporated.
X/Open is a trademark of X/Open Company Ltd.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subdivision (b)(3)(ii) of the Rights in Technical Data and Computer Software Clause at 52.227-7013 (and any other applicable license provisions set forth in the Government contract).

Copyright © 1981-1991 by Informix Software, Inc.

Table of Contents

Introduction

C-ISAM and Other Informix Products	Intro-4
Other Useful Documentation	Intro-4
How to Use This Manual	Intro-4
Typographical Conventions	Intro-5
Useful On-Line Files	Intro-5
Compliance with Industry Standards	Intro-6
Changes in Locking Mechanisms	Intro-6
Migrating C-ISAM Files	Intro-6

Chapter 1

How to Use C-ISAM

Chapter Overview	1-3
What Is a C-ISAM File?	1-3
Data Records in C-ISAM Files	1-4
Programming with Variable-Length Records	1-5
Representation of Data	1-6
Comparison of C-ISAM to C Library Functions	1-8
Indexed Sequential Access Method	1-9
Keys in C-ISAM Files	1-10
Using Keys	1-10
Organization of C-ISAM Files	1-13
Building a C-ISAM File	1-13
C-ISAM Error Handling	1-17
Manipulating Records in C-ISAM Files	1-18
Identifying Records	1-18
Adding Records	1-19
Deleting Records	1-21
Updating Records	1-22
Finding Records	1-24
Opening and Closing Files	1-28
Opening a File in Exclusive Mode	1-29

	Opening a Variable-Length File	1-30
	Maximum Number of Open Files	1-30
	Closing Fixed and Variable-Length Files	1-30
	Compiling Your C-ISAM Program	1-31
	C-ISAM Data File Structure	1-31
	Summary	1-32
Chapter 2	Indexing	
	Overview	2-3
	Defining an Index	2-3
	Key Structures	2-6
	Manipulating Indexes	2-7
	Adding Indexes	2-8
	Deleting Indexes	2-9
	Defining Record Number Sequence	2-10
	Determining Index Structures	2-11
	B+ Tree Organization	2-12
	Searching for a Record	2-15
	Adding Keys	2-16
	Removing Keys	2-21
	Index File Structure	2-22
	Performance Considerations	2-23
	Key Size and Tree Height	2-23
	Key Compression	2-24
	Multiple Indexes	2-27
	Summary	2-28
Chapter 3	Data Types	
	Overview	3-3
	Defining Data Types for Keys	3-3
	C-ISAM Machine-Independent Data Types	3-4
	Defining Data Records	3-5
	Data Types in Variable-Length Records	3-7
	C-ISAM Data Type Conversion Routines	3-8
	DECIMALTYPE Data Type	3-11
	Using DECIMALTYPE Data Type Numbers	3-11
	Summary	3-16
Chapter 4	Locking	
	Overview	4-3
	Concurrency Control	4-3
	Types of Locking	4-6
	File-Level Locking	4-6

	Record-Level Locking	4-8
	Increasing Concurrency	4-11
	Error Handling	4-11
	Summary	4-12
Chapter 5	Transaction Management Support Routines	
	Overview	5-3
	Why Use Transaction Management?	5-3
	Transaction Management Services	5-4
	Implementing Transactions	5-4
	Transactions with Variable-Length Records	5-6
	Logging and Recovery	5-7
	Data Integrity	5-8
	Concurrent Execution of Transactions	5-8
	Summary	5-11
Chapter 6	Additional Facilities	
	Overview	6-3
	File Maintenance Functions	6-3
	Forcing Output	6-4
	Unique Identifiers	6-5
	Audit Trail Facility	6-6
	Using the Audit Trail	6-6
	Audit Trail File Format	6-8
	Clustering a File	6-9
	File Maintenance with Variable-Length Records	6-9
	If Data Files Are Corrupted	6-10
	If Index Files Are Corrupted	6-10
	Summary	6-13
Chapter 7	Sample Programs Using C-ISAM Files	
	Overview	7-3
	Record Definitions	7-3
	Error Handling in C-ISAM Programs	7-4
	Building a C-ISAM File	7-5
	Adding Additional Indexes	7-6
	Adding Data	7-7
	Random Update	7-10
	Sequential Access	7-14
	Chaining	7-17
	Using Transactions	7-22
	Summary	7-25

Chapter 8

Call Formats and Descriptions

Overview 8-3

Functions for C-ISAM File Manipulation 8-6

ISADDINDEX 8-8

ISAUDIT 8-10

ISBEGIN 8-13

ISBUILD 8-15

ISCLEANUP 8-18

ISCLOSE 8-19

ISCLUSTER 8-20

ISCOMMIT 8-22

ISDELCURR 8-24

ISDELETE 8-25

ISDELINDEX 8-27

ISDELREC 8-29

ISERASE 8-31

ISFLUSH 8-32

ISINDEXINFO 8-33

ISLOCK 8-36

ISLOGCLOSE 8-38

ISLOGOPEN 8-39

ISOPEN 8-40

ISREAD 8-42

ISRECOVER 8-46

ISRELEASE 8-47

ISRENAME 8-48

ISREWCURR 8-50

ISREWREC 8-52

ISREWRITE 8-54

ISROLLBACK 8-56

ISSETUNIQUE 8-58

ISSTART 8-60

ISUNIQUEID 8-63

ISUNLOCK 8-64

ISWRCURR 8-65

ISWRITE 8-67

Format-Conversion and Manipulation Functions 8-69

Format-Conversion Functions 8-69

LDCHAR 8-70

LDDBL 8-71

LDDBLNULL 8-72

LDDECIMAL 8-73

LDFLOAT 8-75



LDFLTNULL	8-76
LDINT	8-77
LDLONG	8-78
STCHAR	8-79
STDBL	8-80
STDBLNULL	8-81
STDECIMAL	8-82
STFLOAT	8-84
STFLTNULL	8-85
STINT	8-86
STLONG	8-87
DECIMALTYPE Functions	8-88
DECCVASC	8-89
DECTOASC	8-91
DECCVINT	8-93
DECTOINT	8-94
DECCVLONG	8-95
DECTOLONG	8-97
DECCVFLT	8-98
DECTOFLT	8-99
DECCVDBL	8-100
DECTODBL	8-101
DECADD, DECSUB, DECMUL, and DECDIV	8-102
DECCMP	8-104
DECCOPY	8-105
DECECVT and DECFCVT	8-106
Summary	8-108

Appendix A	The <i>bcheck</i> Utility
Appendix B	Header Files
Appendix C	Error Codes
Appendix D	File Formats
Appendix E	System Administration
	Index

Preface

The *C-ISAM Programmer's Manual* describes the **C-ISAM** functions and facilities. The book assumes that you are familiar with the C programming language and have used the standard C library functions related to files and input/output operations.

Summary of Chapters

The *C-ISAM Programmer's Manual* includes the following chapters:

Chapters 1, 2, and 3 explain major features that are part of every program using **C-ISAM** functions.

- Chapter 1 explains how to create and manipulate **C-ISAM** files.
- Chapter 2 explains the organization and use of indexes.
- Chapter 3 describes the data types that may be used in **C-ISAM** files and how they are handled in **C-ISAM** programs.

Chapters 4, 5, and 6 explain specialized facilities.

- Chapter 4 describes file and record locking and how these are implemented.
- Chapter 5 explains how to ensure data integrity using transaction management.
- Chapter 6 describes additional **C-ISAM** functions and explains the use of audit trails.

The rest of the manual contains sample programs and reference material.

- Chapter 7 contains several complete programs that use the **C-ISAM** functions described in earlier chapters.

-
- Chapter 8 serves as the reference section for each **C-ISAM** function. It is organized so that the syntax and details of each function are easy to locate and use.
 - Appendix A describes the utility program for checking the integrity of **C-ISAM** files.
 - Appendix B contains the source code for the header files you need to include in **C-ISAM** programs.
 - Appendix C lists the errors that can occur during execution of **C-ISAM** calls.
 - Appendix D shows the physical file layouts for files that **C-ISAM** uses.
 - Appendix E explains how to set up your operating system to use **C-ISAM**.

Related Reading

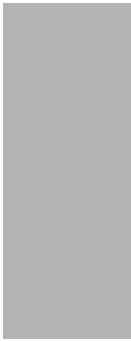
If you are not familiar with the C language, refer to the C programmer's manual that comes with your system, or any other book on the C language. One such reference book is *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, 1978.

If you have had no prior experience with database management, you may want to refer to an introductory text like C. J. Date's *Database: A Primer* (Addison-Wesley Publishing, 1983). If you want more technical information on database management, consider consulting the following texts, also by C. J. Date:

- *An Introduction to Database Systems, Volume I* (Addison-Wesley Publishing, 1990)
- *An Introduction to Database Systems, Volume II* (Addison-Wesley Publishing, 1983)

Introduction

C-ISAM and Other Informix Products	4
Other Useful Documentation	4
How to Use This Manual	4
Typographical Conventions	5
Useful On-Line Files	5
Compliance with Industry Standards	6
Changes in Locking Mechanisms	6
Migrating C-ISAM Files	6



C-ISAM is an Indexed Sequential Access Method defined and implemented for the C language by Informix. An *access method* is a way to retrieve pieces of information, *records*, from a larger set of information, a *file*.

An indexed sequential access method allows you to find records in a specific order, such as finding all employees in order by employee number or in order by name (sequential access). It also allows you to find specific pieces of information quickly, such as information about employee 100 or employee R. Smith, without having to look at extra records (indexed access).

The ability to find specific records quickly is important if you are interested only in one or a few pieces of information out of a much larger set. If you are only interested in the record belonging to R. Smith, you should be able to access it directly without accessing other records.

When you want to produce a list of all employees, ordered in some way, such as by name or by number, you need the ability to access the records in sequential order.

C-ISAM is a library of C language functions that create and manipulate indexed files. An index allows you to do the following tasks without additional programming:

- Find a specific record within a large file very quickly
- Define an order for sequential processing of the file

Each index is defined by a key. The *key* is the field or fields (including parts of fields) that you use to locate records. Keys also define the order in which you want to process the file. Employee number or employee name are examples of fields that can be indexed to allow you to find specific employees by name or number, or to process the file in number or name sequence.

C-ISAM allows great flexibility for defining and using indexes. You can have as many indexes as you need, without restrictions. You can create or remove indexes at any time without affecting data records or other indexes.

You are not required to use an index to locate a record. You can access records by relative location within the file; for example, the 100th record from the beginning of the file.

C-ISAM includes several other features, such as locking and support for transactions, to provide data integrity. The use of these facilities allows you to ensure that information is accessible, accurate in its consistency, and correctly processed.

The locking facility allows you to write programs so that two or more programs cannot interfere with each other and cause inconsistencies in the data.

C-ISAM provides support routines for transaction management to extend your ability to write programs that maintain the consistency and accuracy of **C-ISAM** files. These routines also allow you to recover data that is lost due to hardware failures.

C-ISAM and Other Informix Products

Applications that you develop with **C-ISAM** do not need to interact with any other database-management software. However, **C-ISAM** is not the only Informix product available. Informix Software produces a variety of application development tools, CASE tools, database servers, and client/server products.

Other Useful Documentation

You may want to refer to a number of related Informix product documents that complement the *C-ISAM Programmer's Manual*.

- You, or whoever installs **C-ISAM**, should refer to the *UNIX Products Installation Guide* for your particular release to ensure that **C-ISAM** is properly set up before you begin to work with it.
- When errors occur, you can look them up, by number, and find their cause and solution in the *Informix Error Messages* manual. The error messages are also documented in Appendix C of this manual.

How to Use This Manual

This manual assumes that you are using 5.0 **C-ISAM** on a UNIX operating system.

Typographical Conventions

The *C-ISAM Programmer's Manual* uses a standard set of conventions to introduce new terms, illustrate screen displays, describe command syntax, and so forth. The following typographical conventions are used throughout the manual:

<i>italics</i>	When new terms are introduced, they are printed in italics.
boldface	Database names, table names, column names, filenames, utilities, and other similar terms are printed in boldface.
computer	Information that you enter are printed in a computer typeface.
KEYWORD	All keywords appear in uppercase letters.

Useful On-Line Files

In addition to the Informix set of manuals, the following on-line files, located in the directory you have indicated at install-time to hold the sample programs, may supplement the information in the *C-ISAM Programmer's Manual*:

Documentation Notes	describe features not covered in the manual or which have been modified since publication. The file containing the Documentation Notes for this product is called ISAMDOC_5.0 .
Release Notes	describe feature differences from earlier versions of Informix products and how these differences may affect current products. The file containing the Release Notes for C-ISAM and other products is called ENGREL_5.0 .
Machine Notes	describe any special actions required to configure and use Informix products on your machine. The file containing the Machine Notes for this product is called ISAM_5.0 .

Please examine these files because they contain vital information about application and performance issues.

When you install **C-ISAM**, several sample programs (**ex1.c** through **ex7.c**) are also installed. These examples are documented in this manual, so you may find it helpful to compile and run these programs at some point.

Compliance with Industry Standards

C-ISAM conforms to the guidelines put forth in the *X/Open Portability Guide, Issue 3 (XPG3)*.

Changes in Locking Mechanisms

Version 5.0 of C-ISAM and INFORMIX-SE may use a different locking mechanism from past versions on your hardware platform. For example, this is the case if you are using Version 5.0 of these products on a Sun platform and your previous version was 4.10.UC1 or earlier, where the SYS5LOCK locking mechanism (**fcntl/flock**) replaces CREATLOCK(**.lok** files). It may also be the case if your previous version of INFORMIX-SE or C-ISAM was a shared-memory implementation. If the locking mechanism has changed on your platform, this fact is noted in the on-line **SE_5.0** or **ISAM_5.0** file.

In general, C-ISAM files created with different locking mechanisms are not compatible. If you have C-ISAM files or INFORMIX-SE databases created with an earlier version of the product on an affected platform, and you wish to use them with Version 5.0, follow the migration procedures outlined in this section after your 5.0 software is installed. Once the migration is complete, you cannot revert to the previous locking mechanism.

Before you migrate C-ISAM files or INFORMIX-SE databases, you should back them up. You must also set the RESETLOCK environment variable. You do not have to set RESETLOCK to a specific value; simply specify the variable in the exported environment as follows:

```
Bourne or Korn shell:  RESETLOCK=
                        export RESETLOCK

C shell:                setenv RESETLOCK
```

Migrating C-ISAM Files

Two methods are available for migrating C-ISAM files. The first method is to set the RESETLOCK environment variable and then run your existing application, making sure that all files are opened at least once. The second method is to run the **bcheck** utility on any files that you want to convert for use under Version 5.0. The **bcheck** opens the referenced C-ISAM file or files and automatically updates them to the new locking method.

Once the migration is complete, you do not need to continue to set the RESET-LOCK variable to work with the new files. You can safely remove any remaining **.lok** files.

How to Use C-ISAM

Chapter Overview	3
What Is a C-ISAM File?	3
Data Records in C-ISAM Files	4
Programming with Variable-Length Records	5
Representation of Data	6
Comparison of C-ISAM to C Library Functions	8
Indexed Sequential Access Method	9
Indexed Access	9
Sequential Access	9
Flexibility	9
Keys in C-ISAM Files	10
Using Keys	10
Choosing a Key	11
Key Descriptions	11
Unique and Duplicate Keys	12
Primary Keys	12
Organization of C-ISAM Files	13
Building a C-ISAM File	13
Building a Variable-Length File	17
C-ISAM Error Handling	17
Manipulating Records in C-ISAM Files	18
Identifying Records	18
Using the Key Value	18
Using the Current Record	18
Using the Record Number	19
Summary of Record Identification Methods	19
Adding Records	19
Deleting Records	21
Updating Records	22

Finding Records	24
Using the isstart Function	26
Finding Records by Record Number	28
Opening and Closing Files	28
Opening a File in Exclusive Mode	29
Opening a Variable-Length File	30
Maximum Number of Open Files	30
Closing Fixed and Variable-Length Files	30
Compiling Your C-ISAM Program	31
C-ISAM Data File Structure	31
Summary	32

Chapter Overview

C-ISAM is a set of functions that can be used in C language programs. This chapter gives an overview of the basic concepts that you need to begin using **C-ISAM**. It also explains how to use the most common functions to perform the following tasks:

- Create a **C-ISAM** file
- Add records to the file
- Remove records from the file
- Update existing records
- Find and retrieve records
- Open and close the file
- Determine the length and number of file records

This chapter also shows you how to compile your program and introduces details about the structure and organization of **C-ISAM** data files.

What Is a C-ISAM File?

A **C-ISAM file** is a collection of data that you would like to keep on the computer. For example, you may want to keep information about all employees on the computer. To do this, you must first decide what data to keep for each employee. Each item that you decide to keep is called a *field*.

You may decide to keep an employee number, the first and last names, address, and city for each employee. This collection of fields is called a *record*. You must determine the data type and the length of each field.

This manual uses an **employee** file with employee records as the primary example to show you how to use **C-ISAM**. Figure 1-1 and Figure 1-2 show the Employee record for this example.

Description	Type	Length	Pointer	Offset from Beginning of Record
Employee Number	Long	4	p_empno	0
Last Name	Char	20	p_lname	4
First Name	Char	20	p_fname	24
Address	Char	20	p_eaddr	44
City	Char	20	p_ecity	64
Total Length in Bytes				84

Figure 1-1

Employee Record

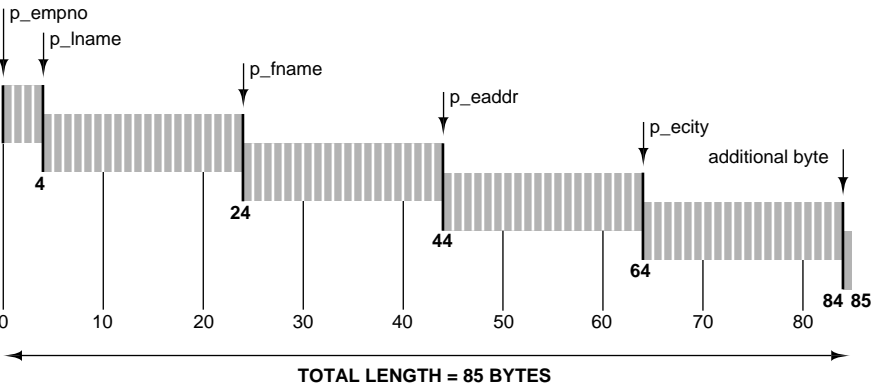


Figure 1-2

Employee Record Illustration

The record is the collection of fields. Each field has a data type and a length. The *offset* is the relationship of the field to the beginning of the record. The Employee Number field starts at the beginning of the record, at offset 0, and the Last Name field starts after the Employee Number, at offset 4.

Data Records in C-ISAM Files

Records in a C-ISAM file can be of either fixed or variable length. You must reserve space for at least one record in your program. The record must hold the contents of the fields and one more byte. The easiest way to do this is to

declare a character variable of the size that your record layout indicates plus one byte (See Figure 1-1). The following declarations are sufficient for the employee record:

```
char emprec[84+1];
```

or

```
char emprec[85];
```

You can define the location of each field by its offset from the beginning of the record and declare a pointer variable for each field. The pointers become the arguments to functions that operate on fields. To set up the Employee Number and Name fields, you declare the following pointer variables:

```
char *p_empno = emprec+ 0;  
char *p_lname = emprec+ 4;  
char *p_fname = emprec+24;
```

These declarations use pointer arithmetic to define the field position. The offset within the record is added to the address of the record in memory. The following declarations are equivalent:

```
char *p_empno = &emprec[0];  
char *p_lname = &emprec[4];  
char *p_fname = &emprec[24];
```

Use the record address, **emprec**, to refer to the record.

Programming with Variable-Length Records

A file can contain either variable-length or fixed-length records. Variable-length records can have a fixed-length portion. The variable-length portion of a record is at the end of the record, after the fixed-length portion. For compatibility with earlier versions of C-ISAM, a record that is not specifically labeled fixed length or variable length defaults to fixed length. As with fixed-length records, you must declare a C variable that holds the data in the variable-length record while you manipulate it.

The fixed-length portion of a variable-length record is stored in the data file, along with a four-byte pointer to the variable-length portion of the record. The variable-length portion of the record is stored in the index file. For this reason, *it is important that you do not remove the index files (.idx)*. If you remove the .idx files, there is no way of restoring the files and the variable-length data contained within them, other than restoring them from a backup.

If the index portion of the .idx files becomes corrupted, run the **bcheck** utility without removing the .idx files. This leaves the variable-length data intact. Complete information for recovery in the event of a data loss is described in the section “File Maintenance with Variable-Length Records” on page 6-10 of this manual.

The ability to use variable-length records is only available with C-ISAM; it is not available with any Informix products that use INFORMIX-SE.

Representation of Data

C-ISAM uses data types that are equivalent to the C language data types on your machine. C-ISAM representation of these data types, however, is machine independent. Thus, the way C-ISAM stores the data can be different from the internal representation of the data while your program executes.

For example, Employee Number is a **long** integer. The C-ISAM equivalent is LONGTYPE. The size of a C-ISAM LONGTYPE is LONGSIZE. The other items in the record are CHARTYPE, corresponding to the C language **char** data type. (These parameters, as well as other parameters you need in programs that use C-ISAM, are in the header file **isam.h** that you must include in your programs. Appendix B, “Header Files,” contains a listing of **isam.h**.)

C-ISAM provides functions to convert between the internal representation of data on your machine and the way that C-ISAM stores the data. (See Figure 1-3.) For example, the function **stlong** takes a C language **long** integer and stores it into the record. The function **ldlong** retrieves the C-ISAM representation of a **long** integer from the record and places it in a C language **long** variable. You must always convert between the internal representation of data on your machine and the machine-independent C-ISAM representation of the data. Chapter 3, “Data Types,” describes the conversion functions that you can use.

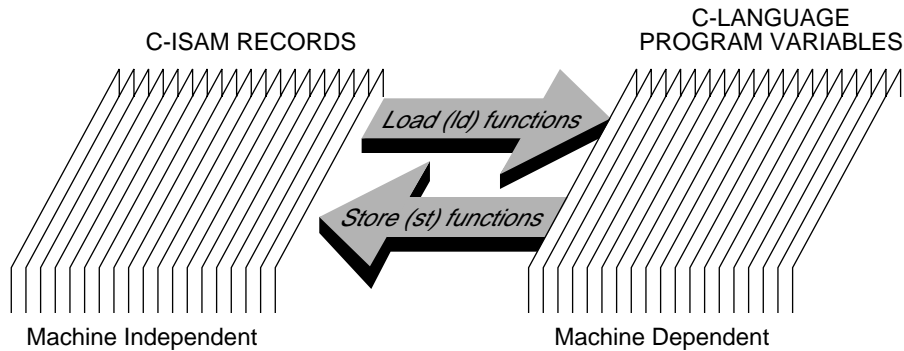


Figure 1-3

Converting the Internal Representation of Data to the C-ISAM Representation of Data

Figure 1-4 shows how you can transfer data between a C-ISAM data file record and the internal program variables for the record in Figure 1-1.

```

char emprec[85]; /* C-ISAM Record */

char *p_empno = emprec + 0; /* Field Definitions */
char *p_lname = emprec + 4;
char *p_fname = emprec + 24;
char *p_eaddr = emprec + 44;
char *p_ecity = emprec + 64;

/* Program Variables */
long empno;
char lname[21];
char fname[21];
char eaddr[21];
char ecity[21];
/* Store program variables in C-ISAM data record */
stlong (empno, p_empno);
stchar (lname, p_lname, 20);
stchar (fname, p_fname, 20);
stchar (eaddr, p_eaddr, 20);
stchar (ecity, p_ecity, 20);
/* Load program variables from C-ISAM data record */
empno = ldlong(p_empno);
ldchar (p_lname, 20, lname);
ldchar (p_fname, 20, fname);
ldchar (p_eaddr, 20, eaddr);
ldchar (p_ecity, 20, ecity);
    
```

Figure 1-4

Transferring Data Between Program Variables and a C-ISAM Data Record

The function **stlong** takes the long integer **empno**, converts it into the C-ISAM machine-independent representation of a **long** integer, and places it in the record, starting at address **p_empno**. The function converts the C-ISAM long integer starting at position **p_empno** in the data record, and returns its value to the program variable **empno**.

The function **stchar** takes program variables, such as **lname**, removes the null character, and places the data in the C-ISAM data record, starting in position **p_lname** as shown in Figure 1-2. It pads the C-ISAM data record with trailing spaces up to the number specified, which is 20.

The function **ldchar** is the reverse of **stchar**. The data at the starting position in the record, **p_lname** for example, is transferred to a program variable **lname**. The transfer stops after 20 characters. Trailing spaces are removed and the program variable is null-terminated.

Comparison of C-ISAM to C Library Functions

You can use the data structure described in Figure 1-2 to write records to a file created by the C standard library function **creat**. You can also use the structure to retrieve those records. The standard library functions, **read** and **write**, allow you to read and write the next arbitrary group of bytes (you specify the number) in relation to the last group read or written. The C function **lseek** allows you to change the starting position for the next read or write.

C-ISAM also allows you to perform these operations. C-ISAM functions, however, operate on the records that you define. You do not have to concern yourself with the byte positions within the file in order to find the information that you wish to access. This, however, is not the main advantage of using C-ISAM files.

C-ISAM offers you the following advantages:

- You can define one or more orders for processing the records. The contents of the records determine the order, not the physical ordering of records in the file.
- You can quickly find specific records within files, even when the files are quite large.

Indexed Sequential Access Method

You can store thousands, or tens of thousands, of data records in a file using the standard library functions. If you wanted to find employee 100, or the employee R. Smith, your program might have to search the entire file.

C-ISAM gives you a much faster way to find a record, which eliminates the need for your program to search a data file sequentially when it looks for just a few records. **C-ISAM** provides an access method that uses an index.

Indexed Access

The indexes of a **C-ISAM** file are similar in function to the index of this book. You use a book index to locate a page that contains the information that you need. The index is composed of words that identify the contents of the page. These entries are called *keywords*. The **C-ISAM** index, however, is not restricted to words. Its entries are simply called *keys*.

In the book, the keyword refers you to a page number. In the **C-ISAM** file, the key points to a record that is identified by its record number. In both cases, you use the *pointer* (page number in a book or record number in a file) to locate the item of interest.

This book has only one index. With **C-ISAM**, however, you can have as many indexes as you need. For example, you can define two indexes: one for the Employee Number field, and another for the Employee Name field. This allows you to find quickly the record for Employee Number 100 or employee R. Smith.

Sequential Access

C-ISAM also allows sequential processing of records in the order defined by the key. You can access all or part of the file in any of the following orders:

- By the Employee Number key
- Alphabetically by the Employee Name key
- By any other order that you define with an index

Flexibility

C-ISAM enhances the functionality of your programs through its flexibility. If you add a section to a book, rearrange paragraphs or sections, remove a few pages, you must re-create your index since the keywords must appear in relation to each other. In this case, the relationship of the keywords to each other

is alphabetic order. A **C-ISAM** index changes automatically whenever a data record changes. If you hire or terminate an employee, or change anything in a record, **C-ISAM** immediately updates all indexes.

You can create an index on any field, on several fields to be used together, or on parts thereof, that you want to use as a key. The keys in indexes allow you quick access to specific records and define orders for sequential processing of a **C-ISAM** file. When you no longer need an index, you can delete it. Addition and deletion of indexes have no effect on the data records or on other indexes.

Keys in C-ISAM Files

In the analogy to the book index in the section, “Indexed Access,” earlier in this chapter, an entry in the index for this book is a keyword. With each keyword there is a pointer to a page number. In the analogy, each key in a **C-ISAM** file points to a data record, or simply, a record.

In the **employee** file, you may want to access records by employee number. This requires an index, just as the book does. The keys are the employee numbers. In other words, the Employee Number index contains the employee number for each employee in the file. (Conceptually, you should think of the index as ordering the records by employee number. Chapter 2, “Indexing,” shows the actual organization of the index.)

The employee numbers in the index point to data records. The format of the data record is shown in Figure 1-1 and Figure 1-2. The data records are not in a particular order. The index, however, is always in a specific order. In this case, it is in order by employee number.

Using Keys

To find a record, you supply the key value for which you are searching. The **C-ISAM** function rapidly performs the search by looking through the index. If it finds a match on the key value, it uses the pointer to read the data record. **C-ISAM** then returns the data record to your program.

Your program does not need to know where the record is in the data file. It needs only to supply the search value to a function. If you provide a search value of 100 and use the Employee Number index, the **C-ISAM** function locates the record corresponding to Employee Number 100, regardless of where it is in the file.

Choosing a Key

You may also need to find specific records in the **employee** file by employee name. Once again, this requires an index. The choice of the key, in this case, is a little more complex since the record contains two name fields: First Name and Last Name. You can define the key to include any one of the following fields:

- Last Name field only
- First and Last Name fields together, in the form last/first
- Some other combination, such as the first 10 characters of the Last Name field and the first character of the First Name field

The key that you choose determines the order of the index.

The search value that you use to find a record is different for different key definitions. For example, if you define the key on the first ten characters of the Last Name field and the first character of the First Name field and you are looking for an exact match, a search value of `Smith` cannot find the desired data record if you are looking for the record that belongs to R. Smith.

Key Descriptions

Each index has a description of its key. This key description defines the fields that make up the key. For the Employee Number index, the key description indicates that the keys consist of only one field, the Employee Number. For the Name index, the key description is more complex. If you choose to use the first 10 characters of the Last Name field and the first character of the First Name field as the key, the key description specifies that the keys consist of two fields: part of the Last Name (the first 10 characters) and part of the First Name (the first character).

C-ISAM does not keep information about the names or uses of individual fields. A field is simply a location in the record that is defined by its offset from the beginning of the record. You use the offsets to identify the fields that define the key. For the employee record, these field offsets are shown in Figure 1-1.

You identify the key fields to **C-ISAM** by creating a *key description structure* that contains information about the key. This includes the number of parts that the key contains (one for Employee Number key and two for the Employee Name key) and information about each part. The information for each part of the key includes the offset of the field in the data record, the data type, and the length. You can specify several other options in the key description structure. (Chapter 2 explains these options.)

Unique and Duplicate Keys

You may want a field in each record to uniquely identify that record from all other records in the file. For example, the Employee Number field is unique if you do not assign the same number to two different employees, and you never reassign these numbers to other employees. If you wish to find or modify the record belonging to a specific employee, this unique field saves the trouble of determining whether you have the correct record.

If you do not have a unique field, you must find the first record that matches your key and determine whether that record is the one you want. If it is not the correct one, you must search again to find others.

If you know that you have a unique field within your records, you can include this fact in the key description, and C-ISAM will allow only unique keys. For example, if you specify that the employee numbers are unique, C-ISAM only lets you add records to the file for, or change numbers to, employee numbers that do not already exist in the file.

There are times when you do not want to specify a key as unique. If you want an index on Employee Name, you may want to allow for duplicate keys in the event that two or more employees have the same name, for example, two R. Smith. If you use this index to find and update a record, however, you must determine that only one R. Smith exists in the file or that you are updating the correct record if there is more than one.

Primary Keys

When you create your C-ISAM file, you ordinarily specify a description of the key in the index. The keys in this index are called *primary keys*. This index is the *primary index*. Other non-primary indexes can be added later. Chapter 2 discusses how to add indexes.

In general, very little difference exists between a primary index and any other. The primary index, however, cannot be deleted. Also, several functions work only on records that have unique primary keys. These functions are described in the section “Manipulating Records in C-ISAM Files” on page 1-18 of this chapter.

Usually you want to build your primary index on a key that you are most likely to need throughout the life of the file, especially if it is a unique key. It is possible to build a C-ISAM file that does not have a primary index. Chapter 2 also discusses this option.

Organization of C-ISAM Files

Each **C-ISAM** file contains data records and, usually, one or more indexes that point to the data records. Even if there are two indexes for the employee file, one on Employee Number and the other on Employee Name, there is still only one data record for each employee. If R. Smith is Employee Number 100, the entry in the Employee Number index for key 100 points to the same record as the entry for employee R. Smith in the Name index.

Physically, a **C-ISAM** file consists of two operating system files, one to hold the data records and another to hold the indexes. The data file has the extension **.dat**, for example, **employee.dat**. The index file has the extension **.idx**; for example, **employee.idx**. These two operating system files are always used together as a logically single **C-ISAM** file. On some platforms, an additional file is used to keep track of locks on data records. This lock file has the extension **.lok**.

Building a C-ISAM File

You must use the **isbuild** function call to create a **C-ISAM** file. If the file is to contain variable-length records is very similar to using files with fixed-length records. With variable-length records, you have to use the global variable **isrecen** in addition to **isbuild** when you build the file.

The call to build the **C-ISAM** file **employee** (a fixed-length record) is as follows:

```
fd = isbuild("employee",84,&key,ISINOUT+ISEXCLLOCK);
```

This function creates the **.dat** and **.idx** operating system files and opens them. It returns a file descriptor, **fd**, which identifies the **C-ISAM** file in other function calls.

The first argument to the function is the **C-ISAM** filename. You do not specify a filename extension.

In the example used here, each record contains an Employee Number, First Name, Last Name, Address, and City field. The layout of the record is shown in Figure 1-5.

Description	Type	Length	Offset from Beginning of Record
Employee Number	LONGTYPE	LONGSIZE	0
Last Name	CHARTYPE	20	4
First Name	CHARTYPE	20	24
Address	CHARTYPE	20	44
City	CHARTYPE	20	64
Total Length in Bytes			84

Figure 1-5

Employee Record

The **isbuild** function does not use any information about the actual organization of the record. You should lay out the record, however, to determine the length of the record and the location of the key within the record.

For the **employee** file example, you must provide **isbuild** with the four parameters shown here:

employee	is the name of the file that is being built, and the first parameter.
84	is the record size, in bytes, in this example.
&key	is the third argument and the address of the structure that describes the primary key. It is, by definition, the primary key since it is the key that you create when you build the file.
ISINOUT+	
ISEXCLLOCK	specifies the mode and locking to be used.

Figure 1-6 shows the key description structure. It is defined in the header file **isam.h**, which you include when you compile your program. (See Appendix B for a complete listing of **isam.h**.)

```
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 */
};
```

Figure 1-6

Key Description Structure

You must set up a **keydesc** structure to define your key. At this point, consider only what is necessary to define the primary index containing employee numbers as keys. Chapter 2 describes in detail how to set up key description structures.

The key description structure **keydesc** defines the number of fields that the key contains and, for each field, gives information about its location in the record, its data type, and the number of bytes that are part of the key. The structure also contains information that is related to the overall key; for example, whether or not duplicate keys are allowed.

The Employee Number index contains keys with only one part, the Employee Number field. In this case, you initialize **k_nparts** equal to one.

As previously mentioned, **C-ISAM** files contain no information about fields in a record. When you choose key fields, you must specify an offset that is the distance in bytes from the beginning of the record to the beginning of the field. This offset depends upon the lengths of the fields that precede the key field in the record. Since the Employee Number field starts at the beginning of the record, the offset is zero; therefore you initialize **kp_start** to zero.

The key length is defined by the data type that you use or the length of the data if it is a CHARTYPE. Since the Employee Number is a C language **long** data type, its data type is LONGTYPE and the length is LONGSIZE. In this case, you set **kp_type** to LONGTYPE and **kp_leng** to LONGSIZE.

If you want C-ISAM functions to enforce uniqueness on the primary key, set **k_flags** equal to ISNODUPS (no duplicates allowed).

After you create the file, it remains open and available for use. The fourth argument to **isbuild** specifies the access mode and locking mode of the open file. You can open the file for output (write only), input (read only), or both input and output. You can also lock the file for exclusive access, which means that only the program that opens the file can use it (until the file is closed).

Figure 1-7 shows the code that you use to create the **employee** file. The access mode allows both input and output. The locking mode, which is ISEXCLLOCK, specifies exclusive use by the program.

```
#include <isam.h>
struct keydesc key;
.
.
.
key.k_flags = ISNODUPS;
key.k_nparts = 1;
key.k_part[0].kp_start = 0;
key.k_part[0].kp_leng = LONGSIZE;
key.k_part[0].kp_type = LONGTYPE;

if ((fd=isbuild("employee",84,&key,ISINOUT+ISEXCLLOCK)) < 0)
{
    printf ("isbuild error %d",iserrno);
    exit (1);
}
.
.
.
```

Figure 1-7

Creating a C-ISAM File

The function returns a code. If this code is greater than or equal to zero, the number is the file descriptor that you use in subsequent C-ISAM calls to uniquely identify the file. If the return code is less than zero, it is an indication of an error.

The file opening modes are discussed in the section “Opening and Closing Files” on page 1-29 of this chapter. Locking is described in Chapter 4, “Locking.”

Building a Variable-Length File

Use the **isbuild** function to create a C-ISAM file for variable-length records.

1. Before you call **isbuild**, set **isreclen** to the minimum number of bytes in the variable-length record. This establishes the length of the fixed-length portion of the record. The total record length can range from 2 to 32,511 bytes; the fixed-length portion can range from 1 to 32,510 bytes.
2. Call **isbuild**, specifying **ISVARLEN** as part of the mode parameter to indicate that the file will contain variable-length records. Give the *reclen* parameter the maximum length of the record, including the fixed- and variable-length parts. The smallest value you can use in **ISVARLEN** is 1. The smallest variable-length record that you can use is two bytes; one byte for the fixed-length portion, one for the variable-length portion.

For example, the following two statements build the C-ISAM file **employee** with a maximum record size of 1284 bytes, a minimum record size of 84 bytes, and a variable-length portion of up to 1200 bytes.

```
isreclen = 84;
fd = isbuild("employee:", 1284, &key, ISINOUT + ISEXCLLOCK +
            ISVARLEN);
```

The **employee** file also is read/write and is locked exclusively. See the complete description of **isbuild** on page 8-15 of this manual.

C-ISAM Error Handling

C-ISAM functions return an integer code. If this code is greater than or equal to zero, the function executed successfully. If the return code is negative, the function failed.

To determine the reason for failure or to test for certain conditions, such as the end of a file, you can examine the contents of the global variable, **iserrno**. Appendix C, "Error Codes," contains a description of all error conditions, their values, and mnemonics.

Figure 1-7, on page 1-16, shows an example of the use of the **iserrno** variable. You should check the return code of each C-ISAM call and take appropriate action based upon the value in **iserrno**.

Manipulating Records in C-ISAM Files

You can manipulate records in a **C-ISAM** file in several ways. When the file is created, you add records. Later you will need to find them again. Perhaps you may also need to delete some of the records and update the contents of others. **C-ISAM** provides several ways to perform each of these operations.

Identifying Records

Several **C-ISAM** functions perform the same task. The differences among these functions are a result of the different ways that you identify records within a **C-ISAM** file. For example, you can delete a record with either of three function calls. The way you identify the record dictates the function that you use.

Using the Key Value

You can identify a record by its key value. If you specify a unique primary key, you can, for example, delete a record using the **C-ISAM** function call **isdelete**.

You can use an employee number with the function **isdelete** to delete a record from the **employee** file, since Employee Number is the unique primary key. (See the section “Building a C-ISAM File” earlier in this chapter, for an example of how to build the **employee** file.)

If you do not use a primary index with unique keys, you cannot use **isdelete** to delete a record. Functions that use unique primary keys guarantee that the record you want is the only possible match. These functions return error codes if the index definition does not guarantee unique keys.

C-ISAM functions give you two other ways to identify records, in addition to an exact match on the key value.

Using the Current Record

You can use functions that operate on the current record. You can set the current record in several ways. The most common way is to read a record, since the last record that you read becomes the *current record*.

If you have keys that are not guaranteed to be unique, a potential solution is to read the first record with a matching key; this becomes the current record. If the user verifies that this is the correct record to delete, your program can delete it with the function call **isdelcurr**, which deletes the current record.

This method is useful, for example, when you have two R. Smiths in the file. The program can read the first record, using the Name index, and display the Address and City. This record is the current record. The program can prompt for verification. If it is the correct record, the program deletes it with **isdelcurr**. If it is not correct, the program can find another match, and the new record becomes the current record. The program can repeat the process.

Using the Record Number

Some functions allow you to identify a record by its position, relative to the beginning of the data file. Each record has a *record number* that identifies its position in the file. The first record in the file is Record 1.

When a record is accessed for any reason, even for deletion, its record number is set in the global variable **isrecnum**. This variable is defined in **isam.h**. You can use the record number with the function call **isdelrec** to delete a record in the file.

Summary of Record Identification Methods

In summary, C-ISAM functions use one of the following three basic methods to identify a specific record:

<i>key value</i>	uses an index to access the record.
<i>current record</i>	is either the last record read or, in certain cases that are discussed in the following sections, is set by another function.
<i>record number</i>	identifies the relative position of the record from the beginning of the data file. (The first data record in the file is Record Number 1.)

Adding Records

To add records to a file, you must first fill your data record structure with the data to be written to the file. If you add a record to the **employee** file, you must fill in the employee record that is defined by the structure, **emprec**. C-ISAM automatically inserts the key into each index that exists.

You can add records to the file using either **iswrite** or **iswrcurr**. The only difference between the two calls is that **iswrcurr** sets the current record to the record just added, and **iswrite** does not. Figure 1-8 shows examples of each call.

```
#include "isam.h"
.
.
.
int fd;

char emprec[85]; /* C-ISAM Record */

char *p_empno = emprec+ 0; /* Field Definitions */
char *p_lname = emprec+ 4;
char *p_fname = emprec+24;
char *p_eaddr = emprec+44;
char *p_ecity = emprec+64;

/* Program Variables */
long empno;
char lname[21];
char fname[21];
char eaddr[21];
char ecity[21];
.
.
.
/* Store program variables in C-ISAM data record */
stchar (lname,p_lname,20);
stchar (fname,p_fname,20);
stchar (eaddr,p_eaddr,20);
stchar (ecity,p_ecity,20);

stlong(100L,p_empno); /* Employee No. 100 */

if (iswrite(fd,emprec) < 0)
```

```

    {
        printf ("iswrite error %d",iserrno);
        .
        .
    }
    else /* current record position not changed */
    {
        printf("The current record is NOT %d",isrecnum);
        .
        .
        stlong(101L,p_empno); /* Employee No. 101 */

    if (iswrcurr(fd,emprec) < 0)
        {
            printf ("iswrcurr error %d",iserrno);
            .
            .
        }
    else /* this record is the current record */
        {
            printf("The current record is now %d",isrecnum);
            .
            .
            .
        }
    }

```

Figure 1-8

Adding Records to a C-ISAM File

The file descriptor, **fd**, is returned when you execute **isbuild** or when you open an existing file. Both **iswrite** and **iswrcurr** update the Employee Number index. They also update any other indexes that exist. Both functions set the global variable **isrecnum** to the record number of the data record just added.

Deleting Records

You can use three functions to remove a record from a C-ISAM file. All of them remove the corresponding key value for each existing index.

The **isdelete** function removes the record that is located by its key in the unique primary index. Figure 1-9 shows an example that deletes an **emprec** record from the file created in Figure 1-7.

```
char emprec[85]; /* C-ISAM Record */

char *p_empno = emprec+ 0; /* Field Definitions */
char *p_lname = emprec+ 4;
char *p_fname = emprec+24;
char *p_eaddr = emprec+44;
char *p_ecity = emprec+64;

int fd;
int cc;
/* Set up key to delete Employee No. 101 */
stlong(101L,p_empno);

cc = isdelete(fd,emprec);
```

Figure 1-9

Deletion Using the Primary Key

The primary index must contain unique keys. (You set **k_flags** = ISNODUPS when you build the file.) You must place the key value in the data record in the positions defined for the primary key. The **stlong** function places a **long** integer in the data record.

cc is an integer that receives the return code. If it is negative, you can check **iserrno** to determine the reason. The file descriptor **fd** is the number of the file descriptor that identifies the file.

To delete the current record from the file identified by file descriptor **fd**, use the following call:

```
cc = isdelcurr(fd);
```

The current record is either the last record read, or it is set by some other function, for example, **iswrcurr**.

To delete the 100th record from the beginning of the file, or Record Number 100, use the following call:

```
cc = isdelrec(fd,100);
```

The first argument is the file descriptor that identifies the file. The second argument is a **long** integer that is the record number.

In all cases, C-ISAM sets the record number, **isrecnum**, to the position that held the deleted record.

Updating Records

You can use three functions to modify records that exist in the data file.

The **isrewrite** function changes the record that is located by its key in the primary index. The primary index must contain unique keys. (Figure 1-7.) The key value must be placed in the data record in the positions defined for the primary key. Figure 1-10 shows an example of the **isrewrite** function call.

```
.
.
.
char emprec[85]; /* C-ISAM Record */

char *p_empno = emprec+ 0; /* Field Definitions */
char *p_lname = emprec+ 4;
char *p_fname = emprec+24;
char *p_eaddr = emprec+44;
char *p_ecity = emprec+64;

int fd;
int cc;
.
.
.
/* You must either read the emprec record or set up
   all of the items in the record */

                                /* Item to be changed */
stchar("San Francisco",p_ecity,20);

                                /* Primary key cannot change */
cc = isrewrite(fd,emprec);
.
.
.
```

Figure 1-10

Using the Primary Key to Update the Record

You cannot change the primary key. Any other part of the record can change, and C-ISAM updates any other index that exists if the index key value changes.

The **isrewcurr** function rewrites the current record. All key values, including the primary key, can change and **C-ISAM** updates all indexes where required. An example of the call follows:

```
cc = isrewcurr(fd,emprec);
```

The **isrewrec** function rewrites the record that is identified by its record number. This function also updates all indexes that change, including the primary index. An example of a call that rewrites the 404th record from the beginning of the file follows:

```
cc = isrewrec(fd,404L,emprec);
```

Finding Records

Several ways to find records in a **C-ISAM** file are available. To find a specific record, for example, the record belonging to employee 100, you can use the statements that appear in Figure 1-11.

```
.
.
.
stlong(100L,p_empno);
if (isread(fd,emprec,ISEQUAL)<0)
{
    if (iserrno == ENOREC)
        printf ("record not found");
.
.
.
.
```

Figure 1-11

Using a Key to Find an Exact Match

The function **isread** uses an index to locate and read the record with Employee 100 as the key. You must place the key value for the search in the record at the position defined for the key. The third argument is the mode in which you want to conduct the search. In this case, **ISEQUAL** specifies an exact match on the Employee Number.

If **isread** finds the record with a matching key, it returns the record in the same structure or variable that you used to pass the key to the function, in this case **emprec**. If a record with the desired key is not found, the return code

is negative. A negative code indicates an error. You can use the global variable **iserrno** to determine the reason for the error. If the value of **iserrno** is **ENOREC**, a record matching the key cannot be found.

If **isread** finds a locked record, the current record pointer and the contents of the global variable **isrecnum** remain unchanged from the last **isread** call. If you want to skip locked records, you can use the **ISSKIPLOCK** option of **isread**. (See Chapter 4 for more information about locking records. See the description of **isread** in Chapter 8 for more information about reading past locked records.)

You can specify one of several modes to search for records. Use **ISEQUAL** when you want an exact match. When you successfully call **isread**, the record returned is the current record.

You may retrieve records in relation to the current record by changing the mode. **ISNEXT** specifies retrieval of the next record in key sequence. **ISPREV** causes **isread** to retrieve the previous record relative to the current record, as determined by the index. Each call to **isread** changes the current record to the one just retrieved.

Two search modes, **ISFIRST** and **ISLAST**, specify an absolute position in the index. **ISFIRST** reads the record for the first key in the index. **ISLAST** reads the last record in the order of the index.

If you want to process the entire C-ISAM file in ascending key order, call **isread** with the **ISFIRST** mode and make subsequent calls using the **ISNEXT** mode. If you wish to process in descending key order, use the **ISLAST** mode to read the last record and the **ISPREV** mode during subsequent calls to retrieve the previous record.

If you want to locate a starting position in the file for processing and do not know the exact key, you can use **ISGREAT** (greater than the specified key) or **ISGTEQ** (greater than or equal to) for the mode parameter. Figure 1-12 shows

an example of a search where the program reads the file sequentially by employee number from the first employee with a number greater than or equal to 200.

```
/* Read entire file on or after Employee No. 200 */
stlong(200L,p_empno);
if (isread(fd,emprec,ISGTEQ) >= 0)
{
    while (iserrno != EENDFILE)
    {
        .
        .
        .
        cc = isread(fd,emprec,ISNEXT);
    }
    .
    .
    .
```

Figure 1-12

Sequential Search of Part of the employee File in Employee Number Order

The **stlong** function places the starting key value into the data record at the position defined for the key. The **iserrno** value of EENDFILE indicates that you attempted to go beyond the beginning or the end of the file.

When you use the ISFIRST, ISLAST, ISNEXT, ISPREV, or ISCURR (current record) mode, you do not have to specify a key value in the data record. These modes read from predetermined locations, either the beginning or end of file, or in relation to the current record.

The retrieval modes are summarized as follows:

ISEQUAL	specifies an exact match on the key value passed to the function.
ISGREAT	specifies the next record with a key value greater than the one passed to the function.
ISGTEQ	specifies either an exact match or, if there is no exact match, the next greater key value.
ISNEXT	specifies the next record, in key sequence, from the current one.
ISPREV	specifies the record immediately preceding the current record, in the key sequence.
ISCURR	specifies the current record, usually the last record read.
ISFIRST	specifies the first key in an index.

ISLAST specifies the last key in an index.

Using the *isstart* Function

The previous retrieval modes use the primary index to locate records because when you open or build the file, the primary index is the *current index*. The current index is the one that you are currently using to locate records. If your C-ISAM file has other indexes, you can find and read records (with **isread**) using the keys of another index after you choose the index with the **isstart** function call. The **isstart** function also allows you to choose the starting record in the index.

The following call illustrates the use of **isstart** to choose a current index and the position in the index where retrieval of records is to start:

```
cc = isstart(fd,&key,len,emprec,ISGTEQ);
```

fd	is the file descriptor that is associated with the file during its creation or opening.
&key	is the address of a keydesc key description structure, introduced earlier in the section “Building a C-ISAM File” and explained in detail in Chapter 2. A keydesc structure uniquely identifies a specific index. You call isstart with a pointer to the structure that identifies the index that you want to use.
len	allows you to treat a key as if only part of the key exists when you set the starting key position. For example, a key contains the combination of a 20-byte Last Name field and a 20-byte First Name field, in last name/first name order. If you specify a length equal to 20, this instructs C-ISAM to find the starting key using only the Last Name field, regardless of the contents of the First Name field. A value of 0 for this argument is equivalent to specifying the length of the entire key. Subsequent isread calls use the entire key.
emprec	is used to pass the key value for the ISEQUAL, ISGREAT, and ISGTEQ modes. You use this variable or structure exactly as you use it with isread . The isstart function, however, does not return a record.

The **isstart** function call sets the starting position in the index using the key passed in the record, **emprec** in this case, and the mode. The key value must be in the same positions as specified in the **keydesc** structure that defined the index. You do not need to define the remainder of the record.

ISGTEQ is the mode used to locate the starting record in this example. The **isstart** function call positions the index at the first record that is equal to or greater than the key in **emprec**. To read this record, call **isread** with the ISCURR (current record) mode.

The allowable modes are ISEQUAL, ISGREAT, ISGTEQ, ISFIRST, and ISLAST. They are the same modes that you use with the **isread** function call.

Finding Records by Record Number

To find records using their relative position in the file, use **isstart** to specify access in record number order. Figure 1-13 shows an example of code that sets the access mode of a C-ISAM file to retrieve records by record number.

```
#include <isam.h>
struct keydesc pkey;
.
.
.
/* Read record number 500 */
pkey.k_nparts = 0; /* choose physical order */

isrecnum = 500L; /* set record number to first
                  record to be processed */

cc = isstart(fd,&pkey,0,emprec,ISEQUAL);
if (cc >= 0)
    if (isread(fd,emprec,ISCURR)<0)
    {
        printf ("read error %d\n",iserrno);
        .
        .
    }
    else
    .
    .
    .
```

Figure 1-13

Finding Records in a C-ISAM File

You set this retrieval mode by calling the **isstart** function with a pointer to a **keydesc** structure where **k_nparts** is set equal to zero. The number that you place in the global variable **isrecnum** determines the starting position in the file.

Opening and Closing Files

When you create a C-ISAM file using **isbuild**, the file remains open and available for use. When you have finished using the file, you should close it with **isclose**. An example follows:

```
cc = isclose(fd);
```

where **fd** is the file descriptor that was returned when **isbuild** created the file.

If you close a C-ISAM file and want to use it again, you must open it with **isopen**. The following statement opens the file that was created in Figure 1-7.

```
fd = isopen("employee", ISINOUT+ISMANULOCK);
```

employee	is the name of the file that you are opening.
fd	is a file descriptor that identifies the file employee . If isopen fails, fd contains a negative value.
ISINOUT	is the mode that specifies the access and the locking. In this example, read-write access is specified.
ISMANULOCK	specifies either no locking or manual locking. Use ISMANULOCK if you are not concerned about conflicts between programs that access the same file or records simultaneously, or you want to perform locking under the control of your program.

Figure 1-14 shows all of the allowable access modes.

Mode	Description
ISINPUT	File is read-only
ISOUTPUT	File is write-only
ISINOUT	File is read or write

Figure 1-14

Access Modes for `isopen` and `isbuild`

Opening a File in Exclusive Mode

Certain functions require that the file be open in exclusive mode so that only your program can access the file. You can do this by specifying the exclusive lock option, `ISEXCLLOCK`, along with the access mode, as the following example shows:

```
fd = isopen("employee", ISEXCLLOCK+ISINOUT);
```

See Chapter 4 for a discussion of locking options.

Opening a Variable-Length File

When you open a file that uses variable-length records, specify `ISVARLEN` as part of the mode parameter. When you open a file with `ISVARLEN`, the global variable `isreclen` is set to the maximum length of the record. If you do not specify `ISVARLEN` with variable-length records, **C-ISAM** tries to open the file as though it contains fixed-length records. See the complete description of `isopen` on page 8-40 of this manual.

If you want to open a file but you do not know if it contains variable- or fixed-length records, open it one way and if it fails, open it the other way. In Figure 1-15, the file **employee** is first opened as a fixed-length record file. If that **isopen** fails, the mode is reset to include **ISVARLEN** and **isopen** is called again.

```
varlen = FALSE; /* Flag indicating if file is VARLEN */
mode = ISINOUT + ISMANULOCK;
fd = isopen(employee, mode); /*Try opening file as FIXLEN*/
if (fd < 0)
{
    mode += ISVARLEN;
    fd = isopen(employee, mode); /* Try opening file as VARLEN */
    if (fd < 0)
    {
        printf ("isopen failed"); /* Open really did fail */
        exit(-1);
    }
    varlen = TRUE;
    maxlen = isreclen;
}
```

Figure 1-15

Opening a File with Unknown Contents

Maximum Number of Open Files

You can have up to 64 C-ISAM files open at any one time. An operating system limit on the number of open files, however, may impose a lower limit.

Closing Fixed and Variable-Length Files

You can close your C-ISAM file explicitly with a call to **isclose**. You can also close them implicitly with the **iscleanup** function. You can call **iscleanup** at the end of your program (or at any time) to close all of the files opened by the program.

Compiling Your C-ISAM Program

C-ISAM programs must include the **isam.h** header file. If your program uses the decimal data type (see Chapter 3) you must also include **decimal.h**. (Refer to Appendix B for a listing of these header files.)

You compile the program using your C language compiler and the **C-ISAM** library. Consult your system administrator for the location of the files necessary to compile programs that use **C-ISAM** functions. (Appendix E, “System Administration,” identifies the files that are necessary to compile your programs.)

To compile your **C-ISAM** program, use the following command line:

```
cc buildemp.c -lisam -o buildemp
```

If you use the **lint** utility, specify the **C-ISAM** library as follows:

```
lint buildemp.c -lisam
```

C-ISAM Data File Structure

The file containing the data records has the filename extension **.dat**. The data file contains a series of fixed-length records. You define the record length when you create the file. The records in this file contain only data. The **.idx** file contains all other information about the **C-ISAM** file.

You can use the **isindexinfo** function call to display the characteristics of a **C-ISAM** file and its indexes. Figure 1-16 shows the code to print out the data record length and the number of records in the file.

```
include <isam.h>
struct dictinfo info;
fd = isopen ("employee", ISINPUT+ISEXCLLOCK);
isindexinfo (fd,&info,0);
printf ("record size in bytes=%d",info.di_recsz);
printf ("number of records in the file=%d",
        info.di_nrecords);

isclose (fd);
exit (0);
```

Figure 1-16

Determining Data File Characteristics

The **dictinfo** structure is defined in **isam.h**. For further examples using this structure and the **isindexinfo** function, see the section “Determining Index Structures” in Chapter 2.

The data record has a one-byte terminator that is transparent to your program. Do not include this byte when you determine the length of the record. This terminator is either a new line (octal 12) or a null (octal 0). The null character serves as a delete flag for the record. C-ISAM reuses space from deleted records.

Summary

Each C-ISAM file consists of two operating system files, one for data and another for indexes. This chapter discusses how to perform the following tasks:

- Create a file with **isbuild**
- Add records to a file using **iswrite** or **iswrcurr**
- Remove records from a file using **isdelete**, **isdelcurr**, or **isdelrec**
- Update existing records using **isrewrite**, **isrewcurr**, or **isrewrec**
- Find records or retrieve records, or both, using **isread** and **isstart**
- Open and close files using **isopen** and **isclose**
- Compile your program containing C-ISAM functions
- Determine the record length and number of records in a C-ISAM file.

Indexing

Overview	3
Defining an Index	3
Key Structures	6
Manipulating Indexes	7
Adding Indexes	8
Deleting Indexes	9
Defining Record Number Sequence	10
Determining Index Structures	11
B+ Tree Organization	12
Searching for a Record	15
Adding Keys	16
Removing Keys	21
Index File Structure	22
Performance Considerations	23
Key Size and Tree Height	23
Key Compression	24
Leading Character Compression	25
Trailing Space Compression	26
Duplicate Key and Maximum Compression	26
Multiple Indexes	27
Summary	28



Overview

Indexing allows quick access to specific records in the **C-ISAM** file and creates an order for sequential processing of the file. This chapter discusses **C-ISAM** indexes and covers the following topics:

- How to define an index
- How to add and delete indexes
- How indexes are implemented
- What occurs during index operations
- What you can do to improve index performance

Defining an Index

Chapter 1, “How to Use C-ISAM,” introduces you to **C-ISAM** files and keys, and shows you how to create a **C-ISAM** file using **isbuild**. This chapter continues with examples using the **employee** file. Figure 2-1 and Figure 2-2 show the layout of records in this file.

When you create a file, you also define an index for access to specific records and for sequential processing of the **C-ISAM** file in the key order.

You can only define indexes for the fixed-length portion of a record. If you define indexes for the fixed-length portion of variable-length records, you follow the same procedure as for standard fixed-length records.

Description	Type	Length	Pointer	Offset from Beginning of Record
Employee Number	Long	4	p_empno	0
Last Name	Char	20	p_lname	4
First Name	Char	20	p_fname	24
Address	Char	20	p_eaddr	44
City	Char	20	p_ecity	64
Total Length in Bytes				84

Figure 2-1 Employee Record

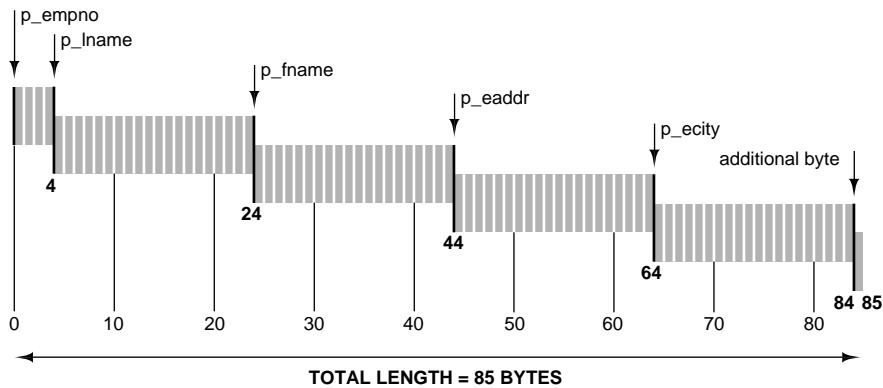


Figure 2-2 Employee Record Illustration

Figure 2-3 shows the code used to build this file.

```
#include <isam.h>
char emprec[85]; /* C-ISAM Record */

char *p_empno = emprec+ 0; /* Field Definitions */
char *p_lname = emprec+ 4;
char *p_fname = emprec+24;
char *p_eaddr = emprec+44;
char *p_ecity = emprec+64;
.
.
.
key.k_flags = ISNODUPS;
key.k_nparts = 1;
key.k_part[0].kp_start = 0;
key.k_part[0].kp_leng = 4;
key.k_part[0].kp_type = LONGTYPE;
.
.
.
if ((fd=isbuild("employee",84,&key,ISINOUT+ISEXCLLOCK)) < 0)
{
    printf ("isbuild error %d",iserrno);
    exit (1);
}
.
.
.
```

Figure 2-3 *Creating a C-ISAM File*

To build the **employee** file with Employee Number as the primary key, you must define the appropriate values in the **keydesc** and corresponding **key-part** structures. (The primary key, by definition, is the key that you define when you build the file.)

The Employee Number index is defined by a key description, which is an occurrence of the structure **keydesc**. (Figure 2-4 shows this structure.) You must use a separate occurrence of a key description structure to define each index. The **keydesc** structure variables define where the key occurs in the record.

This structure is also used to identify each index. For example, when you want to change indexes using **isstart**, you must specify the **keydesc** structure that defines that index. (See the section, “Using the **isstart** Function,” in Chapter 1).

The index shown in Figure 2-3 does not allow duplicate employee numbers. The key consists of only one field, Employee Number, so that the index has only one part. Thus, **k_flags** is set equal to **ISNODUPS**, and **k_nparts** is set equal to 1.

The **keypart** structure is incorporated into **keydesc**. You must have an entry for every part of the key that you define. The maximum number of parts that a key can contain is specified by the parameter **NPARTS**. This parameter is set in **isam.h** and is usually eight.

Since **C-ISAM** does not know about fields in a record, it cannot know what fields, or parts thereof, make up a key. The purpose of each **k_part** is to define a part of the key. All of the parts taken together define the entire key.

The Employee Number index has only one part; therefore you define only the first element of the **keypart** structure, **k_part[0]**.

The Employee Number field starts at the beginning of the record, at offset zero. It is a **long** integer. You set **k_part[0].kp_start** to 0, since this part of the key starts at offset zero from the beginning of the record. You set **k_part[0].kp_leng** to **LONGSIZE** since this is the length of the data type in bytes. You set **k_part[0].kp_type** to **LONGTYPE** since this defines the data type. (Chapter 3, “Data Types,” describes the possible data types and their definitions.)

Key Structures

When you define an index, you define the values that are placed into the key structure. You must use this structure whenever you perform an operation on an index. These operations include building the file, which creates the primary index; changing the index that is used to access records; and adding or deleting indexes.

The C language structures **keydesc** and **keypart** define an index to C-ISAM functions. These structures are shown in Figure 2-4 and are defined in the **isam.h** file.

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

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

Figure 2-4 **Key Description Structures**

The variables within these structures are described as follows:

k_flags	sets one or more of the following flags that may be used to define the index:
ISNODUPS	defines an index that requires unique keys.
ISDUPS	defines an index that allows duplicate keys.
DCOMPRESS	specifies compression of duplicates.
LCOMPRESS	specifies compression of leading characters.
TCOMPRESS	specifies compression of trailing characters.
COMPRESS	specifies maximum compression.

The section “Key Compression” later in this chapter describes compression techniques.

If you use two or more flags, add them together. For example,

```
key.k_flags = ISDUPS+DCOMPRESS;
```

specifies that the index can contain duplicate key values and that they are compressed.

k_nparts specifies the number of parts that the key contains, which ranges between 0 and NPARTS. The **isam.h** file defines NPARTS, which is the maximum number of parts that a key

can contain. (**k_nparts** equal to 0 defines a special case that is explained in the section “Defining Record Number Sequence” later in this chapter.) The maximum key size for all parts is 120 bytes.

k_part is a **keypart** structure that defines each part of the key. Each **keypart** element is composed of the following three items:

kp_start specifies the starting byte in the data record for this part of the key.

kp_leng is the length of this part in bytes.

kp_type is one of the data types described in Chapter 3.

You can add IDESC to the data type parameter to put this part of the key in descending order. To put the Employee Number index in Figure 2-3 into descending order, change **kp_type** as follows:

```
key.k_part[0].kp_type = LONGTYPE+IDESC;
```

Manipulating Indexes

When you create a file, at most one index exists, the primary index. You cannot remove this index until you erase the C-ISAM file. To add the Name index or any other index, you must use the function **isaddindex**. To delete a non-primary index, you use the function **isdelindex**.

C-ISAM allows considerable flexibility for adding and deleting indexes. An operation on an index has no effect on the data records nor on any other indexes that exist. You must open the file exclusively, however, so that no other program can access the file while you are adding or deleting an index. Exclusive access is necessary to prevent conflicts that could arise when another program adds, deletes, or updates records while the index is being added or deleted.

Adding Indexes

You can add indexes at any time; the file does not have to be empty for you to add an index. The larger the file, the longer it takes to add the index since C-ISAM must add a key to the index file for each data record.

Figure 2-3 shows the definition of a key structure for building the primary index. The steps to add another index are similar. You add an index by specifying another key description and using it in a call to **isaddindex**. Chapter 1

describes a Name index consisting of the first 10 characters of the Last Name and the first character of the First Name of the **employee** file. Figure 2-5 shows a **keydesc** structure for this index and a call to **isaddindex** to create the index.

```
#include <isam.h>
struct keydesc nkey;
nkey.k_flags = ISDUPS;
nkey.k_nparts = 2;
nkey.k_part[0].kp_start = 4;
nkey.k_part[0].kp_leng = 10;
nkey.k_part[0].kp_type = CHARTYPE;
nkey.k_part[1].kp_start = 24;
nkey.k_part[1].kp_leng = 1;
nkey.k_part[1].kp_type = CHARTYPE;
if ((fd=isopen("employee", ISEXCLLOCK+ISINOUT)) >= 0)
{
    if (isaddindex(fd, &nkey) < 0)
    {
        printf("isaddindex error %d", iserrno);
        exit(1);
    }
}
else
```

Figure 2-5 *Adding an Index to a C-ISAM File*

This index has two parts, one for each field: Last Name and First Name. It allows duplicate keys. The first part of the index, identified by **k_part[0]**, sets up the Last Name field portion of the key. The second part, **k_part[1]**, defines the First Name field portion of the key.

The starting positions for the name fields are the offsets from the beginning of the record, starting from position 0. (See Figure 2-1 on page 2-4.) The Last Name begins at offset 4 in the record and the First Name begins at offset 24. Put these offsets in the **kp_start** variables.

Both of the fields are data type **char**; therefore the **kp_type** for each one is **CHARTYPE**. (See Chapter 3 for information on **CHARTYPE**.) Each part is in ascending key order since the **ISDESC** parameter is not added to either **kp_type**.

The lengths that you give to **kp_leng** are the size of that part of the key, and not the size of the field itself. In both cases, the size of each part of the key is less than the whole field: 10 characters of the 20-character Last Name field and only the first character of the 20 characters of the First Name field.

You must open the file for exclusive use with **ISEXCLLOCK** before you call the **isaddindex** function.

Deleting Indexes

To delete indexes, define the key description structure for the index that you want to delete and call the function **isdelindex**. You can delete any index except the primary index.

Before you can delete an index, you must first open the file in exclusive mode using **ISEXCLLOCK**. You must specify the same key description structures that you used to create the index. Figure 2-6 shows the code to delete the index created in Figure 2-5.

```
#include <isam.h>
struct keydesc nkey;
nkey.k_flags = ISDUPS;
nkey.k_nparts = 2;
nkey.k_part[0].kp_start = 4;
nkey.k_part[0].kp_leng = 10;
nkey.k_part[0].kp_type = CHARTYPE;
nkey.k_part[1].kp_start = 24;
nkey.k_part[1].kp_leng = 1;
nkey.k_part[1].kp_type = CHARTYPE;
if ((fd=isopen("employee", ISEXCLLOCK+ISINOUT)) >= 0)
{
    if (isdelindex(fd, &nkey) < 0)
    {
        printf ("isdelindex error %d", iserrno);
        exit (1);
    }
}
else
```

Figure 2-6 *Deleting an Index from a C-ISAM File*

Defining Record Number Sequence

You may want to find records based upon the relative location of the records from the beginning of the file. As explained in the section “Finding Records by Record Number” in Chapter 1, you do this by setting **k_nparts** equal to 0 in the **keydesc** structure and then calling **isstart**.

You can specify that the primary index be in record number sequence. In this case, you use the same **keydesc** structure as you did for **isstart**: set **k_nparts** equal to 0. This means that no primary key exists, and whenever you open

the file, the record number defines the key order. If the file has other indexes, you can change the index by calling **isstart** with the appropriate **keydesc** structure.

There is no reason to call **isaddindex** (nor **isdelindex**) with a **keydesc** structure with **k_nparts** equal to 0. You can always process records using the record number, regardless of the indexes that exist.

Determining Index Structures

You can find out which indexes exist in a **C-ISAM** file and determine their structures by using the **isindexinfo** function call. This call has two forms.

You can obtain general information about the file by specifying a **dictinfo** structure and setting the third argument, the index number, equal to 0.

C-ISAM returns the information in this structure:

```
struct dictinfo info;
isindexinfo (fd,&info,0);
```

The **dictinfo** structure is defined in **isam.h** (see Appendix B, “Header Files”). Figure 2-7 shows the structure.

```
struct dictinfo
{
    short di_nkeys;
    short di_recsz;
    short di_idxsz;
    long di_nrecords;
};
```

Figure 2-7 *Dictionary Information Structure*

The variables of this structure are as follows:

di_nkeys	If the file supports variable-length records, the significant bit is set. The remaining bits indicate the number of indexes defined for the file.
di_recsz	This field contains the maximum record size in bytes.
di_idxsz	This field contains the maximum number of bytes in an index node. (Nodes are explained in the section “B+ Tree Organization” on page 2-13.)
di_nrecords	This field contains the number of data records in the file.

The **isindexinfo** function also sets **isreclen** to the minimum size of the record in bytes.

To determine the index characteristics, you must use its index number. The index number of the primary key is 1. The index number of other indexes can change as you add or delete indexes. Figure 2-8 shows how to obtain the characteristics of all the indexes in the **employee** file.

```
#include <isam.h>
struct dictinfo info;
struct keydesc kdesc;
.
.
.
/* get number of keys */
isindexinfo (fd,&info,0);

while (info.di_nkeys > 0)
{
    /* get structure and decrement index number */
    isindexinfo (fd,&kdesc,info.di_nkeys--);
    .
    .
    .
}
.
.
.
```

Figure 2-8 *Determining the Key Structure for All Keys in an Index*

When the program calls **isindexinfo** the first time, with the third argument equal to 0, information about the C-ISAM file is returned in a **dictinfo** structure (the second argument). The **di_nkeys** variable contains the number of indexes that are defined. The program loops, using this variable to determine the index number, and returns the index characteristics for each existing index in the **keydesc** structure.

You should use the technique shown in Figure 2-8 to find a specific index within a C-ISAM file because the index number may change. C-ISAM functions use a key description, not an index number, to identify the index.

B+ Tree Organization

C-ISAM maintains indexes so that programs can find records quickly, and so that it can add, delete, or modify the index keys with minimum impact on the performance of programs that use the file. Programs that use **C-ISAM** files know only which indexes exist and can be used. They know nothing about the actual organization of indexes nor how this organization is maintained and used. You may read this section if you are interested in how the access method is implemented. You do not need this information to use **C-ISAM** functions.

C-ISAM indexes are organized in B+ trees. A *B+ tree* is a set of nodes that contain keys and pointers that are arranged in a hierarchy. A key is a value from the data record; for example, an employee number. The pointer points either to another node in the tree or to a data record. At the top of the hierarchy is the *root* node.

Figure 2-9 illustrates this hierarchy for the Employee Number index. The numbers in the nodes are the Employee Number keys that are also found in the data records. The arrows are the pointers.

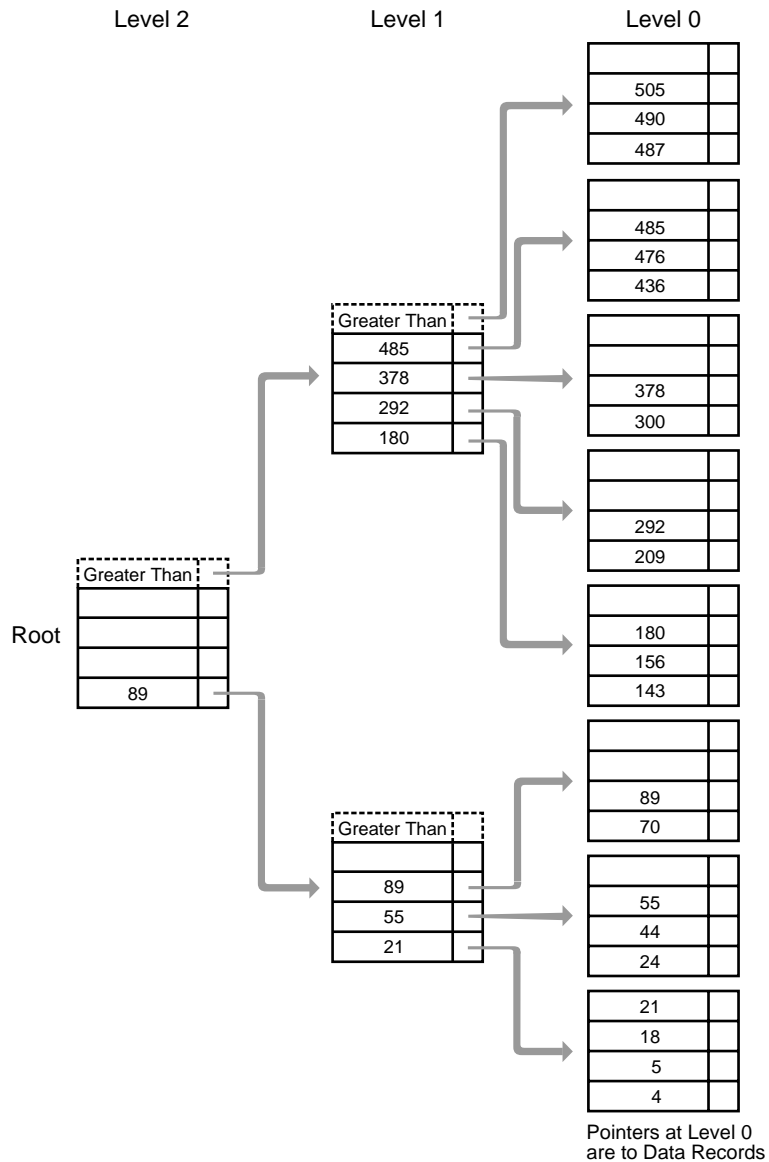


Figure 2-9 B+ Tree Organization

C-ISAM logically organizes the nodes into levels. Level 0 contains a pointer to each data record. At levels higher than zero, the pointer for each key points to a node containing keys that are less than or equal to the key at the higher level.

At levels higher than zero, a node may have an additional pointer that is not associated with a specific key. If it exists, it points to a node that contains keys that are greater than the largest key in that higher level node. A node always has at least as many pointers as it has keys.

Figure 2-9 only shows space for four keys in each node. In reality, **C-ISAM** puts as many keys as possible in each node. The maximum number of keys in different nodes may vary because **C-ISAM** allows keys to vary in length.

Consider the root node in Figure 2-9. It has only one key with the value 89. There are two pointers from the root. One points to a node containing keys with values less than or equal to 89. The other pointer is directed to a node containing keys with values greater than the values in this node, in this case, values greater than 89.

Levels indicate the distance, in nodes, between a node and the pointer to an actual data record. In Figure 2-9, the root node is at Level 2. For nonzero levels, pointers are directed to index nodes at a lower level.

The pointers at Level 0 point to records in the data file; they do not point to nodes in the index file. Every key is represented at Level 0, whether or not it is represented at a higher level.

Searching for a Record

To access a specific record in a **C-ISAM** file, a function starts by comparing the search value with the keys in the root node. The search value is the key that is passed to the function. The function follows the appropriate pointers to the Level 0 node. At Level 0, if a key matches the search value, the key pointer points to the data record. If no match occurs at Level 0, the data record does not exist.

For example, take a search value equal to 44, and use Figure 2-8 to trace the path a function takes to find the record. The function examines the root first and then follows the less-than or equal-to pointer for key 89, since 44 is less than 89. Next, the function examines the node on Level 1 containing keys 21, 55, and 89. The function follows the pointer for key 55, since 44 is less than 55 but greater than 21. The Level 0 node contains keys 24, 44, and 55. Since a match occurs at Level 0, the function finds the data record by following the pointer for key 44.

Repeating the process for search value 475, the function examines the root and follows the greater-than pointer for this node since 475 is greater than 89, the largest key in the node. The node at Level 1 contains keys 180, 292, 378, and 485. The function follows the less than or equal to pointer from key 485 since 475 is less than 485 but greater than 378. At Level 0 the keys are 436, 476, and 485. Since no key matches the search value 475, a data record does not exist.

Adding Keys

When you create the **C-ISAM** file, the index is empty. Figure 2-10 shows a tree that can hold only four keys per node. The first four keys, 18, 143, 414, and 89 are added to the root node. Each key entry points to a data record since the root node is at Level 0.

When the next key is added, with a value of 44, the node is already full and splits to accommodate the new key.

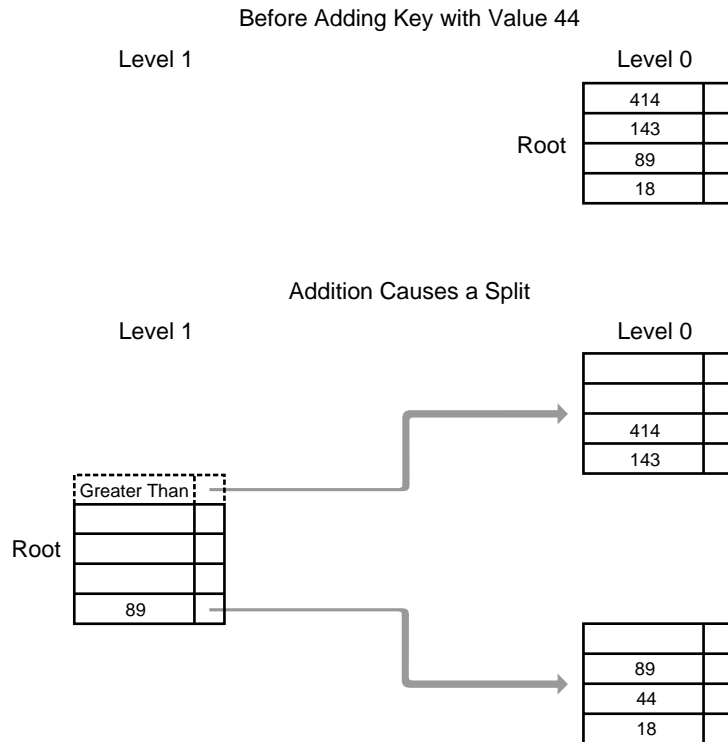


Figure 2-10 Growth of a B+ Tree

C-ISAM splits a node by finding the middle value of the keys in the node, including the value of the key that causes the split. **C-ISAM** puts approximately half the entries into a new node and keeps half the entries in the original node. These two nodes are still in Level 0 after the split, and their keys still point to data records. **C-ISAM** promotes the middle value of the keys, 89 in this case, to the next higher level.

Since no higher level node exists to receive the promoted value, **C-ISAM** creates a new root. The new root node is on Level 1, and the pointer for key 89 points to the original node. (The original node now contains the keys that are less than or equal to 89.) **C-ISAM** forms another pointer directed towards the new Level 0 node. This Level 0 node contains keys that are greater than the highest key value in the next higher level node, in this case 89 in the Level 1 root.

B+ trees grow towards the root from the lowest level, Level 0. Attempting to add a key into a full node forces a split into two nodes and promotion of the middle key value into a node at a higher level. The promotion of a key to the next higher level can also cause a split in the higher level node. If the full node at this higher level is the root, it also splits. When the root splits, the tree grows by one level and a new root node is created.

When a split occurs, approximately half of the entries remain in the original node, and half are transferred to a new node. This process leaves half of each node available to accommodate additional entries. This strategy is useful if the new key values have a random distribution.

If records are added in sequential order, this splitting strategy creates half full nodes that never receive other keys. This means that the effective number of keys per node is approximately half the capacity, and aside from taking more space to store all of the keys, the tree requires more levels to index the same number of data records.

Figure 2-11 shows what happens if you add the key values 415 through 426 sequentially to the tree in Figure 2-10, using the splitting algorithm for the random case.

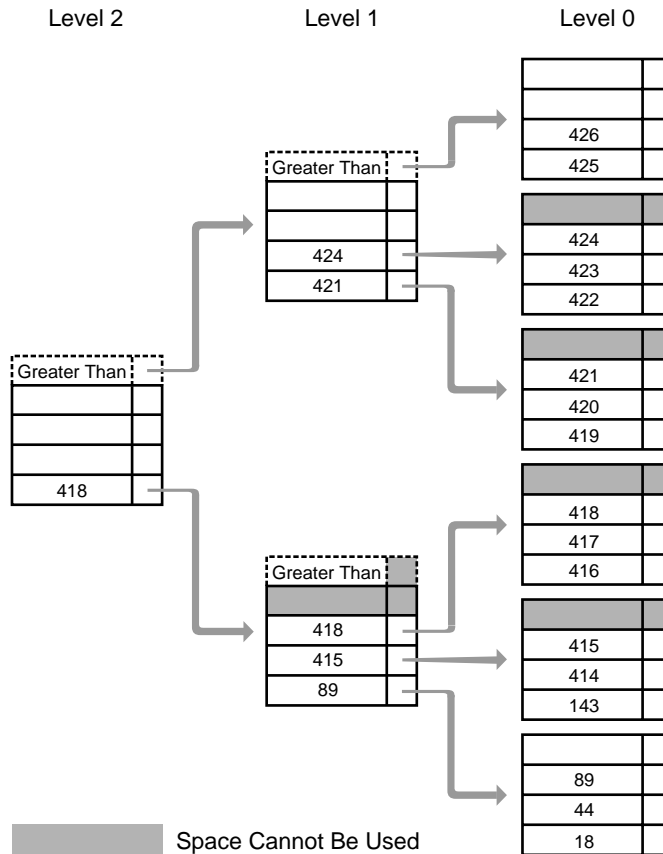


Figure 2-11 Wasted Space in B+ Trees

To avoid this problem, C-ISAM uses a different strategy. If the value that causes the split is greater than the other keys in the node, it is put into a node by itself during the split.

Figure 2-12 shows a split caused by adding key values 415, 416, and 417 to the tree in Figure 2-10.

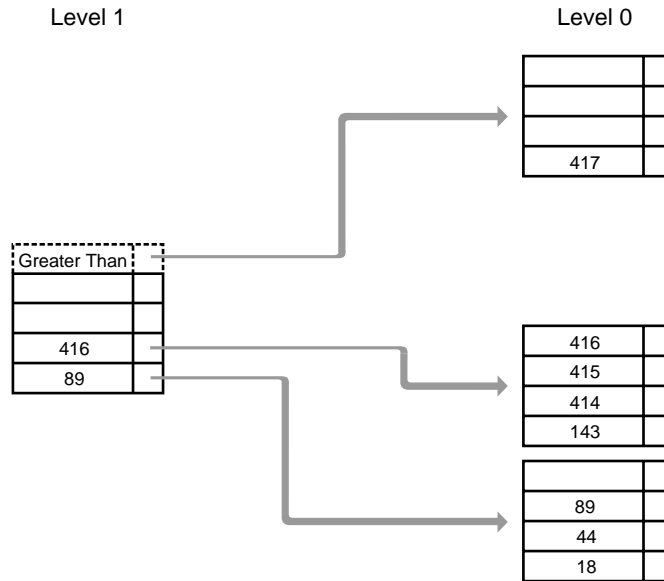


Figure 2-12 *Efficient Growth of B+ Trees*

Figure 2-13 shows the effect of this strategy when key values 415 through 426 are added to this tree.

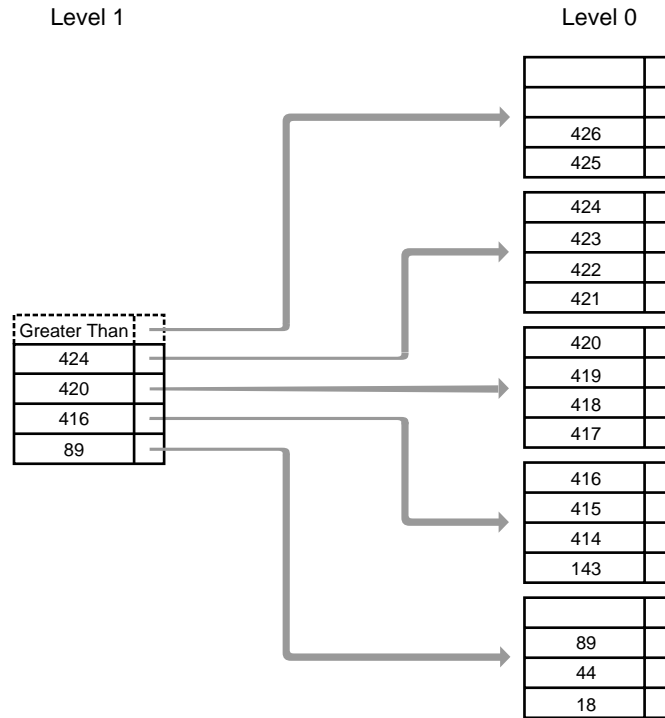


Figure 2-13 *Efficient Sequential Addition of Keys*

Removing Keys

When you delete a record, **C-ISAM** removes the key from the index. If all keys in a node are removed, the node becomes free. **C-ISAM** maintains a list of free nodes (see the following section), and free nodes are reused. **C-ISAM** indexes do not require reorganization.

Index File Structure

C-ISAM stores the index nodes and control information in operating system files with the **.idx** extension. The data file stores only data records.

The index file always contains four kinds of nodes:

- A dictionary node
- Key description nodes
- Index nodes containing keys and pointers
- List nodes

There is usually a one-to-one correspondence between nodes and the unit of transfer between the disk and memory. The unit of transfer is called a *block*. In this discussion, blocks and nodes are interchangeable. Appendix D, “File Formats,” documents the index file nodes.

Each index file has one *dictionary block*. This block contains pointers to the index nodes, as well as other information about the C-ISAM file. Figure 2-14 shows the relationship between the nodes in the index file.

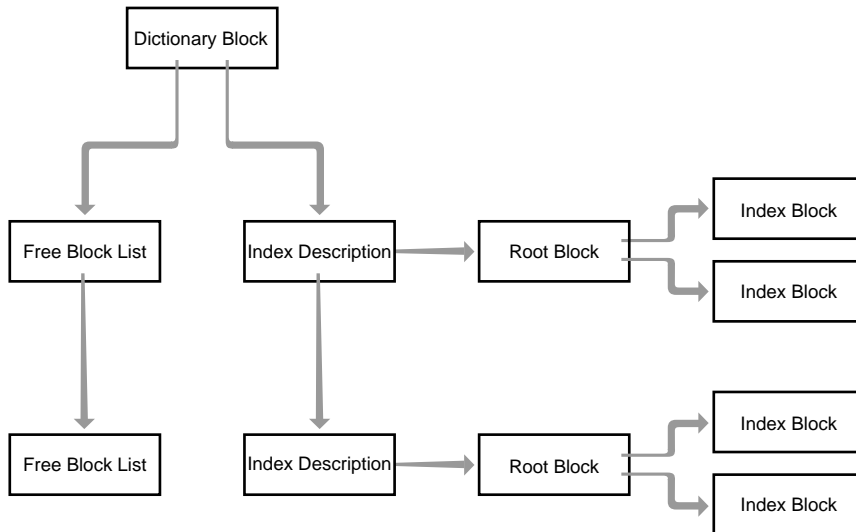


Figure 2-14 Index File Structure

The dictionary block points to the first *key description block*. Each key description block stores information about the indexes, including the address of the root block for each index. All other blocks for an index are addressed via its root block. **C-ISAM** chains key description blocks together, and any index root can be found by following the chain from the dictionary block.

The dictionary block also contains a pointer to the first *free-list block* for the **.idx** file. Free-list blocks are chained together. The free list holds the block numbers that are unused within the file.

When an index block becomes free, **C-ISAM** places the block number on the free list. When a new block is needed, the free list is examined first. The block number of an available block is removed from the list and the block itself is reused. **C-ISAM** uses all free blocks before it extends the length of the file.

Performance Considerations

The choice of key size, the use of compression techniques, and the number of indexes affect the performance of programs that use **C-ISAM** files. This section examines several methods for improving performance.

Key Size and Tree Height

The traversal from one node to another typically requires one disk access. The node size is usually a multiple of the block size of a disk drive, often a one-to-one correspondence. Figure 2-9 shows a diagram representing a B+ tree index. The arrows point to the next node (or block) that must be accessed to find a record. See the earlier section “B+ Tree Organization” for a complete description.

In Figure 2-9, **C-ISAM** requires a maximum of four disk accesses to retrieve the data record, three to traverse the index, and one to fetch the data record. This is a maximum since both the operating system and **C-ISAM** buffer disk blocks in memory, so that a disk access is not required to follow each pointer.

The maximum number of keys that can reside at Level 0 is determined by the number of keys per node and the tree height. The number of levels determines the tree height. If n is the number of keys per node and h is the number of levels, excluding Level 0, the maximum number of keys is equal to $(n+1)^h(n)$. In the index shown in Figure 2-9, the maximum is $(4+1)^2(4)$ or 100.

C-ISAM seldom achieves maximum packing of keys into nodes because additions split nodes into half-full nodes. Deletions also reduce the number of keys in a node. (In most cases, it is also undesirable to have 100 percent pack-

ing of nodes since, if that were possible, every record added would cause a split.) Seventy-five percent of the maximum is a more desirable packing density.

As more records are added, the height of the tree grows. If the tree in Figure 2-9 grows another level, the file might hold 158 records, or $[(.75)(4+1)]^5 (.75)(4)$.

C-ISAM puts as many keys as possible into a node. More realistically, since the keys in Figure 2-9 are short integers requiring six bytes for key and pointer, at least 169 keys can fit into a 1,024 byte node (along with other required information). In two levels, **C-ISAM** can index about $[(.75)(169+1)]^2 (.75)(169)$ or more than 2 million keys.

C-ISAM places as many keys as possible into a single node to reduce the tree height and, consequently, to reduce the number of disk accesses required during a function call. The smaller the key size, the greater the number of records that can be placed into a node. Thus, more records can be accessed in fewer disk operations.

You should consider limiting the key size of your indexes to the minimum that allows you to access the records, without creating too much ambiguity. For example, you can define the Name index of the **employee** file with the entire Last Name and First Name fields of the key. The key size, in that case, is 40 bytes. Alternatively, if you take only 10 characters of the Last Name field and one character of the First Name field, the key size is 11 bytes.

The second choice introduces ambiguity wherever employees have the same last name, or different last names that exactly match on the first 10 characters and the same first initial. If this ambiguity is acceptable, choosing the index with the shorter key significantly increases the number of keys that can be placed in a node.

Key Compression

C-ISAM can compress key values held in indexes. Reducing the key size generally enhances performance. This improvement is more dramatic if the key is more than eight characters long or if duplicate values and leading duplicate characters, trailing blanks, or both, make up a large percentage of the keys. You specify key compression by adding one or more of the following parameters to the **k_flags** element of the **keydesc** structure:

- | | |
|------------------|--|
| LCOMPRESS | specifies removal of leading duplicate characters from the keys in an index. |
| TCOMPRESS | specifies removal of trailing spaces from keys. |

DCOMPRESS specifies removal of duplicate key entries from the index.

You can use any combination of compression techniques. For example, to specify duplicate value and trailing blank compression, set **k_flags** equal to DCOMPRESS + TCOMPRESS + ISDUPS. (It does not make sense to specify duplicate compression unless you define the index to allow duplicates.) COMPRESS specifies that all three techniques are used.

Key compression creates some processing overhead. Generally, compression of non-character keys or keys that are eight bytes or less does not have a positive effect on the performance of programs using C-ISAM files.

Leading Character Compression

Leading character compression reduces the key size by removing all leading characters that are identical with the previous key in the index. The number of bytes that are compressed out of the key is recorded at the beginning of the key.

Figure 2-15 shows an example of this compression technique. The one-byte overhead required to record the number of leading characters compressed is shown as a pound sign (#). The dots (.) represent spaces. If this illustration is representative of the entire index, the compression results in a 5.5 percent savings.

Key Value	Compressed with LCOMPRESS	Bytes Saved
Abbot.....	#Abbot.....	-1
Able.....	#le.....	1
Acre.....	#cre.....	0
Albert.....	#lbert.....	0
Albertson.....	#son.....	5
Morgan.....	#Morgan.....	-1
McBride.....	#cBride.....	0
McCloud.....	#Cloud.....	1
Richards.....	#Richards.....	-1
Richardson.....	#on.....	7

200 bytes	189 bytes	11 bytes
Savings = 5.5 %		

Figure 2-15 *Leading Character Compression, k_flags=LCOMPRESS*

Trailing Space Compression

This compression technique removes trailing blanks from each key. The number of characters compressed is stored in one byte at the beginning of the key.

Figure 2-16 shows an example of this compression technique combined with leading character compression (**k_flags**= TCOMPRESS + LCOMPRESS). The one-byte overhead required to record the number of trailing spaces is shown as a pound sign (#). This byte is in addition to the byte required in the key entry to hold the number of leading characters that are compressed. The dots (.) represent spaces. If this illustration is representative of the entire index, the compression results in a 67.5 percent savings.

Key Value	Compressed with LCOMPRESS + TCOMPRESS	Bytes Saved
Abbot.....	##Abbot	13
Able.....	##le	16
Acre.....	##cre	15
Albert.....	##lbert	13
Albertson.....	##son	15
Morgan.....	##Morgan	12
McBride.....	##cBride	12
McCloud.....	##Cloud	13
Richards.....	##Richards	10
Richardson.....	##on	16

200 bytes	65 bytes	135 bytes
		Savings = 67.5 %

Figure 2-16 *Leading Character and Trailing Blank Compression*

Duplicate Key and Maximum Compression

Duplicate compression (DCOMPRESS) removes duplicate keys from the index. A two-byte duplicate flag replaces the key.

COMPRESS is a shorthand way of specifying maximum compression using duplicate key compression, leading character compression, and trailing blank compression.

Figure 2-17 shows an example using COMPRESS. Two overhead bytes are associated with each non-duplicate key: one to hold the number of leading characters that are compressed and the other to hold the number of trailing blanks that are compressed. This overhead is represented by two pound signs

(##). The dots (.) represent trailing spaces. Two bytes are required for a duplicate key value. If this illustration is representative of the entire index, the compression results in a 75 percent savings.

Key Value	Compressed with LCOMPRESS + TCOMPRESS + DCOMPRESS	Bytes Saved
Abbot.....	##Abbot	13
Abbot.....	(duplicate)	18
Abbot.....	(duplicate)	18
Able.....	##le	16
Able.....	(duplicate)	18
Acre.....	##cre	15
Albert.....	##lbert	13
Albertson.....	##son	15
Albertson.....	(duplicate)	18
Morgan.....	##Morgan	12
McBride.....	##cBride	12
McCloud.....	##Cloud	13
Richards.....	##Richards	10
Richardson.....	##on	16
Richardson.....	(duplicate)	18

300 bytes	75 bytes	225 bytes
Savings = 75 %		

Figure 2-17 Maximum Compression

Multiple Indexes

Indexing allows fast access to specific records in a C-ISAM file. Changes to an index, however, require C-ISAM to update the index. Maintenance of the index imposes an overhead on the use of the file.

Adding a record to the C-ISAM file illustrated in Figure 2-9 requires a maximum of five disk operations: three to read the index to determine that the record did not exist, one write operation to update the index, and another operation to add the record to the data file. If two indexes are involved the number of disk operations, in the worst case, can reach nine: four for each index and one for the data record itself.

The root level of the index and the level that the root points to are often in memory, since the operating system buffers the most recently used index blocks. Therefore, two less disk operations are required per update for each index. The overhead is even less when the updates occur in key sequence.

A linear relationship exists, however, between the time to update a record and the number of indexes that **C-ISAM** must update. A file with two indexes requires approximately twice as much time to update as the same file with only one index, and so on.

If your program is designed for on-line operation, you can achieve better performance by limiting the number of indexes that you need to update in real time.

When you need additional indexes, consider creating the index you need before processing, and deleting it after you are finished. For example, use this method if you want to process the file in different orders at the end of each day.

If you are only reading records, or rewriting records without changing any key fields, the number of indexes has no effect on the speed of processing.

Summary

The principle features of **C-ISAM** indexes are as follows:

- **C-ISAM** indexes are organized in fast and efficient B+ trees.
- You can define indexes on one or more fields or their parts.
- You can define ascending or descending order for any part of an index, and you may specify different orders within a key.
- **C-ISAM** does not impose a limit on the number of indexes allowed for a file.
- **C-ISAM** allows duplicate key values.
- You may restrict an index to require unique keys.
- **C-ISAM** allows three compression techniques to increase the efficiency of storing and processing an index.

Data Types

Overview	3
Defining Data Types for Keys	3
C-ISAM Machine-Independent Data Types	4
Defining Data Records	5
Data Types in Variable-Length Records	7
C-ISAM Data Type Conversion Routines	8
Character Data	8
Integer and Long Integer Data	8
Floating-Point and Double-Precision Data	10
DECIMALTYPE Data Type	11
Using DECIMALTYPE Data Type Numbers	11
DECIMALTYPE Data Type Declaration	11
Sizing DECIMALTYPE Numbers	12
Storing and Retrieving DECIMALTYPE Numbers	13
Manipulating DECIMALTYPE Numbers	14
Summary	16



Overview

C-ISAM data types provide machine independence for standard C language data types. This chapter explains how to perform the following operations:

- How to define data types for keys
- How to use the machine-independent C language data types and the functions to manipulate them
- How to use a data type that stores decimal numbers with many significant digits and the functions to manipulate this data type

Defining Data Types for Keys

When you define a record to **C-ISAM**, you do not specify the data type or length of individual fields. **C-ISAM** needs type information only for keys. For example, consider the Employee record shown in Figure 3-1.

Description	Type	Length	Offset
Employee Number	LONGTYPE	LONGSIZE	0
Last Name	CHARTYPE	20	4
First Name	CHARTYPE	20	24
Address	CHARTYPE	20	44
City	CHARTYPE	20	64
Total Length in Bytes		84	

Figure 3-1

Employee Record

You must specify each part of the key by setting up a **keydesc** structure that contains the location in the record of each part of the key, its data type, and the length of the part. If Employee Number is the key, you must specify that it start at the beginning of the record (offset 0) that it is a **C-ISAM** long integer, LONGTYPE; and that its size is LONGSIZE, the size of a **C-ISAM** long integer.

You identify the data type and size using the parameters that are defined in the **isam.h** file. The values and their mnemonics are shown in Figure 3-2.

C Language Data Type	Data Type Parameter	Data Type Mnemonic	Size Parameter	Size Mnemonic
char	0	CHARTYPE	—	—
int	1	INTTYPE	2	INTSIZE
long	2	LONGTYPE	4	LONGSIZE
double	3	DOUBLETTYPE	size of (double)	DOUBLESIZE
float	4	FLOATTYPE	size of (float)	FLOATSIZE

Figure 3-2

Data Type Parameters

Since **empno** is a **long** integer, you specify the data type as either 2 or **LONGTYPE**, and you define the size as either 4 or **LONGSIZE**. Figure 3-3 shows **empno** defined as a **LONGTYPE** with a size **LONGSIZE**.

```
#include <isam.h>
char emprec[85];
.
.
.
key.k_flags = ISNODUPS;
key.k_nparts = 1;
key.k_part[0].kp_start = 0;
key.k_part[0].kp_leng = LONGSIZE;
key.k_part[0].kp_type = LONGTYPE;
.
.
.
```

Figure 3-3

Setting Up a LONGTYPE Key

If you use any other fields in **emprec** as part of a key, you specify the data type as either 0 or **CHARTYPE**.

C-ISAM Machine-Independent Data Types

C-ISAM stores numbers in a format that is independent of the internal representation of data on your computer.

For example, the word length of your machine usually determines the length of **int** data types. If your machine has a 16-bit word length, an **int** is usually 16 bits long. If your machine has a 32-bit word length, an **int** data type is usually 32 bits long. Using **int** data types can affect where the key is located in relation to the beginning of the record.

Likewise, placing character data in relation to numeric data can affect the position of the key within a record. Most machines require that numbers start on a word boundary. If character data precedes numeric data, the numeric data may be shifted to start on a word boundary. One or more fill bytes can be present between the character data and the numeric data.

C-ISAM stores data in a manner that is independent of any specific machine architecture. This eliminates any confusion surrounding machine-dependent representation of data and locating the position of key fields. It also allows programs to operate without modification on different computers.

The **C-ISAM** data types and their C language equivalents are shown in Figure 3-4.

C-ISAM Data Type	C Language Data Type	Size Mnemonic	Size
CHARTYPE	char	—	—
INTTYPE	int	INTSIZE	2
LONGTYPE	long	LONGSIZE	4
FLOATTYPE	float	FLOATSIZE	sizeof(float)
DOUBLETTYPE	double	DOUBLESIZE	sizeof(double)
DECIMALTYPE	—	—	—

Figure 3-4

C-ISAM Data Types

Note that **C-ISAM** integers always take two bytes, regardless of the internal representation of an integer on your machine.

C-ISAM does not change the representation of **float** and **double** data types. You should consider using the **C-ISAM** **DECIMALTYPE** data type, described in the section “**DECIMALTYPE** Data Type” later in this chapter, as an alternative to **FLOATTYPE** and **DOUBLETTYPE** if you want complete machine independence.

Defining Data Records

Consider the record structure in Figure 3-5.

Field Description	Data Type	Size	Offset From Beginning of Record
Customer Number	LONGTYPE	LONGSIZE	0
Customer Name	CHARTYPE	20	4
Customer Status	INTTYPE	INTSIZE	24
Transaction Amount	FLOATTYPE	FLOATSIZE	26
Account Balance	DOUBLETTYPE	DOUBLESIZE	30
Record Size in Bytes			38

Figure 3-5***Customer Record in a C-ISAM File***

You know the record size and the field offsets because you know the size of each field. (See Figure 3-2 on page 3-4.) The record length does not change from one computer to the next. The location of the fields does not change, regardless of the word length of the machine. A C-ISAM record has the same physical structure on a disk, regardless of the operating environment. Any differences in the way that numbers are stored are hidden from your program.

You do not need to declare the data types of the fields in a record, except when they are part of the key. Drawing the record shown in Figure 3-5, however, helps you to lay out the physical storage and identify the position of keys.

In your program, you should define a **char** variable to receive records from the file and to set up records that are to be put into the file. The variable must be one byte longer than the record size. The following variable declarations are sufficient to reserve space for the record in Figure 3-5:

```
char rec[38+1];
```

or

```
char rec[39];
```

To define the locations of fields within the record, declare a pointer to the beginning of each field. The offset of the field from the beginning of a record defines its position. You can use the offset and pointer arithmetic to declare the pointer. Figure 3-6 shows the pointers for the Customer record shown in Figure 3-5.

```
char rec[39];

char *p_custno = rec;      /* = &rec[0] */
char *p_cname  = rec+ 4;   /* = &rec[4] */
char *p_cstat  = rec+24;   /* = &rec[24] */
char *p_tramt  = rec+26;   /* = &rec[26] */
char *p_acctbal= rec+30;   /* = &rec[30] */
```

Figure 3-6

Field Definitions for the Customer Record

You must have variables to receive the fields after they have been retrieved into **rec**. After the program finishes manipulating these internal variables, it can place them into **rec**. C-ISAM functions that read, write, or update a C-ISAM file use **rec** as the data record argument.

Your program operates on individual variables. Figure 3-7 shows a list that is sufficient to handle the record in Figure 3-5.

```
long   custno;
char   cname[21];
int    cstat;
float  tramt;
double acctbal;
```

Figure 3-7

C Language Variables to Hold the Customer Record

You can define the variables within a structure.

Data Types in Variable-Length Records

Since you cannot place an index on the variable-length portion of a record, you do not need to specify the data type or length of individual fields within the variable-length portion of a record. You can use the **ld** and **st** functions as appropriate to transfer data from a C-ISAM record to a C language variable and back. See the following section “C-ISAM Data Type Conversion Routines” for more information about the **ld** and **st** functions.

C-ISAM Data Type Conversion Routines

Use **C-ISAM** functions to convert between the machine-independent representation of data and the internal representation of data that your program requires when it executes. These functions convert the **C-ISAM** physical representation of the data on a disk to the internal representation of the data that your program requires while it executes. Also use these functions to reconvert the data into machine-independent format when you place the data into a record for transfer to a disk.

Character Data

C-ISAM treats **CHARTYPE** data as bytes, each with a value between 0 and 255. This data is usually treated as ASCII characters.

C-ISAM stores character data in the file as non-terminated strings that are padded with trailing blanks. If your program wants to use strings that are null-terminated without trailing spaces, you can use the functions **ldchar** and **stchar** to transfer data between the variable or structure that contains the **C-ISAM** representation of the string and your program variable.

To transfer data from the record **rec** to the C language variable **cname**, use the next function call:

```
ldchar(&rec[4], 20, cname);
```

To replace the customer name in **rec**, you use the following call:

```
stchar(cname, &rec[4], 20);
```

If you use the pointers in Figure 3-6, the following calls are equivalent:

```
ldchar(p_cname, 20, cname);  
stchar(cname, p_cname, 20);
```

Integer and Long Integer Data

C-ISAM provides two functions for conversion between integers and two functions for conversion between **long** integers.

ldint returns a machine-format integer from the data file record.

stint stores a machine-format integer in a data file record.

ldlong returns a machine-format **long** integer from the data file record.

stlong stores a machine-format **long** integer in a data file record.

These routines are either macros defined in **isam.h** or are in the C-ISAM library. They are described fully in Chapter 8, “Call Formats and Descriptions.”

The code in Figure 3-8 demonstrates the use of data type conversion functions to retrieve and store the Customer Number and Customer Status fields of the Customer record in Figure 3-5.

```
.
.
.
char rec[39]; /* C-ISAM Data File Record */
.
.
.
/* Get Customer Number and Status from Record */
custno = ldlong(&rec[0]);
cstatus = ldint(&rec[24]);
.
.
.
/* Store Customer Number and Status into Record */
stlong(custno,&rec[0]);
stint (cstatus,&rec[24]);
.
.
.
```

Figure 3-8

Conversion of Integers and Long Integers

The C-ISAM machine-independent data types INTTYPE and LONGTYPE consist of two-byte and four-byte binary signed integer data, respectively. C-ISAM integer data is always stored in the data and index 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 memory while the program executes.

Floating-Point and Double-Precision Data

C-ISAM provides four functions for storing and retrieving floating-point numbers and four functions for handling double-precision numbers.

ldfloat	returns a machine-format floating-point number from the data file record.
stfloat	stores a machine-format floating-point number in a data file record.
ldftnull	returns a machine-format floating-point number from the data file record, and checks if it is <code>null</code> .
stftnull	stores a machine-format floating-point number in a data file record, and checks if it is <code>null</code> .
lddbl	returns a machine-format double-precision number from a data file record.
stdbl	stores a machine-format double-precision number in the data file record.
lddblnull	returns a machine-format double-precision number from a data file record, and checks if it is <code>null</code> .
stdbnull	stores a machine-format double-precision number in the data file record, and checks if it is <code>null</code> .

Figure 3-9 shows how these functions are used to retrieve the Transaction Amount and Account Balance fields in the record shown in Figure 3-5.

```
.
.
.
char rec[39]; /* C-ISAM Data File Record */
.
.
.
/* Retrieve Trans. Amt. and Acct. Balance from Record */
tramt = ldfloat(&rec[26]);
acctbal = lddbl(&rec[30]);
.
.
.
/* Store Trans. Amt. and Acct. Balance into Record */
stfloat(tramt,&rec[26]);
stdbl(acctbal,&rec[30]);
.
.
.
```

Figure 3-9

Conversion Routines for float and double Data Types

Both data types can differ in length and format from machine to machine. No difference exists between the floating-point format used by C-ISAM in each operating environment and its counterpart in the C language, except that C-ISAM floating-point numbers are not required to start on word boundaries. If you want to ensure machine independence for floating-point and double-precision numbers, you must represent them as C-ISAM DECIMALTYPE data types.

DECIMALTYPE Data Type

The DECIMALTYPE data type is a machine-independent method for the representation of numbers of up to 32 significant digits, with or without a decimal point, and exponents in the range -128 to +126. You use the parameter DECIMALTYPE to specify a decimal key.

C-ISAM provides routines for converting DECIMALTYPE numbers to and from every data type allowed in the C language. There are also routines that allow compact storage of DECIMALTYPE numbers in a C-ISAM file and conversion from this format to the representation used by an executing program. DECIMALTYPE and CHARTYPE indexes are equivalent within C-ISAM.

Using DECIMALTYPE Data Type Numbers

If your program uses the DECIMALTYPE data type, you must include **decimal.h**. (Appendix B, “Header Files,” contains a listing of **decimal.h**.)

DECIMALTYPE Data Type Declaration

DECIMALTYPE data type numbers have the structure **dec_t**. Your program does not need to know anything about this structure. All operations on the structure are made through function calls.

Consider the **float tramt** and **double acctbal** in Figure 3-7, which hold the Transaction Amount and Account Balance fields. These variables are redefined as DECIMALTYPE data types in Figure 3-10.

```
#include <decimal.h>
.
.
.
dec_t tramt;
dec_t acctbal;
```

Figure 3-10

Defining DECIMALTYPE Data Type Variables

Sizing DECIMALTYPE Numbers

The size of a DECIMALTYPE data type number can vary in the C-ISAM file, depending upon the number of significant digits to the left and to the right of the decimal point. For example, if **tramt** can contain a value of 9,999.99, there are six significant digits.

In memory, you can always use numbers with up to 32 significant digits. DECIMALTYPE data is, however, packed in the C-ISAM file. You must choose the length of the field based upon the number of significant digits that you want to store.

Each byte of a decimal number in the C-ISAM file can hold two digits. Each byte is located either to the right or left of the decimal point. You cannot store a significant digit to the left of the decimal point in the same byte as a digit to the right of the decimal point.

For example, if you want to store numbers less than 100,000 and represent the number to the nearest one-thousandth, you must have space for 10 significant digits, even though the greatest precision that you want to represent is 99,999.999. (Note that the DECIMALTYPE data type with 10 digits allows you to store a larger number with greater precision, or 999,999.9999.)

The file also requires one byte to store the sign and exponent. Therefore, the total number of bytes required to hold a DECIMALTYPE data type number in a C-ISAM file is equal to the sum of the following three items: the number of significant digits before the decimal point, divided by two (and rounded up to the nearest whole byte if necessary); the number of significant digits to the right of the decimal point divided by two (and also rounded up if necessary); plus one more byte.

If you decide to redefine the Transaction Amount and Account Balance fields in Figure 3-5 as DECIMALTYPE numbers, they can hold 6 and 14 significant digits, respectively, in the same space required for the **float** and **double** data types. The new record is shown in Figure 3-11.

Field Description	Data Type	Size	Offset
Customer Number	LONGTYPE	LONGSIZE	0
Customer Name	CHARTYPE	20	4
Customer Status	INTTYPE	INTSIZE	24
Transaction Amount	DECIMALTYPE	4	26
Account Balance	DECIMALTYPE	8	30
Record Size in Bytes			38

Figure 3-11

Customer Record Using DECIMALTYPE Data Type

The decimal point is implied; it is not physically present in either the **dec_t** structure or the data record. You should take care not to perform arithmetic that results in the loss of accuracy. For example, in six significant digits, you can represent 7,777.77 or 333,333. If you add these two numbers together, however you lose accuracy. The result is 341,110, not 341,110.77.

Storing and Retrieving DECIMALTYPE Numbers

In the data file, decimal numbers are stored in a packed format, or two decimal digits per byte. Two functions are provided to convert between the C-ISAM file representation of decimal numbers and the format used during program execution:

- stdecimal** converts a decimal number into packed format and puts it in the data record.
- lddecimal** takes a packed decimal number from the data record and places it in a variable with the structure **dec_t**.

The code in Figure 3-12 demonstrates moving the account balance and transaction amount to and from the data record shown in Figure 3-11.

```
#include <decimal.h>
dec_t tramt;
dec_t acctbal;
char rec[39]; /* C-ISAM Data Record */
.
.
.
/* Load Transaction Amount and Account Balance from Record */
lddecimal(&rec[26],4,&tramt);
lddecimal(&rec[30],8,&acctbal);
.
.
.
/* Store Transaction Amount and Account Balance into Record */
stdecimal(&tramt,&rec[26],4);
stdecimal(&acctbal,&rec[30],8);
.
.
.
```

Figure 3-12

Converting DECIMALTYPE Numbers to and from Record

Format

The **lddecimal** function has three arguments:

1. The location where the DECIMALTYPE data starts in the data record. This is determined by the offset in the record layout in Figure 3-11.
2. The length of the DECIMALTYPE data, not the number of significant digits. (See the previous section, “Sizing DECIMALTYPE Numbers,” for a discussion on how to determine the size of a DECIMALTYPE number in a C-ISAM file.)
3. The address of the **dec_t** structure to receive the DECIMALTYPE number.

The arguments for **stdecimal** are as follows:

1. The **dec_t** structure containing the DECIMALTYPE data
2. The location in the record to receive the data
3. The length of the data as it is represented in the record.

Manipulating DECIMALTYPE Numbers

You must use DECIMALTYPE numbers only with the appropriate C-ISAM functions that manipulate them. For example, you can add two DECIMALTYPE numbers using the function **decadd**. Figure 3-13 shows how to add **tramt** to **acctbal**.

```
#include <decimal.h>
dec_t tramt;
dec_t acctbal;
.
.
.
decadd(&tramt, &acctbal, &acctbal);
.
.
.
```

Figure 3-13

Decimal Addition of acctbal plus tramt

Alternatively, you can convert the numbers to another data type and then perform the calculation. This is shown in Figure 3-14.

```
#include <decimal.h>
dec_t tramt;
dec_t acctbal;
double dtramt;
double dacctbal;
.
.
.
/* convert decimal numbers to double data type */
dectodbl(&tramt,&dtramt);
dectodbl(&acctbal,&dacctbal);

dacctbal += dtramt;

/* convert double to decimal data type */
deccdbl(dacctbal,&acctbal);
.
.
.
```

Figure 3-14

Conversion and Addition of `acctbal+=tramt;`

C-ISAM provides the following C function calls for using DECIMALTYPE numbers:

Function Call	Description
stddecimal	Convert unpacked to packed DECIMALTYPE
lddecimal	Convert packed to unpacked DECIMALTYPE
deccvasc	Convert C char type to DECIMALTYPE
dectoasc	Convert DECIMALTYPE to C char type
deccvint	Convert C int type to DECIMALTYPE
dectoint	Convert DECIMALTYPE to C int type
deccvlong	Convert C long type to DECIMALTYPE
dectolong	Convert DECIMALTYPE to C long type
deccvflt	Convert C float type to DECIMALTYPE
dectoflt	Convert DECIMALTYPE to C float type
deccvdbl	Convert C double type to DECIMALTYPE
dectodbl	Convert DECIMALTYPE to C double type
decadd	Add two DECIMALTYPE numbers

decsub	Subtract two DECIMALTYPE numbers
decmul	Multiply two DECIMALTYPE numbers
decdiv	Divide two DECIMALTYPE numbers
deccmp	Compare two DECIMALTYPE numbers
deccopy	Copy a DECIMALTYPE number
dececv	Decimal equivalent to UNIX ecvt(3)
decfcvt	Decimal equivalent to UNIX fcvt(3)

Chapter 8 describes these calls in detail.

Summary

C-ISAM data types provide machine independence for standard C language data types. In addition, **C-ISAM** provides a DECIMALTYPE data type that allows compact, machine-independent representation of numbers.

C-ISAM provides the following data types:

- CHARTYPE is equivalent to the C language **char** data type.
- INTTYPE is a machine-independent integer corresponding to the C language **int** data type.
- LONGTYPE is a machine-independent long integer corresponding to the C language **long** integer data type.
- FLOATTYPE is a machine-dependent floating-point data type corresponding to the C language **float** data type.
- DOUBLETTYPE is a machine-dependent double-precision data type corresponding to the C language **double** data type.
- DECIMALTYPE is a machine-independent data type, which allows you to represent numbers of up to 32 significant digits with exponents in the range of -128 to +126.

Locking

Overview 3

Concurrency Control 3

Types of Locking 6

File-Level Locking 6

Exclusive File Locking 6

Manual File Locking 7

Record-Level Locking 8

Automatic Record Locking 9

Manual Record Locking 9

Waiting for Locks 10

Increasing Concurrency 11

Error Handling 11

Summary 12

Overview

You can control the access to specific records or files through locking. You should use locking when your program is in the process of updating a record and you need to prevent other programs from updating that same record simultaneously.

You can choose one of the following locking options for a **C-ISAM** file:

- Lock an entire file so that your program has exclusive use of the file
- Lock a file so that other programs can read but not update the records in the file
- Lock a record for the interval between **C-ISAM** function calls
- Lock a record for an interval that is under program control

Variable-length and fixed-length record files use the same procedures for locking and unlocking.

Concurrency Control

Two or more programs can be in a state of execution at the same time on multi-user computer systems. This is called *concurrent execution* or *concurrency*. Only one program executes at any point in time, however. A program can be interrupted after the computer executes any number of instructions. These instructions are the machine language that a C language program creates when it is compiled. The programs execute asynchronously; that is, the execution of a program is independent (in time) of the execution of any other program. You cannot predict when instructions from one program will execute and when instructions from another program will execute.

Generally, concurrent execution of programs is desirable because it allows programs to share the resources of the computer, such as the disk drives and the central processing unit (CPU). Since the utilization of the resources is

higher, concurrent execution improves the overall cost-effectiveness of the system. If the programs are interactive, it appears that your program is the only one executing on the computer.

Since programs execute concurrently on multi-user systems, and the execution can be suspended at any time to allow another program to execute, conflicts between programs can arise if two or more programs operate on the same data records at the same time.

Consider Programs A and B in Figure 4-1. Each operates on the same record. Program A increases the Amount field in the record by 100. Program B increases the Amount field in the record by 200. When both programs finish execution, the Amount field is increased by 300. Since the programs execute concurrently, you cannot predict when instructions for Program A will execute and when instructions for Program B will execute.

Figure 4-1 shows only one possible sequence of interleaved execution of the instructions in which the two programs do not interfere with each other.

Time	Amount Field in Record 1	Program A	Program B
0	2500		
1	2500	Reads Record 1	
2	2500	Adds 100 (in memory)	
3	2600	Writes Record 1	
4	2600		Reads Record 1
5	2600		Adds 200 (in memory)
6	2800		Writes Record 1

Figure 4-1

Concurrent Execution of Programs

Figure 4-2 shows the same two programs operating concurrently to produce an incorrect result. Both orders of execution have the same probability of occurring.

Time	Amount Field in Record 1	Program A	Program B
0	2500		
1	2500	Reads Record 1	
2	2500		Reads Record 1
3	2500	Adds 100 (in memory)	
4	2500		Adds 200 (in memory)
5	2700		Writes Record 1
6	2600	Writes Record 1	
7	2600		UPDATE IS LOST

Figure 4-2***Concurrent Updates Without Locking***

You can prevent conflicts either by not allowing concurrency or by forcing synchronization during the critical points of execution. These critical points exist wherever asynchronous execution of programs can lead to errors.

Locking controls the concurrency so that conflicts do not occur. When the entire C-ISAM file is locked, concurrent program execution cannot occur if the programs need the same file. If records are locked when they are read and unlocked after they are updated, programs that want the locked records must wait until they are unlocked. This forces synchronization so that the update operations on the record are done in a controlled manner by each program.

Figure 4-3 shows Program A locking Record 1. When Program B tries to lock and read the record, the lock request fails, and the program logic specifies that the program wait and try again. After Program A releases the lock, Program B can continue execution.

Time	Amount Field in Record 1	Program A	Program B
0	2500		
1	2500	Reads Record 1 and locks	
2	2500		Reads Record 1 and fails
3	2500	Adds 100 (in memory)	
4	2500		Retries and fails
5	2600	Writes Record 1, releases lock	
6	2600		Retry succeeds, read and lock
7	2600		Adds 0 (in memory)
8	2800		Writes Record and releases lock

Figure 4-3***Concurrent Updates with Locking***

Types of Locking

C-ISAM offers two levels of locking: file-level locking and record-level locking. Both levels provide several ways that you can implement locking.

Locking at the file level prevents any other programs from updating, and perhaps reading, the same C-ISAM file simultaneously. Record-level locking prevents programs from updating (and reading, if ISSKILOCK is used) the same record at the same time. In general, record-level locking allows greater concurrency among programs that access the same C-ISAM files.

The section “Increasing Concurrency” later in this chapter discusses the trade-offs that you should consider when you choose a locking option. Several situations require file-level locking. The next section describes these situations.

Single-user systems do not require locking, since they do not allow concurrent execution of programs. Therefore, conflicts cannot occur. However, your program can always use locking calls for compatibility with multiuser systems; the locking is ignored. A program with locking that is written for a multiuser system runs on a single-user system without modification.

You lock files that have variable-length records just as you lock fixed-length record files.

You must specify a locking mode when you open or build a C-ISAM file. If you do not want locking, or if you want to manually control record level locking, choose the ISMANULOCK option, as shown in the following example:

```
fd = isopen ("employee", ISINOUT+ISMANULOCK);
```

File-Level Locking

C-ISAM provides two types of file-level locking: exclusive and manual. You must specify the file-locking method when you build or open your file.

Exclusive File Locking

If you open or build your file with exclusive locking, no other program can access the file until you close it with the **isclose** function call. This is the only way to remove an exclusive lock.

Figure 4-4 gives an example of instructions to open the file in exclusive mode.

```
.  
.   
.   
fd = isopen("employee", ISEXCLLOCK+ISINOUT);  
  
/*  employee file cannot be used by another  
    program until it is closed          */  
.   
.   
.   
isclose (fd);  
.   
.   
.
```

Figure 4-4

Exclusive File Locking

To lock a file exclusively, add the ISEXCLLOCK parameter to the mode in an **isopen** or **isbuild** function call.

You must use exclusive file locking whenever your program uses the function **isaddindex**, **isdelindex**, or **iscluster** to add or delete an index.

Manual File Locking

Manual file locking allows you to explicitly lock and unlock the file, using C-ISAM function calls. Manual locking only applies to updates of the file. Other programs can read the file while it is manually locked.

The code in Figure 4-5 demonstrates manual file locking.

```
.  
.   
.   
fd = isopen("employee", ISMANULOCK+ISINOUT);  
  
/* file is unlocked  
   until explicitly locked with islock */  
.   
.   
.   
islock(fd); /* file is locked at this point */  
  
/* other programs can read employee records but all  
   other operations on the file are prevented */  
.   
.   
.   
isunlock(fd); /* file is unlocked here */  
.   
.   
. 
```

Figure 4-5

Manual File Locking

Specify the parameter **ISMANULOCK** when you open or build the file. The file is not locked until you make the call to **islock**. Other programs can read records from the locked file. However, they cannot change the **C-ISAM** file, nor can they acquire a lock on the file, until you unlock the file with **isunlock**.

Record-Level Locking

C-ISAM provides two types of record locking: automatic and manual. You must specify the locking method when you open or build your file.

Automatic Record Locking

When you open or build your file with **ISAUTOLOCK**, the record that you read with **isread** remains locked until the next **C-ISAM** function call.

Figure 4-6 gives an example of automatic record locking.

```
#include <isam.h>
char emprec[85];
.
.
.
fd = isopen ("employee", ISAUTOLOCK+ISINOUT);
.
.
.
/* Set up key for Employee No. 100 */
stlong(100L, emprec);
isread (fd, emprec, ISEQUAL);
/* record identified by key in
   emprec is automatically locked */
.
.
.
isrewcurr (fd, emprec);
/* the record is automatically unlocked */
.
.
.
```

Figure 4-6

Automatic Record Locking

You can automatically lock only one record per **C-ISAM** file at a time.

If you use the **ISKEEPLOCK** option with an **isstart** call, the **isstart** call will not unlock any locked record. You can use **isrelease** to release the lock manually.

Manual Record Locking

You must specify manual record locking with the **ISMANULOCK** option when you open or build the **C-ISAM** file. (This is the same option that you use for manual file locking.)

You place a lock on the record when you use the **ISLOCK** option in an **isread** function call. The record remains locked until you execute the **isrelease** function call. The **isrelease** call removes locks for all records that your program locked in the file. Transaction logging affects the time at which locks are released. See the section “Data Integrity” in Chapter 5 for more information.

The operating system determines the maximum number of locked records that you can have. With most implementations of C-ISAM, the operating system determines the maximum number of locked records that you can have. On versions with **.lok** files, the maximum is 64.

The code in Figure 4-7 demonstrates an example of manual record locking.

```
fd_emp = isopen ("employee", ISINOUT+ISMANULOCK);
fd_per = isopen ("perform", ISINOUT+ISMANULOCK);
isread (fd_emp, emprec, ISEQUAL+ISLOCK);
/* employee record is locked here */
isread (fd_per, perrec, ISEQUAL+ISLOCK);
/* performance record is locked here */
isrewcurr (fd_per, perrec);
/* both records are still locked */
isrelease (fd_emp);
isrelease (fd_per);
/* employee and performance records are unlocked */
```

Figure 4-7

Manual Record Locking

Waiting for Locks

If the version of C-ISAM that you have uses the system call **fcntl()**, you can program a process to wait for a locked record. Use the **ISWAIT** option of **isread** to cause the program to wait for the locked record to become free. Use the **ISLCKWT** option with **isread** to cause the program to wait for the record to become free and immediately lock the record, as well. **ISLCKWAIT** is equivalent to **ISLOCK+ISWAIT**.

If your program holds onto one or more locks while it is waiting for another record to become free, your program may become *deadlocked* with another program. A deadlock occurs when two (or more) programs each wait for locks that the other program is holding. To illustrate a deadlock, consider two processes, A and B. Process A locks record 105; process B locks record 200. Process A holds the lock on record 105 and tries to lock record 200; it waits for record 200. Process B is programmed so that it will not release record 200 until it can lock record 105. Since there is no way that either process can get the lock it needs, both processes wait forever. Deadlocks are only possible if your process waits for locks.

Only versions of C-ISAM that use **fcntl()** are X/Open compatible. If a version uses **fcntl()**, it is noted on the media as **SYS5LOCK** or **fcntl** locking.

Increasing Concurrency

Locking allows more than one program to access a **C-ISAM** file concurrently without causing conflicts. For example, a conflict could arise if two programs read the same record and each one updates the record. (See Figure 4-2 on page 4-5.) Locking prevents this by ensuring that once the record or file is locked, no other program can update it or, possibly, even read it.

The locking level affects the degree of concurrency that is possible for access of a **C-ISAM** file. When you use file-level locking, only one program at a time can update the file. If you update Record 100, for example, and another program wants to update Record 200, the second program is not allowed to access the record, even though no actual conflict exists. This is because you locked the entire file. Concurrency is unnecessarily impaired, in this case, since a conflict is not present.

Locking at the record level increases concurrency. Only records that are accessed at the same time are potentially in conflict. Record-level locking ensures that conflicts cannot happen, by preventing concurrent access to these records only and not to the entire file.

Error Handling

Calls to **C-ISAM** functions return a status code. If the function fails, it returns a negative status code. You can check the global variable **iserrno** to determine the reason for failure.

Two values of **iserrno** are related to locking:

- **EFLOCKED** (value 113) indicates that the file is exclusively locked.
- **ELOCKED** (value 107) indicates that either the file, or record within the file, is locked.

Figure 4-3 shows Program B waiting because the record it wants is locked. When the record is released, Program B can continue to execute. Figure 4-8 shows how you can implement a “wait for lock” strategy using a **sleep** function, which delays program execution for one second each time you call it.

```
.
.
.
/* Read and lock record */
readit:
    if (cc = (isread(fd,emprec,ISEQUAL+ISLOCK)) < 0)
    {
        if (cc == ELOCKED || cc == EFLOCKED)
        {
            /* Record is already locked,
             wait 1 second and try again */
            sleep (1);
            goto readit;
        }
    }
    else
    .
    .
    .
```

Figure 4-8

Program That Handles Locked Records

In practice, you may want to retry the **isread** call only a few times, rather than to retry forever.

Summary

C-ISAM supports both file-level and record-level locking. You can lock files or individual records to prevent concurrent update and, in some cases, to prevent concurrent reading of a file.

C-ISAM provides two types of file-level locks:

- **ISEXCLOCK** prevents any other program from accessing the file.
- **ISMANULOCK** allows you to specify when the file is locked for update but allows other programs to read the file.

C-ISAM also provides two types of record-level locks:

- **ISAUTOLOCK** locks a record from one **C-ISAM** call until the next one.
- **ISMANULOCK** allows you to lock specific records and release them under program control.

If you do not want locking, specify ISMANULOCK when you open or build the file.

C-ISAM requires that you open a file with an exclusive lock (ISEXCLLOCK) to add or delete an index.

Transaction Management Support Routines

Overview	3
Why Use Transaction Management?	3
Transaction Management Services	4
Implementing Transactions	4
Transactions with Variable-Length Records	6
Logging and Recovery	7
Data Integrity	8
Concurrent Execution of Transactions	8
Locking	9
Concurrency Issues	10
Summary	11

Overview

There are times when you want to perform multiple operations on a C-ISAM file in such a way that either all of the operations succeed or none of them affect the file. C-ISAM provides support routines for transaction management to implement this strategy. A *transaction* is a set of operations that you want completed entirely or not at all.

Why Use Transaction Management?

Consider the following example. Your program transfers money from one bank account to another. You can write the program to accomplish the transfer in several ways. You can retrieve the account record, deduct the amount, and rewrite the record. Then you can retrieve the account record that receives the money, add the amount, and rewrite the second record. If the second account does not exist, however, you must retrieve the first record again, reverse the entry, and rewrite the record.

A better procedure might be to retrieve both records, make the transfer, and then rewrite both records. You may still encounter a problem if a crash or some other abnormal event occurs after the first record is rewritten but before the second record is rewritten. An inconsistent state results in which either one account has too much money or the other has too little, depending upon the order in which the records were written. In this case, you want to either retrieve the first record written, reset the amount, and rewrite it, or you want to continue updating the second record.

In both cases, either you want to complete the intended action on the records or you want the program to restart from the point of failure. If the operations involve more records or additional files, the interactions between records and files can be more complex. A failure in the middle of processing leaves these records in an unknown, and possibly inconsistent, state. C-ISAM provides an easy way to undo the operations and start over from a state where you know that the records are correct.

Transaction Management Services

The support routines for transaction management enable you to define a set of operations on **C-ISAM** files that you want to be done entirely or not at all. This set of operations is called a transaction.

In the example of transferring money between two accounts, you define a transaction that includes reading and rewriting both records. This kind of transaction defines an undividable unit of work that is either completed entirely or not at all. The transaction cannot be partially completed; thus an inconsistent state cannot result.

Transaction management provides two additional facilities. It provides a recovery mechanism so that, in the event of a crash, the transactions can be recovered automatically, and you can reconstruct the **C-ISAM** files from a backup copy of the files. Transaction management also automatically provides the necessary locking to ensure that two or more transactions do not interfere with each other by updating the same record at the same time.

Implementing Transactions

To define a transaction, you must decide what operations on **C-ISAM** files must be treated as an undividable unit of work. A unit of work is the operations that you want done entirely or not at all. A transaction can involve operations on more than one **C-ISAM** file.

In the example of transferring money between accounts, the unit of work is the complete transfer of funds. The operations that implement the transfer are the reading and rewriting of the **C-ISAM** records that the program updates to effect the transfer.

You implement the transaction by calling the function **isbegin** to mark the start of the operations on the **C-ISAM** files that you want to treat as the unit of work, and by calling **iscommit** to mark the successful completion of those operations. Within the transaction, you can call **isrollback** to cancel the transaction. The **isrollback** call reverses changes to the **C-ISAM** files that the program makes within the transaction.

Figure 5-1 illustrates the function calls that are necessary to add a record to the **employee** file and a record for that employee to another file, **perform**. Assume that you decide to define these two operations as a single transaction to ensure that a record for the employee is added to both files.

```
.
.
.
/* Transaction begins after terminal input has been collected.
   Either both employee and performance record will be added
   or neither will be added. */

/* Files must be opened and closed within the transaction */

isbegin(); /* start of transaction */

fdemploy = cc = isopen("employee", ISMANULOCK+ISOUTPUT+ISTRANS);
if (cc < SUCCESS)
{ isrollback();
  break; }

fdperform = cc = isopen("perform", ISMANULOCK+ISOUTPUT+ISTRANS);
if (cc < SUCCESS)
{ isclose(fdemploy);
  isrollback();
  break; }

cc1 = addemployee();
if (cc1 == SUCCESS)
  cc2 = addperform();

isclose(fdemploy);
isclose(fdperform);

if ((cc1 < SUCCESS) || (cc2 < SUCCESS)) /* transaction failed */
{
  isrollback();
}
else
{
  iscommit(); /* transaction okay */
  printf ("new employee entered\n");
}
```

Figure 5-1

Adding Two Records Within a Transaction

You start a transaction by calling **isbegin** before any other C-ISAM call. You end the transaction by calling **iscommit** after adding both records. If a call to **iswrite** fails, **isrollback** cancels the transaction. You must include the **ISTRANS** parameter in the **isopen** call if you want **isrollback** to reset any changed records to their original state. If you update a file, it is very important that you open and close the file within the transaction.

You can write your program so that any problem that it cannot handle causes the transaction to roll back. Problems can include an error return from a C-ISAM call, application logic that decides the transaction should not be completed, and so forth.

For example, the program may discover any one of the following conditions:

- An account number did not exist.
- The balance was less than zero.
- Another program is using the record.

For any of these problems, the program can roll back the transaction. After **isrollback** executes successfully, the program can retry the transaction starting with another call to **isbegin**.

During the execution of a transaction, the records your program updates are locked. (See the section “Locking” later in this chapter.)

You should, therefore, define a transaction to consist of only the required operations on the records and, where possible, only those operations that execute without user intervention. For example, if your transaction reads and locks a record, and then waits for someone to update it, the record remains locked during that time. Try to minimize the amount of time spent processing inside a transaction since transactions restrict concurrent execution of other programs that need the same records.

You can define recoverable transactions that include the following calls: **isbuild**, **isaddindex**, **isdelindex**, **iscluster**, **isaudit**, **issetunique**, **isuniqueid**, **isrename**, and **iserase**. You cannot, however, roll back their effect.

Transactions with Variable-Length Records

You start and stop transactions with variable-length records in the same way as with fixed-length records. Some of the transaction log formats used with variable-length records are different than those used with fixed-length records. All of the transaction log formats that are used in the transaction logs are listed in “Transaction File Formats” on page -8 of this manual.

If a transaction that contains an update operation reduces the length of a variable-length record, an **isrollback** call will restore the data of the original record to the state it was in at the last **isbegin** call, but the storage location might be different. Therefore, the backup of a file and a file that has been rolled back to the same logical state might not have the same binary image, even though both files contain the same user data.

Logging and Recovery

Each transaction puts records in a log file for the purpose of restoring the **C-ISAM** files if they are rolled back, and to provide a recovery mechanism. The transaction log file is an ordinary operating system file. You should set up procedures to maintain this file. (These procedures include scheduling regular backups of the **C-ISAM** files and purging the log file after each backup and before the transactions are applied to the **C-ISAM** files.)

To set up a transaction log file, you must create an empty log file. You start a new log file after you make backup copies of the **C-ISAM** files that use it. You can do this with the standard C library function **creat**, as follows:

```
creat("recovery.log",0666);
```

Transaction logging starts with the following call to **islogopen**:

```
islogopen("recovery.log");
```

The log file name that you specify in the call must be identical for every program that accesses the same **C-ISAM** file. You cannot recover your **C-ISAM** file if you use different log files.

Every program that is not read-only should call **islogopen**. If a program writes or updates records without using the log, automatic recovery is impossible.

You can close the log file and stop transaction logging with the **islogclose** function call:

```
islogclose();
```

If a **C-ISAM** file becomes corrupted or is destroyed, you can recover it by using the **isrecover** function. The function requires the most recent backup copy of all the files that record their transactions in the same log file and the log file that you started immediately after you created the backup. The **isrecover** function takes the transactions in the log file and applies them to the backup copies of the **C-ISAM** files. You should ensure that no one is using the **C-ISAM** files during the recovery.

Caution: *You should only work with a copy of your backup file, never with the backup file itself. If a failure occurs during the recovery process and you are updating the only backup copy, further attempts at automatic recovery are impossible.*

To ensure successful recovery from a system failure, all **isopen** calls that are intended to open the file in a write mode must be contained in a transaction. The files must also be closed before the transaction is committed or rolled back. If you want a file to be opened in read-only mode and not to be logged in the log file, use **ISINPUT** as the mode on the **isopen** call and do not use **ISTRANS**.

After you run a program that calls **isrecover**, the **C-ISAM** files contain all committed transactions recorded in the transaction log file. This recovery strategy is called *rollforward*. If there are any cases where relative pathnames are used in **isopen** or **isbuild** function calls, be sure that the recovery program is run from the same directory as the original programs.

Data Integrity

Data integrity means you can access data knowing that the data is correct or, at least, consistent.

A transaction defines one or more operations on **C-ISAM** files as a single unit of work. Using transactions ensures data integrity because transactions make it impossible to leave files in logically inconsistent states.

C-ISAM also achieves integrity by providing a recovery mechanism. In the event of a crash, you can recover the transactions.

Concurrent execution of transactions could cause data integrity problems if locking were not present. The following section examines this issue.

Concurrent Execution of Transactions

In a single-process environment, only one transaction executes at a time. The program that executes the transaction either commits all changes to the file, or rolls back without making any changes. After the transaction finishes, the file either reflects the operations contained in the transaction, or the state before the transaction started.

In a multiprocessing environment, it is necessary to prevent two or more transactions from interfering with each other. Interference occurs, for example, if Program A and Program B both read Record 1 and update its contents. If Program B rewrites the record, then Program A rewrites it, the Program B update is lost. This is shown in Figure 5-2.

Time	Amount Field in Record 1	Program A	Program B
0	2500		
1	2500	Reads Record 1	
2	2500		Reads Record 1
3	2500	Adds 100 (in memory)	
4	2500		Adds 0 (in memory)
5	2700		Writes Record 1
6	2600	Writes Record 1	
7	2600		UPDATE IS LOST

Figure 5-2

Concurrent Updates Without Locking

Locking the records that are accessed by a transaction prevents this interference.

Locking

When a transaction begins, all **C-ISAM** function calls that modify a record lock the record. These records remain locked until you execute **iscommit** or **isrollback**. A call to **isrelease** during a transaction only releases unmodified records. Locks on modified records are not released. Likewise, a call to **isunlock** only works if the records in the file are not modified by the transaction.

A transaction that reads a record does not lock the record unless you use the **ISLOCK** option in the **isread** function call. You should use the **ISLOCK** option if the transaction intends to update the record.

The number of record locks that can exist at any one time is operating system dependent. On versions of **C-ISAM** with **.lok** files, the maximum is 64.

You can use the **islock** function call within a transaction to lock an entire file. If you do this, the file remains locked until the end of the transaction.

You should choose an appropriate strategy for handling situations where a **C-ISAM** call returns an indication that a record is locked. (See the section “Error Handling” in Chapter 4 for a description of how locked records are identified.) The safest strategy is to roll back the transaction. This guarantees that transactions will occur in a serial and, therefore, reproducible order.

Concurrency Issues

Locking a record before it is used and holding all locks until the end of a transaction ensure that two or more concurrent transactions cannot interfere with each other. If a transaction wants a locked record, a rollback and one or more retries allow the transaction that holds the lock to finish first. Both transactions are then completed without any unintended interaction.

For example, Figure 5-3 shows Program A and Program B concurrently competing for Record 1.

Time	Amount Field in Record 1	Program A	Program B
0	2500		
1	2500	Reads and locks Record 1	
2	2500		Reads Record 1, fails; rolls back
3	2500		
4	2600	Adds 100 (in memory)	
5	2600	Writes Record 1	
6	2600	Commits	Retries
7	2600		Reads and locks Record 1
8	2800		Adds 200 (in memory)
9	2800		Writes Record 1
			Commits

Figure 5-3

Conflict Resolution with Transactions

Program A reads the record first and locks it. When Program B attempts to read the record, it gets an error. Program B rolls back its transaction and tries again. Meanwhile, Program A commits its transaction. This releases the lock on Record 1, and when Program B tries again, it also succeeds.

To guarantee correct concurrent execution of programs that use transactions, you must use the ISLOCK option with **isread**, even when the transaction is read-only. It is theoretically possible for a read-only program to see the records of a file in a temporarily inconsistent state. The read-only program could read a record that has been changed by a transaction in progress, and then read a record that the same transaction changes later.

Summary

A transaction specifies an undividable unit of work consisting of one or more C-ISAM function calls, operating on one or more files. The following calls implement transactions:

isbegin	marks the beginning of a transaction.
iscommit	marks the end of a transaction and authorizes all changes to the file by a transaction since the last isbegin function call.
isrollback	revokes all changes to the file by a transaction since the isbegin call.
islogopen	opens a transaction log file and starts recording transactions.
islogclose	closes the log file and terminates the recording of changes to the C-ISAM files.
isrecover	uses the transaction log file to restore the file to its original state from a backup copy.

You must include the ISTRANS parameter in the **isopen** function call if you want the ability to roll back the files to their state before you started the transaction.

Additional Facilities

Overview	3
File Maintenance Functions	3
Forcing Output	4
Unique Identifiers	5
Audit Trail Facility	6
Using the Audit Trail	6
Audit Trail File Format	8
Clustering a File	9
File Maintenance with Variable-Length Records	9
If Data Files Are Corrupted	10
If Index Files Are Corrupted	10
Summary	13

Overview

C-ISAM provides several additional facilities that enable you to perform the following tasks:

- Remove or change the names of **C-ISAM** files without having to specify the operating system file names
- Force writing of buffers to the disk
- Define and use a unique field within records that do not already have one
- Create and maintain an audit trail of changes to a **C-ISAM** file
- Put the records of a file into a specific physical order

File Maintenance Functions

You can use the function **isrename** to change the name of the operating system files that comprise a **C-ISAM** file.

A **C-ISAM** file consists of two operating system files that are logically treated as a single unit. For example, when you create the **C-ISAM** file **employee**, two operating system files are created: **employee.dat** and **employee.idx**. (An **employee.lok** file is also created on some platforms.)

The following call renames **employee.dat** to **personnel.dat** and **employee.idx** to **personnel.idx**. If **employee.lok** exists, it too is renamed.

```
isrename ("employee", "personnel");
```

Any other files associated with the **C-ISAM** file, such as a transaction log or an audit trail file, are not affected.

The **C-ISAM** function **iserase** removes the operating system files that comprise the **C-ISAM** file. The following example removes the files **personnel.dat** and **personnel.idx**, and **personnel.lok** if applicable:

```
iserase ("personnel");
```

This function also removes the audit trail file for the **personnel** file if one exists. See the section “Audit Trail Facility” later in this chapter. It does not remove transaction log files.

You can use the function **iscleanup** at the end of your program to close all files opened by your program.

Forcing Output

Ordinarily, **C-ISAM** functions that write records immediately force the output to the operating system and, thus, to the file. You can use the **isflush** function call to force this output; however, an explicit call to flush output is unnecessary except in the following two cases:

- When the file is opened in exclusive mode with **ISEXCLLOCK**
- If you have a single user system that does not support locking

In these cases, the execution of a **C-ISAM** function does not automatically result in output to the operating system, because conflicts in access to the records cannot occur. Therefore, **C-ISAM** keeps the records in memory without forcing them to the operating system. To protect against losing too many records during a crash, you can periodically issue the following call:

```
isflush(fd);
```

fd is the file descriptor that was returned when the file was opened or built.

If you have a multi-user system, and the file is not opened in exclusive mode, you do not have to use the **isflush** function.

Unique Identifiers

C-ISAM provides functions that you can use to set and retrieve unique numbers associated with a **C-ISAM** file. Several **C-ISAM** functions, such as **isdelete** and **isrewrite**, require a unique primary index. If you want to use these functions, in preference to equivalent functions without this primary key restriction, you must specify a unique key field when you build your file.

If your records do not have a reasonably sized field that is guaranteed to be unique, you can add a **long** integer field to them. Define this field as part of the key in your **keydesc** structure. (You must also specify **k_flags=ISNODUPS**.)

You can use the function **isuniqueid** to return a **long** integer that is unique. **C-ISAM** maintains this number and serially increments it whenever you call the function. The initial value is 1. An example of the function call follows:

```
isuniqueid(fd,&key_value);
```

The file descriptor for the **C-ISAM** file that receives the unique value is **fd**. The long integer that receives the key is **key_value**.

You must place this number in the data record in the location specified for the key. If, for example, the first four bytes of the data record, **rec**, are reserved for the key, you could use the following function call:

```
char rec[39];  
...  
stlong(key_value,rec);
```

You can use the function **issetunique** to change the starting unique identifier. If you want the value to start with 10,000, for example, you use the following call:

```
issetunique (fd,10000L);
```

If the unique identifier is already higher than 10,000, the call has no effect. The function ignores attempts to reset the unique value to less than the current value.

Audit Trail Facility

An audit trail is a file that contains a record of all changes to a single **C-ISAM** file. You should consider using it when you want to have a record of all changes to a **C-ISAM** file, yet do not need the additional facilities provided by transactions. For example, you can use an audit trail file to keep changes to a critical **C-ISAM** file and store the audit trail file on another device, such as another disk.

You can have one audit trail for each **C-ISAM** file. Even if you use the support routines for transaction management, you can use an audit trail file. If you do this, **C-ISAM** records changes in both the audit trail file and the transaction log file.

Using the Audit Trail

Use the **isaudit** function call to set or retrieve the audit trail filename, to start or stop recording changes in the C-ISAM file, or to test the status of the audit trail. The code in Figure 6-1 demonstrates the use of the audit trail.

```
#include <isam.h>
char fname[24];
.
.
.
fd = isopen("employee", ISINOUT+ISMANULOCK);
.
.
.
/* Get audit trail filename */
isaudit(fd, fname, AUDGETNAME);
.
.
.
/* Set audit trail filename */
isaudit(fd, "employee.aud", AUDSETNAME);
.
.
.
/* Test status of audit trail and
   start it if necessary */
isaudit(fd, fname, AUDINFO);
cc = strncmp(&fname[0], 0, 1); /* Compare with 0 */
if (cc==0) /* audit trail is off */
    isaudit(fd, fname, AUDSTART); /* start */
.
.
.
/* Stop audit trail */
isaudit(fd, fname, AUDSTOP);
```

Figure 6-1

Using the isaudit Function Call

The **isaudit** function calls in Figure 6-1 perform different tasks depending upon the third argument, the mode. The following list describes the action that **isaudit** takes, based upon the mode:

AUDGETNAME	retrieves the name into the string fname .
AUDSETNAME	changes the audit trail name to employee.aud .
AUDINFO	returns the status of the audit trail in the first character of the fname string. If the character is equal to 0 (ASCII null), nothing is recorded in the audit trail file. If the char-

acter is equal to 1, changes to the C-ISAM file are recorded.

AUDSTART	starts the audit trail running. Changes to the C-ISAM file are appended to the audit trail file.
AUDSTOP	stops recording C-ISAM file changes in the audit trail file.

You can use audit trails with variable-length record files just as with fixed-length record files. The audit trail file format for variable-length records contains an additional two-byte entry that indicates the actual length of the data record. See Appendix D for more information about the audit trail file format.

Audit Trail File Format

An audit trail record consists of a header and a copy of the data record. The header is shown in Figure 6-2. It is defined in **isam.h**.

```
struct audhead
{
    char au_type[2];/* audit record type aa,dd,rr,ww*/
    char au_time[4];/* audit date-time*/
    char au_procid[2];/* process id number*/
    char au_userid[2];/* user id number*/
    char au_recnum[4]; /* record number*/
    char au_reclen[2]; /* audit record length beyond header */
};
#define AUDHEADSIZE 14 /* num of bytes in audit header*/
#define VAUDHEADSIZE 16 /* VARLEN num of bytes in audit header */
```

Figure 6-2

Header for Audit Trail Records

The header variables are defined as follows:

au_type	identifies the operation on a record in the C-ISAM file. <i>aa</i> record added to the file <i>dd</i> record deleted from the file <i>rr</i> copy of the record before update (before image) <i>ww</i> copy of the record after update (after image)
au_time	is a LONGTYPE containing the time in UNIX format.
au_procid	is an INTTYPE containing the process identification number.
au_userid	is an INTTYPE containing the user identification code.
au_recnum	is a LONGTYPE that contains the number of the record that is added, deleted, or modified.

au_reclen is a LONGTYPE that contains the actual length of the variable-length-record data in bytes.

(See Chapter 3, “Data Types,” for a description of LONGTYPE and INTTYPE.)

The rest of the audit trail record is a copy of the affected data record. If the operation is a rewrite, both the before and after images are present in the audit trail file as an *rr* type followed by a *ww* type, each with the same record number.

Clustering a File

You can use **iscluster** to create a physical ordering of the data records in a C-ISAM file that corresponds to the ordering of an index on the file. This is useful if the contents of the file do not change frequently, and you need to process the file sequentially.

Ordinarily, the records in a C-ISAM file are in no particular order. Indexes are used to maintain sequential order and to find specific records within the file. To read the records in sequential order, the index is processed sequentially, and the records are retrieved by following a pointer that corresponds to the record number, or physical location, within the file. While the keys in an index node are physically adjacent, there is no guarantee that the data records are near each other in the data file.

Clustering is the ability to put records physically near each other, in a particular sequence, within a file. The **iscluster** function achieves this by building a copy of the file in the order of one of the indexes on the file.

The clustering of physical records is not permanent. Records that are added are not clustered. Over time, additions and deletions reduce the clustering of the records. If you call **iscluster**, this restores a file so that records are once again clustered.

The following function call clusters a file:

```
fd = iscluster(fd,&key);
```

The function returns a new file descriptor, **fd**, which must be used in subsequent operations on the file. The description structure that defines one of the existing indexes is **key**. This index defines the physical order for the file.

The file must be opened for exclusive use. The file remains open after the call to **iscluster**. All indexes are re-created using the new order of the records in the data file.

File Maintenance with Variable-Length Records

It is important to maintain current backups for both fixed- and variable-length data. Files that contain fixed-length data are vulnerable to data loss if the **.dat** files become corrupted. Files that contain variable-length data are vulnerable to data loss if either the **.dat** or the **.idx** files become corrupted. With fixed-length records, you can recreate an index file simply by knowing the key descriptions and some dictionary information. With variable-length records, you can recreate the index portion of the **.idx** files with the same information, but you cannot recreate the data that resides in the index files.

If data corruption occurs with a file that contains variable-length data, you can use the guidelines in the next two sections to produce a clean file.

If Data Files Are Corrupted

Restore a backup of the data and index files and then use the appropriate transaction logs or audit trails to reconstruct the **.dat** and **.idx** files.

If Index Files Are Corrupted

Do not remove the **.idx** files. Use the **bcheck** utility to clean up inaccuracies in the index portion of the **.idx** files. If the index portions of the **.idx** files are damaged, you can use the **iscluster** function to regenerate the indexes. The **iscluster** function opens the file, copies the records to a new file in the order specified by the parameters, recreates the indexes, removes the old file, and gives the new file the old name and a new file descriptor. The **iscluster** function does not use the indexes in the old file, so if only the index portions of the **.idx** files are damaged, running **iscluster** might correct the problem.

The variable-length data is not repaired by **bcheck** or by **iscluster**. If you run **iscluster** and it generates errors on reading some of the records, you need to restore the data portions of the file. You can write a program to read the records in the old file into a new file and flag any records that are damaged. Then you can use another program to remove these records and replace them. If you cannot replace the damaged records, you have to restore all of the files, both **.dat** and **.idx**, from a backup.

The following program reads the file **oldfile**, creates a new file **newfile** with the same indexes as the old file, reads each record from the old file, and puts each one in the new file. If a record is unreadable, the program puts a dummy record into the new file to retain the record order. You can use another program to delete the dummy records and take appropriate action.

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

#define SUCCESS 0
#define SIZE 32511

char    dumrec[] = "Dummy record placeholder" ;
struct  keydesc key;
struct  dictinfo info ;
int     old_fd, new_fd;

/*This program sequentially reads through an "old" variable-length
 * file and copies all of the records to a new file. If a record is
 * unreadable, a dummy record is inserted for future analysis. Both new
 * and old file names are hardcoded here but could be obtained at run
 * time.
 */

main()
{

    int          minlen,  maxlen, rr, ww;
    char         record[SIZE];

    printf("iserrno is %d\n", iserrno);
    /*open old file to obtain file descriptor and
     * find the maximum length
     */
    old_fd = isopen ("isfile1", ISVARLEN + ISINPUT + ISEXCLLOCK);
    printf("iserrno is %d\n", iserrno);
    maxlen = isreclen;
    printf("Opened old file with fd = %d and maxlen = %d\n",
           old_fd, maxlen);

    /* call isindexinfo on the primary key to obtain key description
     * in key and the minimum (fixed-length) length
     */
    isindexinfo(old_fd, &key, 1);
    minlen = isreclen;
    printf("Used isindexinfo to find minlen = %d\n", minlen);

    /* build the new file with the characteristics of the old one
     * including having the primary key but not the secondary keys.
     */
    new_fd = isbuild("newfile",maxlen,&key, ISVARLEN + ISINOUT +
                     ISEXCLLOCK);
    printf("Built newfile with new_fd = %d\n", new_fd);

    /* add the secondary indexes to the new file */
    addindex();
```

```
/*place the pointer before the first record */
isstart(old_fd, &key, 0, record, ISFIRST);

/* Read each record from the old file.
 * If the read fails, write a dummy record to the new file to
 * preserve the original record numbering. If the read is
 * successful, write the record to the new file. If a read
 * encounters the EOF or other error, or if a write encounters an
 * error, then exit.
 */

rr = SUCCESS;
ww = SUCCESS;

while (rr >= SUCCESS)
{
    rr = (isread(old_fd, record, ISNEXT + ISLOCK));
    printf("did isread and rr = %d\n", rr );
    /*isreclen has been set by isread to number of bytes in record */
    printf("iserrno = %d \n", iserrno);
    if (iserrno == EENDFILE) {printf("breaking now\n"); break;}
    if (rr < SUCCESS ) ww = (iswrite (new_fd, dumrec));
    else ww = iswrite(new_fd, record);
    if (ww < SUCCESS ) break;
}

if (iserrno == EENDFILE)
    printf ("isread encountered end of file.\n");
else if (ww < SUCCESS ) printf("iswrite failed\n");
iscleanup();
}

addindex()
{
    int cc, numkeys;

    cc = isindexinfo (old_fd, &info, 0);
    if (cc != SUCCESS) {printf ("isindexinfo error %d ", iserrno); exit(1);}
    numkeys = info.di_nkeys & 0x7fff;
    while(numkeys > 0)
    {
        isindexinfo(old_fd, &key, numkeys--);
        printf("doing isindexinfo with numkeys = %d\n");
        isaddindex(new_fd,&key);
    }
    return;
}
```

A description of the format of the index files is provided in the section “Index File Formats” on page -1 of this manual.

Summary

C-ISAM provides the following additional functions:

iserase	removes a C-ISAM file.
isrename	changes the name of a C-ISAM file.
isflush	forces output to a C-ISAM file that is opened exclusively or is on a single user machine without locking.
isuniqueid	returns a unique number that you can use in a key.
issetunique	allows you to specify the starting value for the unique number.
isaudit	allows you to set up and maintain a record of changes to your file.
iscluster	puts the records of a C-ISAM file into a specific physical order, as defined by an index.

Sample Programs Using C-ISAM Files

Overview	3
Record Definitions	3
Error Handling in C-ISAM Programs	4
Building a C-ISAM File	5
Adding Additional Indexes	6
Adding Data	7
Random Update	10
Sequential Access	14
Chaining	17
Using Transactions	22
Summary	25

Overview

This chapter introduces sample C language programs that use **C-ISAM** files. These examples are based on a very simple personnel system. The purpose of this system is to keep up-to-date information on employees and their performances.

Record Definitions

The personnel system consists of two **C-ISAM** files, the **employee** file and the **perform** file. The **employee** file holds personal information about each employee. Each record holds the employee number, name, and address. Figure 7-1 shows the file layout.

Field Name	Position	Field Type
Employee Number	0 - 3	LONGTYPE
Last Name	4 - 23	CHARTYPE
First Name	24 - 43	CHARTYPE
Address	44 - 63	CHARTYPE
City	64 - 83	CHARTYPE

Figure 7-1

Employee File Layout

The **perform** file holds information pertaining to each job performance review for an employee. The file contains one record for each performance review, job title change, or salary change. For every record in the **employee** file, at least one record must exist in the **perform** file. The **perform** file can have multiple records for the same employee. Figure 7-2 shows the layout of the **perform** file.

Field Name	Position	Field Type
Employee Number	0 - 3	LONGTYPE
Review Date	4 - 9	CHARTYPE
Job Rating	10 - 10	CHARTYPE
Salary after Review	11 - 18	DOUBLETTYPE
Title after Review	19 - 49	CHARTYPE

Figure 7-2

Performance File Layout

You must allocate one more byte for **C-ISAM** records in memory. Since a record in the **employee** file requires 84 bytes, and a record in the **perform** file requires 50 bytes, the memory storage for these records requires 85 and 51 bytes, respectively.

Error Handling in C-ISAM Programs

Every **C-ISAM** function returns a status code that your program should test.

- If the return code is zero or positive, the call results in successful execution of the function.
- If the return code is negative, however, the call is not successful. Your program can check the global variable **iserrno** to determine the reason for failure. The values returned in **iserrno** and their descriptions are in Appendix C, “Error Codes.”

The sample programs that follow do not always illustrate adequate error checking. (This omission is to shorten the length of the examples.) Programs that are used in a production environment should have much more rigorous error checking than what is presented in the sample programs.

Building a C-ISAM File

Figure 7-3 shows a C language program that creates both the **employee** and the **perform** files.

```
#include <isam.h>

#define SUCCESS 0

struct keydesc ekey, pkey;
int cc, fdemploy, fdperform;

/* This program builds the C-ISAM file systems for the
   employee and perform files */

main()
{
    /* Set up Employee Key */
    ekey.k_flags = ISNODUPS;
    ekey.k_nparts = 1;
    ekey.k_part[0].kp_start = 0;
    ekey.k_part[0].kp_leng = 4;
    ekey.k_part[0].kp_type = LONGTYPE;

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

    /* Set up Performance Key */
    pkey.k_flags = ISDUPS+DCOMPRESS;
    pkey.k_nparts = 2;
    pkey.k_part[0].kp_start = 0;
    pkey.k_part[0].kp_leng = 4;
    pkey.k_part[0].kp_type = LONGTYPE;
    pkey.k_part[1].kp_start = 4;
    pkey.k_part[1].kp_leng = 6;
    pkey.k_part[1].kp_type = CHARTYPE;
    fdperform = cc = isbuild("perform", 49, &pkey, ISINOUT + ISEXCLLOCK);
    if (cc < SUCCESS)
    {
        printf("isbuild error %d for performance file\n", iserrno);
        exit(1);
    }
    isclose(fdperform);
}
```

Figure 7-3

Creating C-ISAM Files

The primary key for the **employee** file has one part, the Employee Number. This is a long integer beginning at offset 0, the start of the record. It is four bytes long. The index does not allow duplicate keys.

The primary key for the **perform** file has two parts: Employee Number and Review Date. The first part, Employee Number, is a long integer, four bytes long, and starts at the beginning of the record, offset 0. The second part is the Review Date, which is a character field of six bytes. It starts immediately after the Employee Number, at offset 4 in the record. The file allows duplicate keys and compresses any duplicate values that are in the index.

Adding Additional Indexes

Occasionally, you need additional indexes for an application. The program in Figure 7-4 creates an index on the Last Name field in the **employee** file, and an index on the Salary field in the **perform** file.

While you add indexes, the file must be opened with an exclusive lock. You can specify exclusive file locks in the mode argument of the **isopen** call by initializing that parameter to include **ISEXCLLOCK**. **ISINOUT** specifies that the file is to be opened for both input and output. **ISEXCLLOCK**, when added to **ISINOUT**, indicates that the file is to be exclusively locked for your program. Therefore, no other program can access the file while it is open.

Both indexes allow duplicate keys. Full compression of leading duplicate characters, trailing spaces, and duplicate values is specified for the last name index.

You can drop these indexes at any time and add them again later. This is an appropriate practice when file insertions, deletions, or updates are a major activity because extra indexes slow down these operations.

```
#include <isam.h>

#define SUCCESS 0

struct keydesc lnkey, skey;
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 for employee file\n", iserrno);
        exit(1);
    }

    /* Set up Last Name Key */
    lnkey.k_flags = ISDUPS + COMPRESS;
    lnkey.k_nparts = 1;
    lnkey.k_part[0].kp_start = 4;
    lnkey.k_part[0].kp_leng = 20;
    lnkey.k_part[0].kp_type = CHARTYPE;

    cc = isaddindex(fdemploy, &lnkey);
    if (cc != SUCCESS)
    {
        printf("isaddindex error %d for employee lname key\n", iserrno);
        exit(1);
    }
}
```

```
    }
    isclose(fdemploy);

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

    /* Set up Salary Key */
    skey.k_flags = ISDUPS;
    skey.k_nparts = 1;
    skey.k_part[0].kp_start = 11;
    skey.k_part[0].kp_leng = sizeof(double);
    skey.k_part[0].kp_type = DOUBLETTYPE;

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

Figure 7-4

Adding Additional Indexes

Adding Data

Figure 7-5 shows a program that adds records to the **employee** file, and adds the first record to the **perform** file for each employee. Both files are open for output.

Both files use the ISAUTOLOCK locking option. When you add an employee record to the file, that record is locked until you either add the next record or close the file. Likewise, when you add a performance record, it is also locked

until you add another record or close the file. The program locks the records so that another program cannot access them until this program finishes with both records.

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

    getemployee();
    getperform();

    while(!finished)
    {
        addemployee();
        addperform();
        getemployee();
        getperform();
    }
    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");
}
addemployee()
{
    int cc;

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

    cc = iswrite(fdperform, perfrec);

    if (cc != SUCCESS)
    {
        printf("iswrite error %d for performance\n", iserrno);
        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);

printf("\n");
}

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

    for (i = 1; i <= num && *src != '0' && src != 0; i++)
        /* don't move carriage */
        *dest++ = *src++;
    while (i++ <= num)
        /* returns or nulls */
        *dest++ = ' ';
    /* pad remaining characters in blanks */
}
```

Figure 7-5***Adding Records to C-ISAM Files***

Random Update

The program in Figure 7-6 updates the fields in an employee record or deletes the employee record and all performance records for that employee from the file.

The program uses manual record locking. When the program reads an employee record, it locks the record. If additional records are needed, the program locks them as well. When the records are no longer needed, the locks are released.

The performance records are located using **isstart** with only the Employee Number part of the primary key. Note that you do not have to use **isstart** with each **isread** if you use the entire key to locate a record.

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

#define WHOLEKEY 0
#define SUCCESS 0
#define TRUE 1
#define FALSE 0
#define DELETE 1
#define UPDATE 2

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

struct keydesc pkey;
int fdemploy, fdperform;
int finished = FALSE;

/* This program updates the employee file.
   If the delete option is requested, all
   performance records are removed along
   with the employee record.
*/
main()
{
    int cc;
    int cmd;

    fdemploy = cc = isopen("employee", ISMANULOCK + ISINOUT);
    if (cc < SUCCESS)
    {
        printf("isopen error %d for employee file\n", iserrno);
        fatal();
    }

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

    /* Set up key description structure for isstart */
    pkey.k_flags = ISDUPS+DCOMPRESS;
    pkey.k_nparts = 2;
    pkey.k_part[0].kp_start = 0;
    pkey.k_part[0].kp_leng = 4;
    pkey.k_part[0].kp_type = LONGTYPE;
    pkey.k_part[1].kp_start = 4;
    pkey.k_part[1].kp_leng = 6;
    pkey.k_part[1].kp_type = CHARTYPE;

    cmd = getinstr();

    while(!finished)
    {
        if (cmd == DELETE)
            delrec();
```

```
        else
        {
            getemployee();
            updateemp();
        }
        cmd = getinstr();
    }

    isclose(fdemploy);
    isclose(fdperform);
}
updateemp()
{
    int cc;

    cc = isrewrite(fdemploy, emprec);
    if (cc != SUCCESS)
    {
        printf("isrewrite error %d for employee\n", iserrno);
        fatal();
    }
}

delrec()
{
    int cc;

    cc = isdelete(fdemploy, emprec);
    if (cc != SUCCESS)
    {
        printf("isdelete error %d for performance\n", iserrno);
        fatal();
    }
}

cc = isstart(fdperform, &pkey, 4, perfrec, ISEQUAL);
if (cc < SUCCESS) fatal();
cc = isread(fdperform, perfrec, ISCURR+ISLOCK);
if (cc < SUCCESS) fatal();

while (cc == SUCCESS)
{
    cc = isdelcurr(fdperform);
    if (cc < SUCCESS)
    {
        printf("isdelcurr error %d for perform\n", iserrno);
        fatal();
    }
    cc = isstart(fdperform, &pkey, 4, perfrec, ISEQUAL);
    if (cc == SUCCESS)
        cc = isread(fdperform, perfrec, ISCURR+ISLOCK);
}
if (iserrno != ENOREC && iserrno != EENDFILE)
{
    printf("isread error %d for perform\n", iserrno);
    fatal();
}

isrelease (fdemploy);
isrelease (fdperform);
}

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++));
}

getinstr()
{
int cc;
char instr[2];

tryagain:
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);
stlong(empnum, perfrec);
cc = isread(fdemploy, emprec, ISEQUAL+ISLOCK);
if (cc < SUCCESS)
{
if (iserrno == ENOREC || iserrno == EENDFILE)
{
printf("Employee No. Not Found");
goto tryagain;
}
else
{
printf("isread error %d for employee file\n", iserrno);
fatal();
}
}

showemployee();
printf("Delete? (y/n): ");
fgets(line,80,stdin);
sscanf(line,"%ls",instr);
if (strcmp(instr,"y")==0)
return (DELETE);
else
{
printf("Update? (y/n): ");
fgets(line,80,stdin);
sscanf(line,"%ls",instr);
if (strcmp(instr,"y")==0)
return (UPDATE);
}
goto tryagain;
}

getemployee ()
```

```
{
int len;

printf("Last name: ");
fgets(line, 80, stdin);
len = strlen(line);
if (len > 1)
    ststring(line, emprec+4, 20);

printf("First name: ");
fgets(line, 80, stdin);
len = strlen(line);
if (len > 1)
    ststring(line, emprec+24, 20);

printf("Address: ");
fgets(line, 80, stdin);
len = strlen(line);
if (len > 1)
    ststring(line, emprec+44, 20);

printf("City: ");
fgets(line, 80, stdin);
len = strlen(line);
if (len > 1)
    ststring(line, emprec+64, 20);

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

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

for (i = 1; i <= num && *src != '0' && src != 0; i++)
    /* don't move carriage */
    *dest++ = *src++;
while (i++ <= num)
    /* pad remaining characters in blanks */
    *dest++ = ' ';
}

fatal()
{
isclose(fdemploy);
isclose(fdperform);
exit(1);
}
```

Figure 7-6***Random Update of C-ISAM Files***

Sequential Access

The code in Figure 7-7 demonstrates how to read a file sequentially. In this program, the **employee** file is read in order of the Last Name key.

The program uses **isstart** to change from the primary index to the Last Name index and to position the file to the first key in the index. The program retrieves the first record by calling **isread** with the mode ISCURR. The current record is the record that **isstart** positions on, in this case, the record with the first key in the index. Subsequent calls to **isread** use the ISNEXT mode to read the next record in index order.

The function returns an error status in the global error variable **iserrno** with a value of EENDFILE when all records are read.

```
#include <isam.h>

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

char emprec[85];

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

/* This program sequentially reads through the employee
   file by employee number printing each record */

main()
{
    int cc;

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

    /* Set File to Retrieve using Last Name Index */
    key.k_flags = ISDUPS+COMPRESS;
    key.k_nparts = 1;
    key.k_part[0].kp_start = 4;
    key.k_part[0].kp_leng = 20;
    key.k_part[0].kp_type = CHARTYPE;
    cc = isstart(fdemploy, &key, WHOLEKEY, emprec, ISFIRST);
    if (cc != SUCCESS)
    {
        printf("isstart error %d", 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);
}
```

Figure 7-7

Sequential Processing of a C-ISAM File

Chaining

The next program uses a chaining technique to locate the performance record for an employee, by finding the highest value key for the employee in the **perform** file. This technique finds the record directly, without reading other performance records for the employee.

Figure 7-8 shows the logical order of records in the **perform** file. The primary key is a composite of the Employee Number and the Review Date fields.

Emp. No.	Review Date	Job Rating	New Salary	New 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

Figure 7-8

Sample Performance Data

The program in Figure 7-9 adds a new performance record to the **perform** file. The program calculates the new salary as a percentage raise, based upon the employee's performance. To do this, the program must find the most recent performance record.

The program finds the performance record by setting the search key to the composite of the employee number and 999999, the highest value that can be stored in the Review Date field. The **isstart** function uses this key and the ISGTEQ mode to position the file to the record immediately after the last performance record for the employee. (There should not be a review date of 999999.) The program obtains the most recent performance record by calling **isread** with ISPREV mode to return the record preceding the one found by **isstart**.

To obtain the most recent record for Employee 1 in Figure 7-8, you must perform the following steps:

1. Call **isstart** with the ISGTEQ mode and a key containing Employee 1 and Review Date 999999. The **isstart** function positions at Employee 2,

Review Date 760301, since this is the next record with a key greater than the one requested (and no key equals the one requested).

2. Call **isread** with the ISPREV mode, which reads the record with the key preceding the one found by **isstart**.

This chaining technique finds the most recent performance record for Employee 1.

Finding a record using the chaining technique is faster than finding the first performance record then finding subsequent records with the ISNEXT mode in the **isread** function call.

```
#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 dictinfo info;
struct keydesc key;
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)         4.5 %
           g (good)         7.5 %
*/

main()
{
    int cc;

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

    /* Set up key for isstart on performance file */
    key.k_flags = ISDUPS+DCOMPRESS;
    key.k_nparts = 2;
    key.k_part[0].kp_start = 0;
    key.k_part[0].kp_leng = 4;
```

```

key.k_part[0].kp_type = LONGTYPE;
key.k_part[1].kp_start = 4;
key.k_part[1].kp_leng = 6;
key.k_part[1].kp_type = CHARTYPE;

isindexinfo (fdperform,&info,0); /* check that records exist */
if (info.di_nrecords==0)
{
    printf ("No records to update\n");
    exit (1);
}
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);

    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;

bytecpy(perfrec, operfrec, 4); /* get employee id no. */
bytecpy("999999", operfrec+4, 6); /* largest possible date */

cc = isstart(fdperform, &key, WHOLEKEY, operfrec, ISGTEQ);
if (cc != SUCCESS)
{
switch(iserrno)
{
case ENOREC:
case EENDFILE:
mode = ISLAST;
break;
default:
printf("isstart error %d ", iserrno);
return(1);
}
}
else
{
mode = ISPREV;
}

cc = isread(fdperform, operfrec, mode);
if (cc != SUCCESS)
{
if (iserrno == EENDFILE)
{
printf("No performance record for employee number %ld.\n",
ldlong(perfrec));
return(1);
}
else
{
printf("isread error %d in get_old_salary\n", iserrno);
return(1);
}
}
if (cmpnbytes(perfrec, operfrec, 4))
{
printf("No performance record for employee number %ld.\n",
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 *= 1.045;
        break;
    case 'g':
        printf("good\n");
        new_salary *= 1.075;
        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);
}

ststring(src, dest, num)
/* move NUM sequential characters from SRC to DEST */
char *src;
char *dest;
int num;
{
    int i;
    for (i = 1; i <= num && *src != '0' && src != 0; i++)
        /* don't move carriage */
        *dest++ = *src++;
    /* returns or nulls */
    while (i++ <= num)
        /* pad remaining characters in blanks */
        *dest++ = ' ';
}

```

Figure 7-9

Chaining to the Last Record in a List

Using Transactions

Figure 7-10 shows a sample program that has been modified to define C-ISAM operations as transactions. (Figure 7-5 shows the non-transaction version of this program.) The program adds a record to the **employee** file and then adds a record to the **perform** file. These operations define a transaction that is repeated until the user inputs a zero for the Employee Number.

The transaction operates on two C-ISAM files. If the transaction succeeds, a record is added to each file. If the transaction fails, any change to either file is rolled back so that neither file is modified.

The functions **isopen** and **isclose** are called within the transaction to identify the files involved. For **isrollback** to reverse changes to the file, **ISTRANS** is added to the mode argument in the **isopen** function call.

Only minimal error checking is implemented in the sample program. A production program should check each function return code for an error value, especially calls to **iscommit** and **isrollback**.

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

#define SUCCESS 0
#define LOGNAME "recovery.log"

char emprec[85];
char perfrec[51];
char line[82];
long empnum;
int fdemploy, fdperform;

extern int errno;

/* 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;
    int cc1;
    int cc2;
    if (access(LOGNAME, 0) == -1)/* log file exist? */
        if ((cc = creat(LOGNAME, 0660)) == -1)
        {
            printf("Cannot create log file \"%s\", system error %d.\n", LOGNAME, errno);
            iscleanup();
            exit(1);
        }
    /* open log file */
    cc = islogopen (LOGNAME);
    if (cc < SUCCESS)
    {
        printf ("Cannot open log file, islogopen error %d\n", iserrno);
        iscleanup();
    }
}
```

```

        exit (1);
    }

while(!getemployee())
{
    /* Transaction begins after terminal input has been collected.
       Either both employee and performance record will be added
       or neither will be added. */

    /* Files must be opened and closed within the transaction */

    isbegin();    /* start of transaction */

    fdemploy = cc = isopen("employee", ISMANULOCK+ISOUTPUT+ISTRANS);
    if (cc < SUCCESS)
    { isrollback();
      break; }

    fdperform = cc = isopen("perform", ISMANULOCK+ISOUTPUT+ISTRANS);
    if (cc < SUCCESS)
    { isclose(fdemploy);
      isrollback();
      break; }

    cc1 =addemployee();
    if (cc1 == SUCCESS)
        cc2 =addperform();

    isclose(fdemploy);
    isclose(fdperform);

    if ((cc1 < SUCCESS) || (cc2 < SUCCESS)) /* transaction failed */
    {
        isrollback();
    }
    else
    {
        iscommit();    /* transaction okay */
        printf ("new employee entered\n");
    }
}

/* Finished */
islogclose();
iscleanup();
exit (0);
}

getperform()
{
double new_salary;

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 for employee\n", iserrno);
}
return (cc);
}

addperform()
{
int cc;
cc = iswrite(fdperform, perfrec);
if (cc != SUCCESS)
{
printf("iswrite error %d for performance\n", iserrno);
}
return (cc);
}

getemployee()
{
printf("Employee number (enter 0 to exit): ");
fgets(line, 80, stdin);
sscanf(line, "%ld", &empnum);

if (empnum == 0)
return(1);

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();
printf("\n\n\n");

return (0);
}

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

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

Figure 7-10

Adding Records Inside a Transaction

Summary

The chapter introduces seven example programs that show you how to perform the following tasks:

- Create C-ISAM files
- Add indexes to C-ISAM files
- Add records to files
- Retrieve, update, and delete specific records
- Sequentially process a C-ISAM file
- Find the end of a subset of records (a chain) in the C-ISAM file
- Implement transactions in an existing program

Call Formats and Descriptions

Overview 3

Functions for C-ISAM File Manipulation 6

ISADDINDEX 8
ISAUDIT 10
ISBEGIN 13
ISBUILD 15
ISCLEANUP 18
ISCLOSE 19
ISCLUSTER 20
ISCOMMIT 22
ISDELCURR 24
ISDELETE 25
ISDELINDEX 27
ISDELREC 29
ISERASE 31
ISFLUSH 32
ISINDEXINFO 33
ISLOCK 36
ISLOGCLOSE 38
ISLOGOPEN 39
ISOPEN 40
ISREAD 42
ISRECOVER 46
ISRELEASE 47
ISRENAME 48
ISREWCURR 50
ISREWREC 52
ISREWRITE 54
ISROLLBACK 56

ISSETUNIQUE 58
ISSTART 60
ISUNIQUEID 63
ISUNLOCK 64
ISWRCURR 65
ISWRITE 67

Format-Conversion and Manipulation Functions 69

LDCHAR 70
LDDBL 71
LDDBLNULL 72
LDDECIMAL 73
LDFLOAT 75
LDFLTNULL 76
LDINT 77
LDLONG 78
STCHAR 79
STDBL 80
STDBLNULL 81
STDECIMAL 82
STFLOAT 84
STFLTNULL 85
STINT 86
STLONG 87
DECCVASC 89
DECTOASC 91
DECCVINT 93
DECTOINT 94
DECCVLONG 95
DECTOLONG 97
DECCVFLT 98
DECTOFLT 99
DECCVDBL 100
DECTODBL 101
DECADD, DECSUB, DECMUL, and DECDIV 102
DECCMP 104
DECCOPY 105
DECECVT and DECFCVT 106

Summary 108

Overview

This chapter describes all the functions that are available as part of **C-ISAM**. They are divided into two major groupings:

- File manipulation functions
- Format-conversion and manipulation functions

The file manipulation functions allow you to perform the following operations:

- Create and destroy files and indexes
- Access and modify records within files
- Lock records or files
- Implement transactions
- Perform other functions associated with maintaining **C-ISAM** files

The following routines allow you to manipulate files and indexes:

isbuild	creates a C-ISAM file.
isopen	opens a C-ISAM file.
isclose	closes a C-ISAM file.
iscleanup	closes all of the C-ISAM files opened by the process.
iscluster	puts the records of a file in the physical order defined by a key.
isrename	changes the name of a C-ISAM file.
iserase	removes a C-ISAM file.
isaddindex	adds an index to a file.
isdelindex	removes an index from a file.

The following functions allow you to manipulate **C-ISAM** records:

isstart	chooses an index or record for retrieval.
isread	reads a record from a C-ISAM file.

iswrite	writes a record to a C-ISAM file.
isrewrite	updates a record in a C-ISAM file.
iswrcurr	writes a record to a C-ISAM file and makes it the current record.
isrewcurr	rewrites the current record.
isrewrec	rewrites the record identified by record number.
isdelete	deletes a C-ISAM record.
isdelcurr	deletes the current record.
isdelrec	deletes the record identified by record number.

The following functions allow you to implement locking:

islock	sets a lock on a C-ISAM file.
isunlock	removes a lock on a C-ISAM file.
isrelease	removes locks on records.

See **isread** later in this chapter for information about locking individual records within a **C-ISAM** file.

The following functions allow you to implement transactions:

isbegin	begins a transaction.
iscommit	completes a transaction.
isrollback	cancels a transaction.
islogopen	opens a transaction log file.
islogclose	closes a transaction log file.
isrecover	recovers C-ISAM files.

The following additional functions are also available with **C-ISAM**:

isaudit	maintains an audit trail.
isuniqueid	determines the last unique ID for a record.
issetunique	sets the starting unique ID.
isindexinfo	determines the characteristics of a file and its indexes.
isflush	forces output to a C-ISAM file.

The following functions convert between machine-dependent representation of numbers and the **C-ISAM** counterparts:

ldchar	copies a C-ISAM character string into a C language string.
stchar	copies a C language string into a C-ISAM format string.

ldint	converts a C-ISAM integer to a machine-dependent integer.
stint	converts a machine-dependent integer to a C-ISAM integer.
ldlong	converts a C-ISAM long integer to a machine-dependent long integer.
stlong	converts a machine-dependent long integer to a C-ISAM long integer.
ldfloat	converts a C-ISAM floating-point number to a machine-dependent floating-point number.
stfloat	converts a machine-dependent floating-point number to a C-ISAM floating-point number.
ldftnull	converts a C-ISAM floating-point number to a machine-dependent floating-point number and checks if it is null.
stftnull	converts a machine-dependent floating-point number to a C-ISAM floating-point number and checks if it is null.
lddbl	converts a C-ISAM double-precision number to a machine-dependent double-precision number.
stdbl	converts a machine-dependent double-precision number to a C-ISAM double-precision number.
lddblnull	converts a C-ISAM double-precision number to a machine-dependent double-precision number and checks if it is null.
stdblnull	converts a machine-dependent double-precision number to a C-ISAM double-precision number and checks if it is null.

The following routines allow you to manipulate the **C-ISAM** DECIMALTYPE data type.

lddecimal	loads a DECIMALTYPE number from a data record into its internal structure.
stdecimal	stores a DECIMALTYPE number in a data record.
deccvasc	converts a character string into a DECIMALTYPE number.
dectoasc	converts a DECIMALTYPE number into a character string.
deccvint	converts a machine-dependent integer into a DECIMALTYPE number.
dectoint	converts a DECIMALTYPE number into a machine-dependent integer.
deccvlong	converts a machine-dependent long integer into a DECIMALTYPE number.

dectolong	converts a DECIMALTYPE number into a machine-dependent long integer.
deccvflt	converts a machine-dependent floating-point number into a DECIMALTYPE number.
dectoflt	converts a DECIMALTYPE number into a machine-dependent floating-point number.
deccvdbl	converts a machine-dependent double precision number into a DECIMALTYPE number.
dectodbl	converts a DECIMALTYPE number into a machine-dependent double-precision number.
decadd	adds two DECIMALTYPE numbers.
decsub	subtracts two DECIMALTYPE numbers.
decmul	multiplies two DECIMALTYPE numbers.
decdiv	divides two DECIMALTYPE numbers.
deccmp	compares two DECIMALTYPE numbers.
deccopy	copies DECIMALTYPE numbers.
dececv	is the DECIMALTYPE equivalent of UNIX ecvt(3) .
decfcvt	is the DECIMALTYPE equivalent of UNIX fcvt(3) .

Functions for C-ISAM File Manipulation

This section describes the following functions in alphabetical order:

ISADDINDEX	ISLOGOPEN
ISAUDIT	ISOPEN
ISBEGIN	ISREAD
ISBUILD	ISRECOVER
ISCLEANUP	ISRELEASE
ISCLOSE	ISRENAME
ISCLUSTER	ISREWCURR
ISCOMMIT	ISREWREC
ISDELCURR	ISREWRITE
ISDELETE	ISROLLBACK
ISDELINDEX	ISSETUNIQUE
ISDELREC	ISSTART
ISERASE	ISUNIQUEID
ISFLUSH	ISUNLOCK
ISINDEXINFO	ISWRCURR
ISLOCK	ISWRITE
ISLOGCLOSE	

ISADDINDEX

Overview

Use **isaddindex** to add an index to a C-ISAM file.

Syntax

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

Explanation

<i>isfd</i>	is the file descriptor returned by isopen or isbuild .
<i>keydesc</i>	is a pointer to a key description structure.

Notes

1. The C-ISAM file must be opened for exclusive access (ISEXCLLOCK) and it must be open for both input and output (ISINOUT).
2. There is no limit to the number of indexes you can add.
3. You can only define indexes on the fixed-length portion of a record. If the character position indicated by *keydesc* exceeds the minimum record size defined for the file, **isaddindex** fails. (See **isbuild** on page 15 for more information.)
4. The maximum number of parts that you can define for an index is NPARTS.
5. The **isam.h** file contains the definition of NPARTS. (Usually, NPARTS equals 8.)
6. The maximum key size is MAXKEYSIZE. The **isam.h** file contains the definition of MAXKEYSIZE. (Usually, it is 120 bytes.)
7. The **isaddindex** call cannot be rolled back within a transaction. It can be recovered, however.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
#include <isam.h>
    struct keydesc nkey;
    .
    .
    .
    nkey.k_flags = ISDUPS;
    nkey.k_nparts = 2;
    nkey.k_part[0].kp_start = 4;
    nkey.k_part[0].kp_leng  = 10;
    nkey.k_part[0].kp_type  = CHARTYPE;
    nkey.k_part[1].kp_start = 24;
    nkey.k_part[1].kp_leng  = 1;
    nkey.k_part[1].kp_type  = CHARTYPE;
    .
    .
    .
    if ((fd=isopen("employee", ISEXCLLOCK+ISINOUT)) >= 0)
    {
        if (isaddindex(fd,&nkey) < 0)
        {
            printf ("isaddindex error %d",iserrno);
            exit (1);
        }
        .
        .
        .
    }
```

ISAUDIT

Overview

Use **isaudit** to perform operations that involve an audit trail file. You can start or stop recording changes to a **C-ISAM** file, or set the name of an audit trail file. You can also determine whether the audit trail is on or off.

Syntax

```
isaudit(isfd, filename, mode)
int isfd;
char *filename;
int mode;
```

Explanation

<i>isfd</i>	is the file descriptor returned by isopen or isbuild .										
<i>filename</i>	is a pointer to the filename or a pointer to a string to retrieve the status of the audit trail.										
<i>mode</i>	is one of the following parameters: <table><tr><td>AUDSTART</td><td>starts recording to the audit trail.</td></tr><tr><td>AUDSTOP</td><td>stops recording to the audit trail.</td></tr><tr><td>AUDSETNAME</td><td>specifies the audit trail filename.</td></tr><tr><td>AUDGETNAME</td><td>returns the audit trail filename.</td></tr><tr><td>AUDINFO</td><td>returns the status of the audit trail.</td></tr></table>	AUDSTART	starts recording to the audit trail.	AUDSTOP	stops recording to the audit trail.	AUDSETNAME	specifies the audit trail filename.	AUDGETNAME	returns the audit trail filename.	AUDINFO	returns the status of the audit trail.
AUDSTART	starts recording to the audit trail.										
AUDSTOP	stops recording to the audit trail.										
AUDSETNAME	specifies the audit trail filename.										
AUDGETNAME	returns the audit trail filename.										
AUDINFO	returns the status of the audit trail.										

Notes

1. When the mode equals AUDINFO, the function sets the first byte of the *filename* parameter (filename[0]) to zero (ASCII null) if the audit trail is off or to one if the audit trail is on.
2. When you set the audit trail filename, **C-ISAM** retains the name in the index (.idx) file.
3. When you stop the audit trail, it is not erased. Further changes to the **C-ISAM** file, however, are not recorded.
4. When you start the audit trail and the audit trail file already exists, changes to the **C-ISAM** file are appended to the audit trail file.

5. You can create a new audit trail file, either by removing the old file or by setting a new filename.
6. The audit trail filename may be any operating system filename or pathname.
7. An audit trail record contains a header and a copy of the data record. The header is defined in **isam.h** and is described in Chapter 6, “Additional Facilities.”
8. The **isaudit** call cannot be rolled back within a transaction. It can be recovered, however.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
#include <isam.h>
char fname[24];
.
.
.
fd = isopen("employee",ISINOUT+ISMANULOCK);
.
.
.
/* Get audit trail filename */
isaudit(fd,fname,AUDGETNAME);
.
.
.
/* Set audit trail filename */
isaudit(fd,"employee.aud",AUDSETNAME);
.
.
.
/* Test status of audit trail and
   start it if necessary */
isaudit(fd,fname,AUDINFO);
cc = strcmp(&fname[0],0,1); /* Compare with 0 */
if (cc==0) /* audit trail is off */
    isaudit(fd,fname,AUDSTART); /* start */
.
.
.
/* Stop audit trail */
isaudit(fd,fname,AUDSTOP);
```

ISBEGIN

Overview

Use **isbegin** to define the beginning of the transaction.

Syntax

```
isbegin()
```

Notes

1. If you are using a log file, you must call **isbegin** before you open the file for a read-only (ISINPUT) operation.
2. You must open a log file with **islogopen** with the name of the log file as the argument before you call the first **isbegin** in a program.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
isbegin();    /* start of transaction */

fdemploy = cc = isopen("employee", ISMANULOCK+ISOUTPUT+ISTRANS);
if (cc < SUCCESS)
    { isrollback();
      break; }

fdperform = cc = isopen("perform", ISMANULOCK+ISOUTPUT+ISTRANS);
if (cc < SUCCESS)
    { isclose(fdemploy);
      isrollback();
      break; }

cc1 = addemployee();
if (cc1 == SUCCESS)
    cc2 = addperform();

isclose(fdemploy);
isclose(fdperform);

if ((cc1 < SUCCESS) || (cc2 < SUCCESS)) /* transaction failed */
    {
        isrollback();
    }
else
    {
        iscommit();    /* transaction okay    */
        printf ("new employee entered\n");
    }
}
```

ISBUILD

Overview

Use **isbuild** to create a C-ISAM file.

Syntax

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

Explanation

<i>filename</i>	is the name of the file without an extension.
<i>reclen</i>	is the length of the record in bytes. If the record is to have a variable-length portion, <i>reclen</i> is the maximum length of the record. <i>reclen</i> is a number between 1 and 32,511, inclusive.
<i>keydesc</i>	is a pointer to a key description structure that defines the primary key.
<i>mode</i>	is a combination of an access mode parameter, a locking mode parameter and, optionally, a length or logging parameter. You add an access mode parameter to a lock mode parameter to specify the mode. Use one of the following access mode parameters:

ISINPUT	opens the file for input.
---------	---------------------------

ISOUTPUT	opens the file for output.
----------	----------------------------

ISINOUT	opens the file for both input and output.
---------	---

Use one of the following locking mode parameters:

ISEXCLLOCK	specifies an exclusive file lock.
------------	-----------------------------------

ISMANULOCK	specifies manual file or record locking, or no locking.
------------	---

ISAUTOLOCK	specifies automatic record locking.
------------	-------------------------------------

You can also specify the following parameters:

ISVARLEN	indicates that the record contains a variable-length portion.
ISFIXLEN	indicates that the record does not contain a variable-length portion.
ISTRANS	enables isrollback to reverse changes to C-ISAM files within a transaction.
ISNOLOG	specifies that this call and subsequent calls on this file are not logged.

Notes

1. If you do not use ISFIXLEN or ISVARLEN, the record length defaults to fixed length.
2. If you use ISVARLEN, you must give **isreclen** the minimum number of bytes in the record. If the record has a fixed-length portion, **isreclen** contains the length of the fixed-length portion. The variable-length portion of the record is at the end of the record.
3. The **isbuild** function creates two operating system files with the names *filename.dat* and *filename.idx*. (If your version of C-ISAM does not use the operating system call **fcntl()**, a third file, *filename.lok*, is also created.) These files are treated together as one logical C-ISAM file.
4. The *filename* parameter should contain a null-terminated character string that is at least four characters shorter than the longest legal operating system filename.
5. The function returns an integer file descriptor that identifies the file.
6. The file is left open with the access and locking modes that are set in the mode parameter.
7. The *keydesc* parameter specifies the structure of the primary index. You can set **k_nparts** = 0, which means that no primary key actually exists and sequential processing takes place in record number (physical) sequence.
8. You can add indexes later by using **isaddindex**.
9. If you have opened a transaction log prior to building the new file, and you want to recover the new file in case of a system failure, you must precede the **isbuild** call with an **isbegin** call.
10. The **isbuild** function cannot be rolled back.
11. If you have opened a transaction log prior to building the new file and you do not wish to recover this new file, use the ISNOLOG mode to prevent logging the **isbuild** and subsequent C-ISAM calls on the file. In this case, be sure that all future **isopen** calls for this file also specify ISNOLOG.

Return Codes

-1	Error; iserrno contains the error code
≥ 0	File descriptor

Example

```
#include <isam.h>
struct keydesc key;
.
.
.
key.k_flags = ISNODUPS;
key.k_nparts = 1;
key.k_part[0].kp_start = 0;
key.k_part[0].kp_leng = LONGSIZE;
key.k_part[0].kp_type = LONGTYPE;
.
.
.
if((fd=isbuild("employee",84,&key,ISINOUT+ISEXCLLOCK))<0)
{
    printf ("isbuild error %d",iserrno);
    exit (1);
}
/*corresponding call for a variable-length record*/
/* first set isreclen to fixed length*/
isreclen = 84
if((fd=isbuild("v_employee", 1084, &key,
               ISINOUT+ISEXCLOCK+ISVARLEN)) <0
{
    printf ("isbuild error %d",iserrno);
    exit (1);
}
```

ISCLEANUP

Overview

Use **iscleanup** to close all of the C-ISAM files opened by your program.

Syntax

```
iscleanup()
```

Note

You should make it standard practice to call **iscleanup** before exiting a C-ISAM program.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
cc = iscleanup();
```

ISCLOSE

Overview

Use **isclose** to close a C-ISAM file.

Syntax

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

Explanation

isfd is the file descriptor returned by **isopen** or **isbuild**.

Note

The program releases any locks it holds.

Caution: *It is extremely important to close C-ISAM files after processing has finished, especially on operating systems without file-locking system calls. Failure to close C-ISAM files using the **isclose** (or **iscleanup**) function leaves the files locked on systems without these system calls.*

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
cc = isclose(fd);
```

ISCLUSTER

Overview

Use **iscluster** to change the physical order of a C-ISAM file to key sequence.

Syntax

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

Explanation

<i>isfd</i>	is the file descriptor of the file that you want to modify.
<i>keydesc</i>	is a pointer to the key description structure that specifies the new physical order for the file.

Notes

1. The C-ISAM file must be opened for exclusive access.
2. The function copies the records of the file to a new file. The records in the new file are physically in the order defined by the key.
3. After successfully copying the file, the function removes the original file, renames the new file to the old filename, and leaves the file open for processing.
4. The **iscluster** function returns a new file descriptor that must be used with the new file.
5. The function re-creates all indexes.
6. Any index can be used to specify the physical order of the file.
7. Addition or deletion of records changes the physical order of records in the file, so that the effect of clustering can be lost over an extended period of time.
8. The **iscluster** call cannot be rolled back within a transaction. It can be recovered, however.
9. The C-ISAM file cannot have an audit trail at the time you use the function.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
#include <isam.h>
struct keydesc nkey;
.
.
nkey.k_flags = ISDUPS;
nkey.k_nparts = 2;
nkey.k_part[0].kp_start = 4;
nkey.k_part[0].kp_leng = 10;
nkey.k_part[0].kp_type = CHARTYPE;
nkey.k_part[1].kp_start = 24;
nkey.k_part[1].kp_leng = 1;
nkey.k_part[1].kp_type = CHARTYPE;
.
.
if ((fd=isopen("employee", ISEXCLLOCK+ISINOUT)) >= 0)
{
    if ((newfd=iscluster(fd,&nkey)) < 0)
    {
        printf ("iscluster error %d",iserrno);
        exit (1);
    }
    /* file is now open and in physical order
       by name                                     */
    fd = newfd;
.
.
```

ISCOMMIT

Overview

Use **iscommit** to end a transaction and release all locks.

Syntax

```
iscommit( )
```

Notes

1. All changes to the **C-ISAM** files within the transaction occur as the various calls are made. **iscommit** marks the transaction as completed in the log file so that the changes are rolled forward when the file must be recovered.
2. The function releases any locks held by the transaction.
3. Calling **iscommit** without a preceding **isbegin** causes an error.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
isbegin();    /* start of transaction */

fdemploy = cc = isopen("employee", ISMANULOCK+ISOUTPUT+ISTRANS);
if (cc < SUCCESS)
    { isrollback();
      break; }

fdperform = cc = isopen("perform", ISMANULOCK+ISOUTPUT+ISTRANS);
if (cc < SUCCESS)
    { isclose(fdemploy);
      isrollback();
      break; }

cc1 = addemployee();
if (cc1 == SUCCESS)
    cc2 = addperform();

isclose(fdemploy);
isclose(fdperform);

if ((cc1 < SUCCESS) || (cc2 < SUCCESS)) /* transaction failed */
    {
        isrollback();
    }
else
    {
        iscommit();    /* transaction okay    */
        printf ("new employee entered\n");
    }
}
```

ISDELCURR

Overview

Use **isdelcurr** to delete the current record from the **C-ISAM** file.

Syntax

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

Explanation

isfd is the file descriptor returned by **isopen** or **isbuild**.

Notes

1. The function removes the key from each existing index.
2. This call is useful when, for example, you want to delete the most recent record read with **isread**.
3. The **isrecnum** global variable is set to the record number of the deleted record.
4. The current record is undefined since it points to space that contained the deleted record.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
cc = isdelcurr(fd);
```

ISDELETE

Overview

Use **isdelete** to delete a record using the primary key.

Syntax

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

Explanation

<i>isfd</i>	is the file descriptor returned by isopen or isbuild .
<i>record</i>	contains a key value in the position defined for the primary key.

Notes

1. The **isdelete** function uses a unique primary index to find the record that you want to delete. You must have defined a unique primary index when you created the file.
2. You cannot use this function with files that are created with **INFORMIX-4GL**, **INFORMIX-SQL**, or an embedded language, such as **INFORMIX-ESQL/C** since the **C-ISAM** files that constitute SQL databases do not contain primary indexes. Use **isdelcurr** instead.
3. If the primary index is not unique, use **isread** to find the record and **isdelcurr** to delete it.
4. The function removes the key of the record from each index.
5. The **isdelete** function does not change the current record.
6. The **isdelete** function sets **isrecnum** to the record number of the deleted record.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
char emprec[85];
int fd;
int cc;
.
.
.
/* Set up key to delete Employee No. 101 */
stlong(101L,&emprec[0]);

cc = isdelete(fd,emprec);
```

ISDELINDEX

Overview

Use **isdelindex** to remove an existing index.

Syntax

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

Explanation

<i>isfd</i>	is the file descriptor returned by isopen or isbuild .
<i>keydesc</i>	is a pointer to a key description structure.

Notes

1. You can use **isdelindex** to delete any index except the primary index.
2. You must open the C-ISAM file for exclusive access.
3. The key description structure identifies the index you want to delete.
4. The **isdelindex** call cannot be rolled back within a transaction. It can be recovered, however.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
#include <isam.h>
struct keydesc nkey;
.
.
.
nkey.k_flags = ISDUPS;
nkey.k_nparts = 2;
nkey.k_part[0].kp_start = 4;
nkey.k_part[0].kp_leng = 10;
nkey.k_part[0].kp_type = CHARTYPE;
nkey.k_part[1].kp_start = 24;
nkey.k_part[1].kp_leng = 1;
nkey.k_part[1].kp_type = CHARTYPE;
.
.
.
if ((fd=isopen("employee", ISEXCLLOCK+ISINOUT)) >= 0)
{
    if (isdelindex(fd,&nkey) < 0)
    {
        printf ("isdelindex error %d",iserrno);
        exit (1);
    }
}
```

ISDELREC

Overview

Use **isdelrec** to delete a record using the record number.

Syntax

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

Explanation

<i>isfd</i>	is the file descriptor returned by isopen or isbuild .
<i>recnum</i>	is the record number of the data file record.

Notes

1. The **isdelrec** function uses the record number to find the record you want to delete.
2. Use this call if you know the record number of the record. You know the record number, for example, if you save the value of **isrecnum** when you find the record.
3. The function call removes the key from each index.
4. The **isdelrec** function does not change the current record position.
5. The **isrecnum** global variable is set to the record number of the deleted record.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

Use the following syntax to delete record 100:

```
cc = isdelrec(fd,100L);
```

ISERASE

Overview

Use **iserase** to remove the operating system files comprising the C-ISAM file.

Syntax

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

Explanation

filename is the C-ISAM file you want to delete.

Notes

1. Do not use a filename extension with the *filename* argument.
2. The function deletes *filename.idx* and *filename.dat* (and *filename.lok* and the audit trail file, if they exist).
3. You must close the file that you want to delete before you call **iserase**.
4. The **iserase** call cannot be rolled back within a transaction. It can be recovered, however.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
iserase ("personnel");
```

ISFLUSH

Overview

Use **isflush** to immediately flush any buffered index pages to the operating system.

Syntax

```
isflush(isfd)
int isfd;
```

Explanation

isfd is the file descriptor returned by **isopen** or **isbuild**.

Notes

1. Ordinarily, C-ISAM flushes data to the operating system after each function call.
2. Data is not immediately written to the operating system on single-user systems where the operating system does not provide a locking facility, nor for C-ISAM files opened for exclusive access. Periodic calls to **isflush** protect you against substantial loss of data during a system crash.
3. Use **isflush** only on files that have been opened with ISOUTPUT or ISINOUT.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
isflush(fd);
```

ISINDEXINFO

Overview

Use **isindexinfo** to determine information about the structure and indexes of a C-ISAM file.

Syntax

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

Explanation

<i>isfd</i>	is the file descriptor returned by isopen or isbuild .
<i>buffer</i>	is a pointer to a structure.
<i>number</i>	is either an index number or zero.

Notes

1. To retrieve information about a specific index, you must provide the index number as the *number* argument. You use a pointer to a **keydesc** structure to receive the information.
2. You get general information, including the number of indexes, index node size, and data record size, when you call **isindexinfo** with *number* set to zero and with a *buffer* of structure type **dictinfo**.
3. Indexes have numbers, starting with 1. The primary index is always index 1.
4. As indexes are added and deleted, the number of a particular index can change. To ensure review of all indexes, loop over the number of indexes indicated in **dictinfo**.

5. If the file has variable-length records, **isindexinfo** stores the minimum record length (that is, the length of the fixed-length portion) in the global variable **isreclen**.

In addition, if the file has variable-length records, the **di_nkeys** and **di_recsz** variables pointed to by *buffer* contain information specific to the variable-length records as follows:

di_nkeys If the file supports variable-length records, the significant bit is set. The remaining bits indicate the number of indexes defined for the file, as it does with fixed-length records.

di_recsz This field contains the maximum record size in bytes.

See “Determining Index Structures” on page 2-11 for more information on the **dictinfo** structure.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Examples

To get general information about the C-ISAM file, call **isindexinfo** as follows:

```
#include <isam.h>
struct dictinfo info;
.
.
.
fd = isopen ("employee", ISINPUT+ISEXCLLOCK);
isindexinfo (fd, &info, 0);
printf ("\nRecord size in bytes=%d", info.di_recsz);
printf ("\nNumber of records in the file=%d",
                                info.di_nrecords);

isclose (fd);
exit (0);
```

To get information about each index, call **isindexinfo** as follows:

```
#include <isam.h>
struct dictinfo info;
struct keydesc kdesc;
.
.
.
/* get number of keys */
isindexinfo (fd,&info,0);
/* Mask off significant bit to leave number of
 * indexes defined for the file */

numkeys = info.di_nkeys & 0x7fff;
while (numkeys > 0)
{
    /* get structure and decrement index number */
    isindexinfo (fd,&kdesc,numkeys--);
    .
    .
    .
}
```

ISLOCK

Overview

Use **islock** to lock the entire C-ISAM file.

Syntax

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

Explanation

isfd is the file descriptor of the file you want to lock that is returned by **isopen** or **isbuild** of the file you want to lock.

Notes

1. You must open the file with the ISMANULOCK mode.
2. You can release the lock with **isunlock**.
3. Other programs can read records but they cannot update records.
4. Other programs cannot lock the same file until you call **isunlock**.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
fd = isopen("employee", ISMANULOCK+ISINOUT);

/* file is unlocked until explicitly locked with islock */
.
.
.
islock(fd); /* file is locked at this point */

/* other programs can read employee records but all
   other operations on the file are prevented */
.
.
.
isunlock(fd); /* file is unlocked here */
```

ISLOGCLOSE

Overview

Use **islogclose** to close the transaction log file.

Syntax

```
islogclose()
```

Note

Subsequent **C-ISAM** function calls do not record anything in the transaction log file.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
islogclose();
```

ISLOGOPEN

Overview

Use **islogopen** to open the transaction log file. All subsequent C-ISAM calls record appropriate information in this file unless they contain parameters specifying not to.

Syntax

```
islogopen(logname) char *logname;
```

Explanation

logname is a pointer to the filename string.

Note

The log file must already exist.

Caution: *If the log file does not exist, C-ISAM calls still work. No log file records are saved, however, and recovery is impossible.*

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
islogopen("recovery.log");
```

ISOPEN

Overview

Use **isopen** to open a C-ISAM file for processing.

Syntax

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

Explanation

<i>filename</i>	is the name of the file.
<i>mode</i>	is a combination of an access mode parameter and a locking mode parameter and, optionally, a transaction- related parameter. You add an access mode parameter to a lock mode parameter to specify the mode. Use one of the following access mode parameters:
ISINPUT	opens the file for input (read only).
ISOUTPUT	opens the file for output (write only).
ISINOUT	opens the file for both input and output.
Use one of the following locking mode parameters:	
ISEXCLLOCK	specifies an exclusive file lock.
ISMANULOCK	specifies manual file or record locking, or no locking.
ISAUTOLOCK	specifies automatic record locking.
You can also specify the following parameters:	
ISVARLEN	indicates that each record contains a variable-length portion. If you built the file with ISVARLEN, you must open it with ISVARLEN.
ISFIXLEN	indicates that the record does not contain a variable-length portion.
ISTRANS	enables isrollback to reverse changes to C-ISAM files within a transaction.

ISNOLOG specifies that this call and subsequent calls on this file are not logged.

Caution! *If at any time, changes are made to a C-ISAM file but not logged in the log file, recovery is rendered impossible. Either all transactions or no transactions must be logged for any given C-ISAM file. If you want changes to be logged, call **isbegin** before you call **isopen**.*

Notes

1. The function returns the file descriptor that you must use in subsequent operations on the C-ISAM file.
2. When you open the file, access is by way of the primary index. If you need another ordering, use **isstart** to select another index or to select record number ordering.
3. The *filename* parameter must contain a null-terminated string without an extension, which is the filename of the C-ISAM file to be processed.
4. If you use the ISVARLEN parameter with the function call, the global integer **isreclen** is set to the maximum record length for the file.
5. If you do not specify ISVARLEN or ISFIXLEN, ISFIXLEN is assumed. If you attempt to open a variable-length record file without ISVARLEN, an error is returned.

Return Codes

-1	Error; iserrno contains the error code
>=0	File descriptor

Example

```
fd_per = isopen("perform", ISINOUT+ISMANULOCK+ISTRANS);  
fd_per = isopen("employee", ISINOUT+ISEXCLOCK);  
fd_per = isopen("v_employee", ISVARLEN+ISINOUT+ISEXCLOCK);
```

ISREAD

Overview

Use **isread** to read records sequentially or randomly, as indicated by the *mode* parameter.

Syntax

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

Explanation

<i>isfd</i>	is the file descriptor returned by isopen or isbuild .																
<i>record</i>	is a pointer to a string that contains the search value and receives the record.																
<i>mode</i>	is one of the following parameters: <table><tr><td>ISCURR</td><td>reads the current record.</td></tr><tr><td>ISFIRST</td><td>reads the first record.</td></tr><tr><td>ISLAST</td><td>reads the last record.</td></tr><tr><td>ISNEXT</td><td>reads the next record.</td></tr><tr><td>ISPREV</td><td>reads the previous record.</td></tr><tr><td>ISEQUAL</td><td>reads the record equal to the search value.</td></tr><tr><td>ISGREAT</td><td>reads the first record that is greater than the search value.</td></tr><tr><td>ISGTEQ</td><td>reads the first record that is greater than or equal to the search value.</td></tr></table>	ISCURR	reads the current record.	ISFIRST	reads the first record.	ISLAST	reads the last record.	ISNEXT	reads the next record.	ISPREV	reads the previous record.	ISEQUAL	reads the record equal to the search value.	ISGREAT	reads the first record that is greater than the search value.	ISGTEQ	reads the first record that is greater than or equal to the search value.
ISCURR	reads the current record.																
ISFIRST	reads the first record.																
ISLAST	reads the last record.																
ISNEXT	reads the next record.																
ISPREV	reads the previous record.																
ISEQUAL	reads the record equal to the search value.																
ISGREAT	reads the first record that is greater than the search value.																
ISGTEQ	reads the first record that is greater than or equal to the search value.																

Optionally, you can add one or more of the following locking options to the search mode:

ISLOCK	locks the record.
ISSKIPLOCK	sets the record pointer and isrecnum to the locked record; if isread encounters a locked record, you can use another isread with the ISNEXT option to skip the locked record.

ISWAIT	causes the process to wait for a locked record to become free.
ISLCKW	is the same as ISLOCK+ISWAIT.

Notes

1. Place the search value in the *record* in the appropriate position for the key.
2. If the search is successful, **isread** fills the remainder of the *record* with the returned record.
3. The record becomes the current record for the file.
4. The **isread** function sets the global variable **isrecnum** to the record number of the record it reads. If the file has variable-length records, **isread** sets the global variable **isreclen** to the number of bytes returned in the record buffer. (The contents of the buffer beyond the value of **isreclen** are undefined.)
5. You can use **isread** to read specific records using the record number. Call **isstart** with a **keydesc** structure that contains **k_nparts** = 0, so that retrieval is in physical order. Subsequent calls to **isread** with *mode* set to ISEQUAL cause the function to look in **isrecnum** and read the record number.
6. Add ISLOCK to one of the retrieval mode parameters to lock a record. The ISMANULOCK locking mode must be set when the file is opened. The record remains locked until you unlock it with **isrelease**, **iscommit**, or **isrollback**.
7. If you are using only part of a composite index, do not use the ISEQUAL mode. The **isread** function in the ISEQUAL mode does not find exact matches for a partial search value. You can use **isstart** with ISEQUAL and **isread** with ISCURRE to find the first occurrence of the record.
8. If you use **isread** with ISCURRE or ISNEXT after you have added a record with **iswrite**, **isread** returns the record that you just added.
9. If you use **isread** with ISCURRE or ISNEXT after you have made an **isstart** call, **isread** returns the starting record in either case.
10. If your **isread** call with the ISCURRE, ISNEXT, or ISPREV option encounters a locked record, the contents of **isrecnum** do not change from the time of the last valid **isread** call. In addition, the current record is still the last valid record as returned by the previous **isread**.

If you want to skip locked records, use the ISSKIPLOCK option. With ISSKIPLOCK set, if **isread** encounters a locked record, **isrecnum** contains the record number of the locked record and the locked record is made the

current record. Issuing another **isread**(ISNEXT) call will skip to the next record.

11. If your **isread** call with the ISFIRST, ISLAST, ISEQUAL, ISGREAT, or ISGTEQ option encounters a locked record, **isrecnum** is set to the record number of the locked record.
12. You can use ISWAIT and ISLCKW only if your version of C-ISAM uses the **fcntl()** call for record locking.
13. If **isread** encounters a locked record without ISSKIPLOCK, one of the following actions occurs:
 - If the ISWAIT flag is used, the process waits for the lock.
 - If ISWAIT is not used, the process returns value 107 (ELOCKED) in **iserrno**.
14. Once an **isread** call returns EENDFILE, the current record position is undefined. If you make another **isread**(ISNEXT) call, the ENOCURR code is returned.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Examples

The following code finds the record with the key value *100* in the primary key field:

```
/* put 100 into the correct position in the record */
stlong(100L,&emprec[0]);

if (isread(fd,emprec,ISEQUAL)<0)
{
    if (iserrno == ENOREC) printf ("record not found");
    .
    .
    .
}
```

The following code reads record 500:

```
pkey.k_nparts = 0; /* choose physical order */
isrecnum = 500L;  /* set record number to first
                  record to be processed */

cc = isstart(fd,&pkey,0,emprec,ISEQUAL);
if (cc >= 0)
    if (isread(fd,emprec,ISEQUAL)<0)
    {
        printf ("read error %d",iserrno);
        .
        .
        .
    }
```

ISRECOVER

Overview

Use **isrecover** along with the log file to redo all committed transactions in a copy of the **C-ISAM** file.

Syntax

```
isrecover()
```

Notes

1. To use **isrecover**, you must have a backup copy of the **C-ISAM** files and a log file that you started immediately after the backup.
2. The log file must already be open by a call to **islogopen**.
3. No one should use the **C-ISAM** files before the function finishes executing.
4. If any filenames are referenced by relative pathnames, it is important to run the program that calls **isrecover** from the same directory location as all other programs that access these files.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
isrecover();
```

ISRELEASE

Overview

Use **isrelease** to unlock records that are locked by calls to **isread** with the ISLOCK option.

Syntax

```
isrelease(isfd)
int isfd;
```

Explanation

isfd is the file descriptor returned by **isopen** or **isbuild**.

Notes

1. The **isrelease** function unlocks all records in the C-ISAM file that your program locked.
2. A call to **isrelease** during a transaction only releases unmodified records.
3. If you have used an **isstart** call with the ISKEEPLOCK option, you must use **isrelease** to unlock the record.
4. Locks held within a transaction are not released until **iscommit** or **isrollback** is called.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
isrelease(fd);
```

ISRENAME

Overview

Use **isrename** to change the name of a **C-ISAM** file.

Syntax

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

Explanation

<i>oldname</i>	is the file you want to rename.
<i>newname</i>	is the name of the new file.

Notes

1. Do not specify a filename extension for the **C-ISAM** file.
2. The **isrename** function renames the **.dat**, **.idx**, and **.lok** files.
3. The function does not change the name of audit trail or transaction log files since their names are not logically tied to the **C-ISAM** filename.
4. The **isrename** function uses the *newname* parameter exclusively to determine placement in the file system of the newly named file. Be careful to correctly specify this position by using an explicit pathname or relative pathname. If you use a relative pathname, keep in mind the current working directory of the program.
5. The **isrename** call cannot be rolled back within a transaction. It can be recovered, however.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
isrename ("employee","personnel");
```

ISREWCURR

Overview

Use **isrewcurr** to modify or update fields in the current record.

Syntax

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

Explanation

<i>isfd</i>	is the file descriptor returned by isopen or isbuild .
<i>record</i>	contains the complete record including updated fields.

Notes

1. If you are using **isrewcurr** on a variable-length record, you must first set the global variable **isrecLEN** to the actual length of the data in the record parameter.
2. If you change a key field, C-ISAM updates the index entry.
3. You can change the value of the primary key field.
4. The function sets **isrecnum** to the record number of the current record. The current record position does not change, that is, **isrecnum** contains the record number of the record just written.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Examples

```
cc = isrewcurr(fd,emprec);
```

If you are using a variable-length record, you might use the following call. If the minimum length of the record is 84 bytes, the maximum length is 1084 bytes, and the data being passed to the function is 923 bytes long, set **isreclen** to 923 before calling **isrewcurr**.

```
isreclen = 923;  
cc = isrewcurr(fd, emprec);
```

ISREWREC

Overview

Use **isrewrec** to update a record identified by its record number.

Syntax

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

Explanation

<i>isfd</i>	is the file descriptor returned by isopen or isbuild .
<i>recnum</i>	is the record number.
<i>record</i>	contains the complete record including updated fields.

Notes

1. If you are using **isrewrec** on a variable-length record, you must first set the global variable **isreclen** to the actual length of the data in the record parameter.
2. If you change a key field, C-ISAM updates the index entry.
3. You can change the value of the primary key field.
4. The function sets **isrecnum** to the record number of the record.
5. The current record position does not change.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

The following call rewrites record 404:

```
cc = isrewrec(fd,404L,emprec);
```

ISREWRITE

Overview

Use **isrewrite** to rewrite the nonprimary key fields of a record in a C-ISAM file.

Syntax

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

Explanation

<i>isfd</i>	is the file descriptor returned by isopen or isbuild .
<i>record</i>	contains the complete record including the primary key and the updated fields.

Notes

1. If you are using **isrewrite** on a variable-length record, you must first set the global variable **isreclen** to the actual length of the data in the record parameter.
2. The primary key in the *record* identifies the record you want to rewrite.
3. The primary index must be unique.
4. You cannot change the value of the primary-key field.
5. You cannot use this function with files that are created with **INFORMIX-4GL**, **INFORMIX-SQL**, or an embedded language such as **INFORMIX-ESQL/C** because the C-ISAM files that comprise SQL databases do not contain primary indexes. Use **isrewcurr** or **isrewrec** instead.
6. If you change a key field in a nonprimary index, the function updates the index.
7. C-ISAM does not change the current record position.
8. The function sets **isrecnum** to the record number of the record.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
stchar("San Francisco",&emprec[64],20);/* Item to be changed */  
cc = isrewrite(fd,emprec);    /* Primary key cannot change */
```

ISROLLBACK

Overview

Use **isrollback** to cancel the effect of C-ISAM calls since the last call to **isbegin**.

Syntax

```
isrollback()
```

Notes

1. The **isrollback** function returns any modified records to their original unmodified state.
2. You must include the ISTRANS parameter as part of the mode in the **isopen** call to effect the reversal of modified records.
3. You cannot roll back the following calls: **isbuild**, **isaddindex**, **iscluster**, **isdelindex**, **isaudit**, **issetunique**, **isuniqueid**, **isrename**, or **iserase**.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
isbegin();    /* start of transaction */

fdemploy = cc = isopen("employee", ISMANULOCK+ISOUTPUT+ISTRANS);
if (cc < SUCCESS)
{ isrollback();
  break; }

fdperform = cc = isopen("perform", ISMANULOCK+ISOUTPUT+ISTRANS);
if (cc < SUCCESS)
{ isclose(fdemploy);
  isrollback();
  break; }

cc1 = addemployee();
if (cc1 == SUCCESS)
    cc2 = addperform();

isclose(fdemploy);
isclose(fdperform);

if ((cc1 < SUCCESS) || (cc2 < SUCCESS)) /* transaction failed */
{
    isrollback();
}
else
{
    iscommit();      /* transaction okay    */
    printf ("new employee entered\n");
}
```

ISSETUNIQUE

Overview

Use **issetunique** to set the value of the internally stored unique identifier.

Syntax

```
issetunique(isfd, uniqueid)
int isfd;
long uniqueid;
```

Explanation

<i>isfd</i>	is the file descriptor returned by isopen or isbuild .
<i>uniqueid</i>	is a long integer specifying the new unique identifier.

Notes

1. A *uniqueid* is maintained for each C-ISAM file.
2. You can use this function if you need a unique primary key value for a record, and no other part of the record is suitable.
3. If the value of the *uniqueid* is less than the current unique identifier, the function does not change the value.
4. You can use **isuniqueid** to determine the greatest *uniqueid*.
5. The **issetunique** call cannot be rolled back within a transaction. It can be recovered, however.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

The following call sets the unique identifier to 10,000, if the identifier is less than 10,000:

```
issetunique (fd,10000L);
```

ISSTART

Overview

Use **isstart** to select the index and the starting point in the index for subsequent calls to **isread**.

Syntax

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

Explanation

<i>isfd</i>	is the file descriptor returned by isopen or isbuild .	
<i>keydesc</i>	is a pointer to a key description structure.	
<i>length</i>	specifies the part of the key that is to be considered significant when locating the starting record.	
<i>record</i>	specifies the key search value.	
<i>mode</i>	is one of the following parameters:	
	ISFIRST	finds the first record by positioning the starting point just before the first record.
	ISLAST	finds the last record by positioning the starting point just before the last record.
	ISEQUAL	finds the record equal to the search value.
	ISGREAT	finds the first record greater than the search value.
	ISGTEQ	finds the first record greater than or equal to the search value.
	ISKEEPLOCK	causes isstart to keep locks held on any record in automatic locking mode.

Notes

1. **isstart** selects the index that you want to use for subsequent calls to **isread**, but does not read a record in the C-ISAM file.
2. The key description structure that defines the index you want to use is *keydesc*.
3. If you choose the ISEQUAL, ISGREAT, or ISGTEQ *mode*, place the search key value in the *record* in the appropriate position for the key. Alternatively, you can use these modes with a record number by setting **isrecnum**.
4. If you want to locate a record using the entire key, set the *length* to either zero or the length of the entire key.
5. If you wish to locate a record using only part of the key, place in *length* the number of bytes that you want **isstart** to use when it compares the search key with the index entries. Subsequent calls to **isread** using the ISEQUAL, ISGREAT, or ISGTEQ use the entire key, however.
6. If the *mode* is ISFIRST or ISLAST, **isstart** ignores the contents of *record* and *length*.
7. If the function cannot find the search value, it returns a value of -1. The **isstart** call, however, still sets the index to the one defined by *keydesc*.
8. You can use **isstart** to specify retrieval by record number when you use a key description structure with **k_nparts**= 0.
9. If you use **isstart** with **k_nparts**= 0 and the ISFIRST option, and then issue an **isread**(ISCURR) call, C-ISAM looks for the first record (**isrecnum** = 1). If the first record is no longer available, C-ISAM returns the first valid record.
10. The function sets **isrecnum** to the starting record number.
11. The contents of the current record do not change.
12. Use **isstart** only when you want to change an index or use part of a key as the search criterion. You do not need to use **isstart** before each **isread** call.
13. Without the ISKEEPLOCK option, an **isstart** call will unlock any record locked in automatic mode.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Examples

The following call uses the key description structure **key** to select the index. **C-ISAM** ignores the contents of **len** and **emprec**, because the mode specifies the first index entry.

```
cc = isstart(fd,&key,len,emprec,ISFIRST);
```

The following example shows you how to start the index in record order, beginning with record number 500:

```
pkey.k_nparts = 0; /* choose physical order */
isrecnum = 500L; /* set record number to first
                  record to be processed */

cc = isstart(fd,&pkey,0,emprec,ISEQUAL);
```

ISUNIQUEID

Overview

Use **isuniqueid** to return a **long** integer that is guaranteed to be unique for the C-ISAM file.

Syntax

```
isuniqueid(isfd, uniqueid)
int isfd;
long *uniqueid;
```

Explanation

<i>isfd</i>	is the file descriptor returned by isopen or isbuild .
<i>uniqueid</i>	is a pointer to the long integer that receives the unique identifier.

Notes

1. The value returned by the function is serially incremented with each call.
2. This function is useful when you need a unique primary key, and the data record does not contain any fields of reasonable size that are guaranteed to be unique.
3. You must place *uniqueid* in the data record.
4. The **isuniqueid** call cannot be rolled back within a transaction. It can be recovered, however.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
isuniqueid(fd,&key_value);
```

ISUNLOCK

Overview

Use **isunlock** to remove a lock on a file.

Syntax

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

Explanation

isfd is the file descriptor returned by **isopen** or **isbuild**.

Note :The **isunlock** function removes the file lock set by **islock**.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
islock(fd); /* file is locked at this point */

/* other programs can read employee records but all
   other operations on the file are prevented */
.
.
.
isunlock(fd); /* file is unlocked here */
```

ISWRCURR

Overview

Use **iswrcurr** to write a record and make it the current record.

Syntax

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

Explanation

<i>isfd</i>	is the file descriptor returned by isopen or isbuild .
<i>record</i>	is a pointer to the record you want to write.

Notes

1. If you are using **iswrcurr** on a variable-length record, you must first set the global variable **isreclen** to the actual length of the data in the record parameter.
2. Each index receives a key for the record.
3. The function sets **isrecnum** to this record.
4. The record becomes the current record.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
stlong(101L,&emprec[0]);
.
.
.
if (iswrcurr(fd,emprec) < 0)
{
    printf ("iswrcurr error %d",iserrno);
    .
    .
    .
}
else /* this record is the current record */
{
    .
    .
    .
}
```

ISWRITE

Overview

Use **iswrite** to write a record to a C-ISAM file.

Syntax

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

Explanation

<i>isfd</i>	is the file descriptor returned by isopen or isbuild .
<i>record</i>	is a pointer to the record you want to write.

Notes

1. If you are using **iswrite** on a variable-length record, you must first set the global variable **isreclen** to the actual length of the data in the record parameter.
2. Each index receives a key for the record.
3. The current record does not change.
4. The function sets **isrecnum** to the record number of this record.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
stlong(100L,&emprec[0]);  
.  
.  
.  
if (iswrite(fd,emprec) < 0)  
{  
    printf ("iswrite error %d",iserrno);  
    .  
    .  
    .  
}  
else /* current record position not changed */  
{  
    .  
    .  
    .  
}
```

Format-Conversion and Manipulation Functions

This section is divided into two parts. The first part defines the functions that convert between machine-dependent C language data types and the **C-ISAM** equivalents. The second part defines functions that you can use to manipulate the **C-ISAM** DECIMALTYPE data type.

Format-Conversion Functions

The functions that allow you to convert between machine-dependent C language data types and the **C-ISAM** equivalents are defined on the following pages. They are presented in alphabetical order.

LDCHAR	STCHAR
LDDBL	STDBL
LDDBLNULL	STDBLNULL
LDDECIMAL	STDECIMAL
LDFLOAT	STFLOAT
LDFLTNULL	STFLTNULL
LDINT	STINT
LDLONG	STLONG

LDCHAR

Overview

Use **ldchar** to convert a character string in a **C-ISAM** data record to a null-terminated string.

Syntax

```
ldchar(fstr,length,cstr);
char *fstr;
int length;
char *cstr;
```

Explanation

<i>fstr</i>	is a pointer to the starting byte of a C-ISAM character string.
<i>length</i>	is the length of the C-ISAM character string.
<i>cstr</i>	is the destination string in memory.

Notes

1. **C-ISAM** does not terminate a character string with a null character. Instead, it pads the string with trailing spaces.
2. The function removes trailing spaces and places a null byte after the last non-blank character.

Example

```
char rec[39]; /* C-ISAM data file record */
char cname[21]; /* Null-terminated string
                without trailing blanks */
.
.
.
ldchar(&rec[4],20,cname);
```

LDDBL

Overview

Use **lddbl** to return a machine-dependent, double-precision floating-point number from a **C-ISAM** format DOUBLETTYPE.

Syntax

```
double lddbl(p)
char *p;
```

Explanation

p is a pointer to the starting byte of a **C-ISAM** DOUBLETTYPE number.

Note: A **C-ISAM DOUBLETTYPE** has the same format as the **C *double***, except that a **C-ISAM** number may not be aligned on a word boundary.

Example

```
char rec[39]; /* C-ISAM Data File Record */
/* Retrieve Trans. Amt.
   and Acct. Balance from Record */
tramt = ldfloat(&rec[26]);
acctbal = lddbl(&rec[30]);
```

LDDBLNULL

Overview

Use **lddblnull** to return a machine-dependent double-precision floating-point number from a **C-ISAM** format DOUBLETYPED and simultaneously test if the value is `null`.

Syntax

```
double lddblnull(p, nullflag)
char *p;
short *nullflag;
```

Explanation

<i>p</i>	is a pointer to the starting byte of a C-ISAM DOUBLETYPED number.
<i>nullflag</i>	is a pointer to the null code.

Notes

1. A **C-ISAM** DOUBLETYPED has the same format as the **C double**, except that a **C-ISAM** number may not be aligned on a word boundary.
2. If the value of the DOUBLETYPED number is `null`, then **lddblnull** sets **nullflag* to 1, and returns a 0.
3. If the value of the DOUBLETYPED number is not `null`, then **lddblnull** sets **nullflag* to 0, and returns the value.

Example

```
char rec[39]; /* C-ISAM Data File Record */
/* Retrieve Trans. Amt.
   and Acct. Balance from Record */
tramt = ldfltnull(&rec[26],nlflg);
acctbal = lddblnull(&rec[30],nlflg2);
```

LDDECIMAL

Overview

Use **lddecimal** to return a DECIMALTYPE number in a **dec_t** structure from a C-ISAM data record.

Syntax

```
lddecimal (cp,len,decp)
char *cp;
int len;
dec_t *decp;
```

Explanation

<i>cp</i>	is a pointer to the position in the data record where the decimal data starts.
<i>len</i>	is the length of the decimal data in the data record.
<i>decp</i>	is the dec_t structure that receives the decimal data.

Notes

1. DECIMALTYPE data is stored in a packed format within the **C-ISAM** file.
2. DECIMALTYPE data must be transferred into a **dec_t** structure before the program can manipulate it.
3. The length parameter *len* specifies the length of the packed data and is between 2 and 17 bytes, inclusive.
4. The packed length is the sum of the following three items: the number of significant digits to the left of the decimal point, divided by two and rounded up; the number of significant digits to the right of the decimal point, divided by two and rounded up; plus one byte. (See the section “Sizing DECIMALTYPE Data” in Chapter 3 for more information.)

Return Codes

-1201	Underflow error
-1200	Overflow error
0	Successful

Example

```
#include <decimal.h>
dec_t tramt;
dec_t acctbal;
char rec[39]; /* C-ISAM Data Record */
.
.
.
/* Load Transaction Amount and Account Balance from Record */
lddecimal(&rec[26],4,&tramt);
lddecimal(&rec[30],8,&acctbal);
```

LDFLOAT

Overview

Use **ldfloat** to return a machine-dependent floating-point number from a C-ISAM format FLOATTYPE.

Syntax

```
double ldfloat(p)
char *p;
```

Explanation

p is a pointer to the C-ISAM format FLOATTYPE number.

Notes

1. A C-ISAM FLOATTYPE has the same format as the C **float**, except that a C-ISAM number may not be aligned on a word boundary.
2. Floating-point numbers are returned as double-precision floating-point numbers.

Example

```
char rec[39]; /* C-ISAM Data File Record */
.
.
.
/* Retrieve Trans. Amt. and Acct. Balance from Record */
tramt = ldfloat(&rec[26]);
acctbal = lddbl(&rec[30]);
```

LDFLTNULL

Overview

Use **ldfltnull** to return a machine-dependent floating-point number from a C-ISAM format FLOATTYPE and simultaneously test if the value is null.

Syntax

```
double ldfltnull(p,nullflag)
char *p;
short *nullflag;
```

Explanation

<i>p</i>	is a pointer to the starting byte of the C-ISAM format FLOATTYPE number.
<i>nullflag</i>	is a pointer to the null code.

Notes

1. A C-ISAM FLOATTYPE has the same format as the C **float**, except that a C-ISAM number may not be aligned on a word boundary.
2. Floating point numbers are returned as double-precision floating point numbers.
3. If the value of the FLOATTYPE is null, then **ldfltnull** sets **nullflag* to 1, and returns a 0.
4. If the value of the FLOATTYPE is not null, then **ldfltnull** sets **nullflag* to 0, and returns the value.

Example

```
char rec[39]; /* C-ISAM Data File Record */
.
.
.
/* Retrieve Trans. Amt. and Acct. Balance from Record */
tramt = ldfltnull(&rec[26],nlflg);
acctbal = lddblnull(&rec[30],nlflg2);
```

LDINT

Overview

Use **ldint** to return a machine-dependent integer from a **C-ISAM** INTTYPE.

Syntax

```
short ldint(p)
char *p;
```

Explanation

p is a pointer to a **C-ISAM** integer.

Note

C-ISAM stores an INTTYPE integer as a two-byte signed binary integer with the most significant byte first.

Example

```
char rec[39]; /* C-ISAM Data File Record */
.
.
.
/* Get Customer Number and Status from Record */
custno = ldlong(&rec[0]);
cstatus = ldint(&rec[24]);
```

LDLONG

Overview

Use **ldlong** to return a machine-dependent **long** integer from a **C-ISAM** format LONGTYPE.

Syntax

```
long ldlong(p)
char *p;
```

Explanation

p is a pointer to the **C-ISAM** LONGTYPE number.

Note

C-ISAM stores a LONGTYPE integer as a four-byte signed binary integer with the most significant byte first.

Example

```
char rec[39]; /* C-ISAM Data File Record */
.
.
.
/* Get Customer Number and Status from Record */
custno = ldlong(&rec[0]);
cstatus = ldint(&rec[24]);
```

STCHAR

Overview

Use **stchar** to store a character string in a **C-ISAM** data record.

Syntax

```
stchar(cstr,fstr,length);
char *cstr;
char *fstr;
int length;
```

Explanation

<i>cstr</i>	is the character string in memory.
<i>fstr</i>	is a pointer to the starting byte of the destination C-ISAM character string.
<i>length</i>	is the length of the C-ISAM character string.

Notes

1. **C-ISAM** does not terminate a character string with a null character; instead it pads the string with trailing spaces.
2. The function removes the null character and pads the destination string with trailing blanks to the length specified by *length*.

Example

```
char rec[39];    /* C-ISAM data file record */
char cname[21]; /* Null-terminated string
                  without trailing blanks */
.
.
.
stchar(cname,&rec[4],20);
```

STDBL

Overview

Use **stdbl** to store a machine-dependent double-precision number in a C-ISAM DOUBLETTYPE.

Syntax

```
stdbl(d,p)
double d;
char *p;
```

Explanation

d is the double-precision number to be stored.
p is the pointer to the C-ISAM DOUBLETTYPE that receives the number.

Note

A C-ISAM DOUBLETTYPE has the same format as the C **double**, except that a C-ISAM number may not be aligned on a word boundary.

Example

```
char rec[39]; /* C-ISAM Data File Record */
.
.
.
/* Store Trans. Amt.
   and Acct. Balance into Record */
stfloat(tramt,&rec[26]);
stdbl(acctbal,&rec[30]);
```

STDBLNULL

Overview

Use **stdbldnull** to store a machine-dependent double-precision number or a null in a **C-ISAM DOUBLETTYPE**.

Syntax

```
stdbldnull(d,p,nullflag)
double d;
char *p;
short nullflag;
```

Explanation

<i>d</i>	is the double-precision number to be stored.
<i>p</i>	is the pointer to the C-ISAM DOUBLETTYPE that receives the number.
<i>nullflag</i>	is the null code.

Notes

1. A **C-ISAM DOUBLETTYPE** has the same format as the C **double**, except that a **C-ISAM** number may not be aligned on a word boundary.
2. If you set *nullflag* to one, a **C-ISAM NULL** is stored. If *nullflag* is set to 0, the value passed is stored.

Example

```
char rec[39]; /* C-ISAM Data File Record */
.
.
.
/* Store Trans. Amt.
   and Acct. Balance into Record */
stfloat(tramt,&rec[26]);
stdbldnull(acctbal,&rec[30],nlflag);
```

STDECIMAL

Overview

Use **stdecimal** to store a DECIMALTYPE number in a **dec_t** structure into a C-ISAM record in packed format.

Syntax

```
stdecimal (decp,cp,len)
    dec_t *dec;
    char *cp;
    int len;
```

Explanation

<i>dec</i>	is the dec_t structure that contains the decimal data.
<i>cp</i>	is a pointer to the position in the data record where the decimal data starts.
<i>len</i>	is the length of the decimal data in the data record.

Notes

1. DECIMALTYPE data is stored in a **dec_t** structure in your **C-ISAM** program. It is stored in packed format, however, within the **C-ISAM** file.
2. The length parameter *len* specifies the length of the packed data and is between 2 and 17 bytes, inclusive.
3. The packed length is the sum of the following three items: the number of significant digits to the left of the decimal point, divided by two and rounded up; the number of significant digits to the right of the decimal point, divided by two and rounded up; plus one byte. (See the section “Sizing DECIMALTYPE Numbers” in Chapter 3 for more information.)

Examples

```
char rec[39]; /* C-ISAM Data Record */
.
.
.
/* Store Transaction Amount and Account Balance into Record */
stdecimal(&tramt,&rec[26],4);
stdecimal(&acctbal,&rec[30],8);
```

STFLOAT

Overview

Use **stfloat** to store a machine-dependent floating-point number in a **C-ISAM FLOATTYPE** number.

Syntax

```
stfloat(f,p)
float f;
char *p;
```

Explanation

f is the floating-point number to be stored in **C-ISAM FLOATTYPE** format.

p is the pointer to the **C-ISAM FLOATTYPE** to receive the number.

Note

A **C-ISAM FLOATTYPE** has the same format as the **C float**, except that a **C-ISAM** number may not be aligned on a word boundary.

Example

```
char rec[39]; /* C-ISAM Data File Record */
/* Store Trans. Amt. and Acct. Balance into Record */
stfloat(tramt,&rec[26]);
stdbl(acctbal,&rec[30]);
```

STFLTNULL

Overview

Use **stfltnull** to store a machine-dependent floating-point number or a null in a C-ISAM FLOATTYPE number.

Syntax

```
stfltnull(f,p,nlflg)
float f;
char *p;
short nullflag;
```

Explanation

<i>f</i>	is the floating-point number to be stored in C-ISAM FLOATTYPE format.
<i>p</i>	is the pointer to the C-ISAM FLOATTYPE that receives the number.
<i>nullflag</i>	is the null code.

Notes

1. A C-ISAM FLOATTYPE has the same format as the C **float**, except that a C-ISAM number may not be aligned on a word boundary.
2. If *nullflag* = 1, a C-ISAM null is stored; if *nullflag* = 0, the passed value is stored.

Example

```
char rec[39]; /* C-ISAM Data File Record */
/* Store Trans. Amt. and Acct. Balance into Record */
stfltnull(tramt,&rec[26],nlflg);
stdbl(acctbal,&rec[30]);
```

STINT

Overview

Use **stint** to store a machine-dependent short integer in a C-ISAM INTTYPE number.

Syntax

```
stint(i,p)
short i;
char *p;
```

Explanation

i is the machine-dependent short integer to be stored.

p is a pointer to the C-ISAM INTTYPE number that receives the integer.

Note

C-ISAM stores an INTTYPE integer as a two-byte signed binary integer with the most significant byte first.

Example

```
char rec[39]; /* C-ISAM Data File Record */
.
.
.
/* Store Customer Number and Status into Record */
stlong(custno,&rec[0]);
stint (cstatus,&rec[24]);
```

STLONG

Overview

Use **stlong** to store a machine-dependent **long** integer in a C-ISAM format LONGTYPE.

Syntax

```
stlong(l,p)
long l;
char *p;
```

Explanation

l is the machine-dependent **long** integer.

p is the pointer to the C-ISAM format LONGTYPE that receives the number.

Note

C-ISAM stores a LONGTYPE integer as a four-byte signed binary integer with the most significant byte first.

Example

```
char rec[39]; /* C-ISAM Data File Record */
.
.
.
/* Store Customer Number and Status into Record */
stlong(custno,&rec[0]);
stint (cstatus,&rec[24]);
```

DECIMALTYPE Functions

Functions for manipulation of DECIMALTYPE numbers are described in the following pages:

DECCVASC

DECTOASC

DECCVINT

DECTOINT

DECCVLONG

DECTOLONG

DECCVFLT

DECTOFLT

DECCVDBL

DECTODBL

DECADD, DECSUB, DECMUL, and DECDIV

DECCMP

DECCOPY

DECECVT and DECFCVT

DECCVASC

Overview

Use **deccvasc** to convert a value held as printable characters in a C **char** type into a DECIMALTYPE number.

Syntax

```
deccvasc(cp, len, np)
char *cp;
int len;
dec_t *np;
```

Explanation

<i>cp</i>	points to a string that holds the value you want to convert.
<i>len</i>	is the length of the string.
<i>np</i>	is a pointer to a dec_t structure that receives the result of the conversion.

Notes

1. The **deccvasc** function ignores leading spaces in the character string.
2. The character string can have a leading plus (+) or minus (-) sign, a decimal point (.), and numbers to the right of the decimal point.
3. The character string can contain an exponent preceded by either *e* or *E*. The exponent may be preceded by a + or - sign.

Return Codes

-1216	Bad exponent
-1213	Non-numeric characters in string
-1201	Underflow; number is too small
-1200	Overflow; number is too large
-1	Error; iserrno contains the error code
0	Successful

Example

```
#include <decimal.h>

char input[80];
dec_t number;
    .
    .
/* Get input from terminal */
getline(input);

/* Convert input into decimal number */
deccvasc(input, 32, &number);
```

DECTOASC

Overview

Use **dectoasc** to convert a DECIMALTYPE number to a printable ASCII string.

Syntax

```
dectoasc(np, cp, len, right)
    dec_t *np;
    char *cp;
    int len;
    int right;
```

Explanation

<i>np</i>	is a pointer to the decimal structure whose associated decimal value you want to convert to an ASCII string.
<i>cp</i>	is a pointer to the beginning of the character buffer that holds the ASCII string.
<i>len</i>	is the maximum length in bytes of the string buffer.
<i>right</i>	is an integer indicating the number of decimal places to the right of the decimal point.

Notes

1. If *right* equals -1, the number of decimal places is determined by the decimal value of **np*.
2. If the number does not fit into a character string of length *len*, **dectoasc** converts the number to exponential notation. If the number still does not fit, **dectoasc** fills the string with asterisks. If the number is shorter than the string, it is left-justified and padded on the right with blanks.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
#include <decimal.h>

char input[80];
char output[16];
dec_t number;
    .
    .
    .

/* Get input from terminal */
getline(input);

/* Convert input into decimal number */
deccvasc(input, 32, &number);

/* Convert number to printable string */
dectoasc(&number, output, 16, 1);

/* Print the value just entered */
printf("You just entered %s", output);
```

DECCVINT

Overview

Use **deccvint** to convert a C type **short** into a DECIMALTYPE number.

Syntax

```
deccvint(integer, np)
    int integer;
    dec_t *np;
```

Explanation

<i>integer</i>	is the integer you want to convert.
<i>np</i>	is a pointer to a dec_t structure that receives the result of the conversion.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
#include <decimal.h>

dec_t decnum;

/* Convert the integer value -999
 * into a DECIMAL type number
 */
deccvint(-999, &decnum);
```

DECTOINT

Overview

Use **dectoint** to convert a DECIMALTYPE number into a C **int** type.

Syntax

```
dectoint(np, ip)
    dec_t *np;
    int *ip;
```

Explanation

<i>np</i>	is a pointer to a decimal structure whose value is converted to an integer.
<i>ip</i>	is a pointer to the integer.

Return Codes

-1200	DECIMALTYPE number greater than 32,767
-1	Error; iserrno contains the error code
0	Successful

Example

```
#include <decimal.h>

dec_t mydecimal;
int    myinteger;

/* Convert the value in
 * mydecimal into an integer
 * and place the results in
 * the variable myinteger.
 */
dectoint(&mydecimal, &myinteger);
```

DECCVLONG

Overview

Use **deccvlong** to convert a C type **long** value into a DECIMALTYPE number.

Syntax

```
deccvlong(lng, np)
    long lng;
    dec_t *np;
```

Explanation

<i>lng</i>	is a pointer to a long integer.
<i>np</i>	is a pointer to a dec_t structure that receives the result of the conversion.

Return Codes

-1200	DECIMALTYPE number greater than 2,147,483,647
-1	Error; iserrno contains the error code
0	Successful

Example

```
#include <decimal.h>

dec_t mydecimal;
long mylong;

/* Set the decimal structure
 * mydecimal to 37.
 */
deccvlong(37L, &mydecimal);

mylong = 123456L;
/* Convert the variable mylong into
 * a DECIMAL type number held in
 * mydecimal.
 */
deccvlong(mylong, &mydecimal);
```

DECTOLONG

Overview

Use **dectolong** to convert a DECIMALTYPE into a C type **long**.

Syntax

```
dectolong(np, lngp)
    dec_t *np;
    long *lngp;
```

Explanation

<i>np</i>	is a pointer to a decimal structure.
<i>lngp</i>	is a pointer to a long where the result of the conversion will be placed.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
#include <decimal.h>

dec_t mydecimal;
long mylong;

/* Convert the DECIMALTYPE value
 * held in the decimal structure
 * mydecimal to a long pointed to
 * by mylong.
 */
dectolong(&mydecimal, &mylong);
```

DECCVFLT

Overview

Use **deccvflt** to convert a C type **float** into a DECIMALTYPE number.

Syntax

```
deccvflt(flt, np)
    float flt;
    dec_t *np;
```

Explanation

<i>flt</i>	is a floating-point number.
<i>np</i>	is a pointer to a dec_t structure that receives the result of the conversion.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
#include <decimal.h>

dec_t mydecimal;
float myfloat;

/* Set the decimal structure
 * myfloat to 3.14159.
 */
deccvflt(3.14159, &mydecimal);

myfloat = 123456.78;

/* Convert the variable myfloat into
 * a DECIMALTYPE number held in
 * mydecimal.
 */
deccvflt(myfloat, &mydecimal);
```

DECTOFLT

Overview

Use **dectoflt** to convert a DECIMALTYPE number into a C type **float**.

Syntax

```
dectoflt(np, fltp)
    dec_t *np;
    float *fltp;
```

Explanation

np is a pointer to a decimal structure.

fltp is a pointer to a floating-point number to receive the result of the conversion.

Note: The resulting floating-point number has eight significant digits.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
#include <decimal.h>

dec_t mydecimal;
float myfloat;

/* Convert the DECIMALTYPE value
 * held in the decimal structure
 * mydecimal to a floating-point number pointed to
 * by myfloat.
 */
dectoflt(&mydecimal, &myfloat);
```

DECCVDBL

Overview

Use **deccvdbl** to convert a C type **double** into a DECIMALTYPE number.

Syntax

```
deccvdbl(dbl, np)
double dbl;
dec_t *np;
```

Explanation

<i>dbl</i>	is a double-precision floating-point number.
<i>np</i>	is a pointer to a dec_t structure that receives the result of the conversion.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
#include <decimal.h>

dec_t mydecimal;
double mydouble;

/* Set the decimal structure
 * mydecimal to 3.14159.
 */
deccvdbl(3.14159, &mydecimal);

mydouble = 123456.78;

/* Convert the variable mydouble into
 * a DECIMALTYPE number held in
 * mydecimal.
 */
deccvdbl(mydouble, &mydecimal);
```

DECTODBL

Overview

Use **dectodbl** to convert a DECIMALTYPE number into a C type **double**.

Syntax

```
dectodbl(np, dblp)
    dec_t *np;
    double *dblp;
```

Explanation

<i>np</i>	is a pointer to a decimal structure.
<i>dblp</i>	is a pointer to a double-precision floating-point number that receives the result of the conversion.

Note

The resulting double-precision number receives a total of 16 significant digits.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
#include <decimal.h>

dec_t mydecimal;
double mydouble;

/* Convert the DECIMALTYPE value
 * held in the decimal structure
 * mydecimal to a double pointed to
 * by mydouble.
 */
dectodbl(&mydecimal, &mydouble);
```

DECADD, DECSUB, DECMUL, and DECDIV

Overview

The decimal arithmetic routines take pointers to three decimal structures as parameters. The first two decimal structures hold the operands of the arithmetic function. The third decimal structure holds the result.

Syntax

```
decadd(n1, n2, result)/* result = n1 + n2 */  
    dec_t *n1;  
    dec_t *n2;  
    dec_t *result;
```

```
decsb(n1, n2, result)/* result = n1 - n2 */  
    dec_t *n1;  
    dec_t *n2;  
    dec_t *result;
```

```
decmul(n1, n2, result)/* result = n1 * n2 */  
    dec_t *n1;  
    dec_t *n2;  
    dec_t *result;
```

```
decdiv(n1, n2, result)/* result = n1 / n2 */  
    dec_t *n1;  
    dec_t *n2;  
    dec_t *result;
```

Explanation

<i>n1</i>	is a pointer to the decimal structure of the first operand.
<i>n2</i>	is a pointer to the decimal structure of the second operand.
<i>result</i>	is a pointer to the decimal structure of the result of the operation.

Note

The *result* pointer can be the same pointer as either *n1* or *n2*.

Return Codes

-1202	Attempt to divide by zero
-1201	Underflow; result is too small
-1200	Overflow; result is too large
-1	Error; iserrno contains the error code
0	Successful

DECCMP

Overview

Use **deccmp** to compare two DECIMALTYPE numbers.

Syntax

```
int deccmp(n1, n2)
    dec_t *n1;
    dec_t *n2;
```

Explanation

n1 is a pointer to the decimal structure of the first number.
n2 is a pointer to the decimal structure of the second number.

Return Codes

-1	<i>n1</i> is less than <i>n2</i>
0	The arguments are equal
1	<i>n1</i> is greater than <i>n2</i>

DECCOPY

Overview

Use **deccopy** to copy one **dec_t** structure to another.

Syntax

```
deccopy(n1, n2)
    dec_t *n1;
    dec_t *n2;
```

Explanation

<i>n1</i>	is a pointer to the source dec_t structure.
<i>n2</i>	is a pointer to the destination dec_t structure.

DECECVT and DECFCVT

Overview

These functions convert a DECIMALTYPE value to an ASCII string.

Syntax

```
char *dececvt(np, ndigit, decpt, sign)
    dec_t *np;
    int ndigit;
    int *decpt;
    int *sign;
```

```
char *decfcvt(np, ndigit, decpt, sign)
    dec_t *np;
    int ndigit;
    int *decpt;
    int *sign;
```

Explanation

<i>np</i>	is a pointer to a dec_t structure that contains the number that you want to convert.
<i>ndigit</i>	is, for dececvt , the length of the ASCII string; for decfcvt , it is the number of digits to the right of the decimal point.
<i>decpt</i>	points to an integer that is the position of the decimal point relative to the beginning of the string. A negative value for <i>*decpt</i> means to the left of the returned digits.
<i>sign</i>	is a pointer to the sign of the result. If the sign of the result is negative, <i>*sign</i> is nonzero; otherwise, the value is 0.

Notes

1. The **dececvt** function converts the decimal value pointed to by *np* into a null-terminated string of *ndigit* ASCII digits and returns a pointer to the string.
2. The low-order digit of the DECIMALTYPE number is rounded.
3. The **decfcvt** function is identical to **dececvt**, except that *ndigit* specifies the number of digits to the right of the decimal point instead of the total number of digits.

Examples

In the following example, let *np* point to 12345.67 and suppress all arguments except *ndigit*:

dececvt(4)	= "1235"	*decpt = 5
dececvt(10)	= "1234567000"	*decpt = 5
decfcvt(1)	= "123457"	*decpt = 5
decfcvt(3)	= "12345670"	*decpt = 5

In this example, let *np* point to .001234:

dececvt(4)	= "1234"	*decpt = -2
dececvt(10)	= "1234000000"	*decpt = -2
decfcvt(1)	= ""	*decpt = -2
decfcvt(3)	= "1"	*decpt = -2

Summary

This chapter describes all the functions that are available as part of **C-ISAM**.

- File manipulation functions
- Format-conversion and manipulation functions

The file manipulation functions allow you to perform the following operations:

- Create and remove files and indexes
- Access and modify records from within files
- Lock records or files
- Implement transactions
- Perform other functions associated with maintaining **C-ISAM** files

The format-conversion functions allow you to convert between machine-dependent representation of numbers and the **C-ISAM** counterparts. The format-manipulation routines allow you to manipulate the **C-ISAM** DECIMALTYPE data type.

The chapter includes explanations, syntax, return codes, and examples for each function.

The *bcheck* Utility

The **bcheck** program is a C-ISAM utility program that checks and repairs C-ISAM index files. It is distributed with C-ISAM. You should run it whenever a system crash occurs or whenever you suspect the integrity of a C-ISAM index.

The **bcheck** program compares an index file (**.idx**) to a data file (**.dat**) to see if the two are consistent. If they are not, **bcheck** asks you if you want to delete and rebuild the corrupted indexes.

You can use the **bcheck** utility with fixed-length or variable-length record files. The syntax for using **bcheck** with variable-length records, as shown here, is the same as using it with fixed-length records. Only the **-i** option has special functionality for variable-length records.

The **bcheck** utility does not repair the variable-length data portion of the index files.

bcheck - [**i** | **l** | **y** | **n** | **q** | **s**] *filename*

-i	Check index file only
-l	List entries in B+ trees
-n	Answer no to all questions
-y	Answer yes to all questions
-q	Suppress printing of the program banner
-s	Resize the index file node size. This option resets the NODESIZE parameter from the existing value to the value set in the code. This option does not change any of the characteristics of the index keys themselves.

Unless you use the **-n** or **-y** option, **bcheck** is interactive, waiting for you to respond to each error it finds.

Use the **-y** option with caution. Do not run **bcheck** using the **-y** option if you are checking the files for the first time.

Checking Indexes

If you use the **-i** option with fixed-length records, the index information contained in the index files is checked for consistency with the data files.

If you use the **-i** option with variable-length records, the entire contents of the index file are checked for free space as well as for consistency. This includes the variable-length data that is stored in the index file. The **bcheck** utility uses this information if it is necessary to rebuild the index file.

Resizing Nodes and Indexes

The **bcheck -s** option does not affect the variable-length records that reside in the index files. If you need to resize the keys within an index, use **iscluster** with a new **keydesc** structure.

Messages Received with Variable-Length Records

If you use **bcheck** with variable-length records, you will receive the following two messages with relevant values along with the rest of the standard **bcheck** messages:

```
64 index pages are used for variable-length record storage.  
15761 bytes are free in those pages, an average of 246 bytes per page.
```

Recovering Resources from Irretrievable Files

If you are using variable-length records and the files become severely corrupted, **bcheck** can repair the damaged index portion of the files, but it cannot repair damaged data records. Since the variable-length data is stored in the index files, you might not be able to retrieve the data.

To repair index files that contain corrupted variable-length data, you have to delete corrupted records with your own **C-ISAM** program. See “File Maintenance with Variable-Length Records” on page 6-10 for more information about retrieving data from corrupted **.idx** files.

Examples of Using *bcheck*

In the following example, **bcheck** checks all indexes for **custome100** and finds no errors. For each index, **bcheck** prints a group of up to eight numbers. These numbers indicate the position of the key in each record.

```
bcheck -n custome100

BCHECK  C-ISAM B-tree Checker version 5.00.UC1
Copyright (C) 1981-1991 Informix Software, Inc.
Software Serial Number INF#R000000

C-ISAM File: custome100

Checking dictionary and file sizes.
Index file node size = 1024
Current C-ISAM index file node size = 1024
Checking data file records.
Checking indexes and key descriptions.
Index 1 = unique key
      0 index node(s) used -- 1 index b-tree level(s) used
Index 2 = unique key  (0,4,2)
      1 index node(s) used -- 1 index b-tree level(s) used
Index 3 = duplicates  (111,5,0)
      1 index node(s) used -- 1 index b-tree level(s) used
Checking data record and index node free lists.
4 index node(s) used, 0 free -- 18 data record(s) used, 4 free
```

Here is a sample run in which **bcheck** finds errors. The **-n** option is selected so that each question that **bcheck** asks is automatically answered “no.”

```
BCHECK  C-ISAM B-tree Checker version 5.00.UC1
Copyright (C) 1981-1991 Informix Software, Inc.
Software Serial Number INF#R000000

C-ISAM File: customel00

Checking dictionary and file sizes.
Index file node size = 1024
Current C-ISAM index file node size = 1024
Checking data file records.
Checking indexes and key descriptions.
Index 1 = unique key
    0 index node(s) used -- 1 index b-tree level(s) used

ERROR:  3 bad data record(s)
Delete index  ? no

Index 2 = unique key  (0,4,2)
    1 index node(s) used -- 1 index b-tree level(s) used

ERROR:  3 bad data record(s)
Delete index  ? no

Index 3 = duplicates  (111,5,0)
    1 index node(s) used -- 1 index b-tree level(s) used

ERROR:  3 bad data record(s)
Delete index  ? no

Checking data record and index node free lists.

ERROR:  3 missing data record(s)
Fix data record free list  ? no

4 index node(s) used, 0 free -- 18 data record(s) used, 4 free
```

Since **bcheck** finds errors, you must delete and rebuild the corrupted indexes. The **-y** option is used to answer “yes” to all questions asked by **bcheck**:

```
BCHECK  C-ISAM B-tree Checker version 5.00.UC1
Copyright (C) 1981-1991 Informix Software, Inc.
Software Serial Number INF#R000000

C-ISAM File: customel00

Checking dictionary and file sizes.
Checking data file records.
Checking indexes and key descriptions.
Index 1 = unique key
      1 index node(s) used -- 1 index b-tree level(s) used

ERROR:  3 bad data record(s)
Delete index ? yes

Remake index ? yes
Index 2 = unique key  (0,4,2)
      1 index node(s) used -- 1 index b-tree level(s) used

ERROR:  3 bad data record(s)
Delete index ? yes

Remake index ? yes

Index 3 = duplicates  (111,5,0)
      1 index node(s) used -- 1 index b-tree level(s) used

ERROR:  3 bad data record(s)
Delete index ? yes

Remake index ? yes

Checking data record and index node free lists.

ERROR:  3 missing data record(s)
Fix data record free list ? yes

Recreate data record free list
Recreate index 3
Recreate index 2
Recreate index 1

4 index node(s) used, 0 free -- 18 data record(s) used, 4 free
```

Header Files

isam.h

Figure B-1 shows the contents of the **isam.h** header file.

```

#ifndef ISAM_INCL      /* avoid multiple include problems */
#define ISAM_INCL

#define CHARTYPE      0
#define DECIMALTYPE   0
#define CHARSIZE      1

#define INTTYPE       1
#define INTSIZE       2

#define LONGTYPE      2
#define LONGSIZE      4

#define DOUBLETYP     3
#ifndef NOFLOAT
#define DOUBLESIZE    (sizeof(double))
#endif /* NOFLOAT */

#ifndef NOFLOAT
#define FLOATTYPE     4
#define FLOATSIZE    (sizeof(float))
#endif /* NOFLOAT */

#define USERCOLL(x)   ((x))

#define COLLATE1      0x10
#define COLLATE2      0x20
#define COLLATE3      0x30
#define COLLATE4      0x40
#define COLLATE5      0x50
#define COLLATE6      0x60
#define COLLATE7      0x70

#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 */

```

```

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

#ifndef ldlong
long ldlong();
#endif

#ifndef NOFLOAT
#ifndef ldfloat
double ldfloat();
#endif
#ifndef lddbl
double lddbl();
#endif
double ldfltnull();
double lddblnull();
#endif

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

/* isread lock modes */
#define ISLOCK 0x100 /* record lock */
#define ISSKILOCK 0x200 /* skip record even if locked */
#define ISWAIT 0x400 /* wait for record lock */
#define ISLCKW 0x500 /* ISLOCK + ISWAIT */

/* isstart lock modes */
#define ISKEEPLOCK 0x800 /* keep rec lock in autolk mode */

/* 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 ISINPUT 0 /* open for input only */
#define ISOUTPUT 1 /* open for output only */
#define ISINOUT 2 /* open for input and output */
#define ISTRANS 4 /* open for transaction proc */
#define ISNOLOG 8 /* no loggin for this file */
#define ISVARLEN 0x10 /* variable length records */
#define ISFIXLEN 0x0 /* (non-flag) fixed length records only */

/* audit trail mode parameters */
#define AUDSETNAME 0 /* set new audit trail name */
#define AUDGETNAME 1 /* get audit trail name */
#define AUDSTART 2 /* start audit trail */
#define AUDSTOP 3 /* stop audit trail */
#define AUDINFO 4 /* audit trail running ? */

/*
 * Define MAXKEYSIZE 240 and NPARTS 16 for AF251
 */
#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_leng;            /* 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 allowed */
#define ISDUPS      001      /* duplicates allowed */
#define DCOMPRESS   002      /* duplicate compression */
#define LCOMPRESS   004      /* leading compression */
#define TCOMPRESS   010      /* trailing compression */
#define COMPRESS    016      /* all compression */
#define ISCLUSTER   020      /* index is a cluster one */

struct dictinfo
{
    short di_nkeys;          /* number of keys defined (msb set for VARLEN)*/
    short di_recsz;         /* (maximum) data record size */
    short di_idxsz;         /* 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 ELCKED       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 ENOLOK       115      /* can't create lock file */
#define EBADMEM      116      /* can't alloc memory */
#define EBADCOLL     117      /* bad custom collating */
#define ELOGREAD     118      /* cannot read log rec */
#define EBADLOG      119      /* bad log record */
#define ELOGOPEN     120      /* cannot open log file */
#define ELOGWRIT     121      /* cannot write log rec */
#define ENOTRANS     122      /* no transaction */
#define ENOSHMEM     123      /* no shared memory */
#define ENOBEGIN     124      /* no begin work yet */
#define ENONFS       125      /* can't use nfs */
#define EBADROWID    126      /* reserved for future use */

```

```

#define ENOPRIM 127      /* no primary key */
#define ENOLOG 128      /* no logging */
#define EUSER 129       /* reserved for future use */
#define ENODBS 130      /* reserved for future use */
#define ENOFREE 131     /* no free disk space */
#define EROWSIZE 132    /* row size too big */
#define EAUDIT 133      /* audit trail exists */
#define ENOLOCKS 134    /* no more locks */
#define ENOPARTN 135    /* reserved for future use */
#define ENOEXTN 136     /* reserved for future use */
#define EOVBUNK 137     /* reserved for future use */
#define EOVBDBS 138     /* reserved for future use */
#define EOVBLOG 139     /* reserved for future use */
#define EGBLSECT 140    /* global section disallowing access - VMS */
#define EOVBPARTN 141   /* reserved for future use */
#define EOVBPPAGE 142   /* reserved for future use */
#define EDEADLOK 143    /* reserved for future use */
#define EKLOCKED 144    /* reserved for future use */
#define ENOMIRROR 145   /* reserved for future use */
#define EDISKMODE 146   /* reserved for future use */
#define EARCHIVE 147    /* reserved for future use */
#define ENEMPTY 148     /* reserved for future use */
#define EDEADDEM 149    /* reserved for future use */
#define EDEMO 150       /* demo limits have been exceeded */
#define EBADVCLLEN 151   /* reserved for future use */
#define EBADRMSG 152    /* reserved for future use */
#define ENOMANU 153     /* must be in ISMANULOCK mode */
#define EDEADTIME 154   /* reserved for future use */
#define EPMCHKBAD 155   /* reserved for future use */
#define EB_BUSY 160     /* reserved for future use */
#define EB_NOOPEN 161   /* reserved for future use */
#define EB_NOBS 162     /* reserved for future use */
#define EB_PAGE 163     /* reserved for future use */
#define EB_STAMP 164     /* reserved for future use */
#define EB_NOCOL 165    /* reserved for future use */
#define EB_FULL 166     /* reserved for future use */
#define EB_PSIZE 167    /* reserved for future use */
#define EB_ARCH 168     /* reserved for future use */
#define EB_CHKNLOG 169   /* reserved for future use */
#define EB_IUBS 170     /* reserved for future use */
#define EBADFORMAT 171  /* locking or NODESIZE change */

/* Dismountable media blobs errors */
#define EB_SFULL 180     /* reserved for future use */
#define EB_NOSUBSYS 181 /* reserved for future use */
#define EB_DUPBS 182     /* reserved for future use */
/* Shared Memory errors */
#define ES_PROCDEFS 21584 /* can't open config file */
#define ES_IILLVAL 21586 /* illegal config file value */
#define ES_ICONFIG 21595 /* bad config parameter */
#define ES_ILLSRS 21596 /* illegal number of users */
#define ES_IILLCKS 21597 /* illegal number of locks */
#define ES_IILLFILE 21598 /* illegal number of files */
#define ES_IILLBUFF 21599 /* illegal number of buffs */
#define ES_SHMGET 25501 /* shmget error */
#define ES_SHMCTL 25502 /* shmctl error */
#define ES_SEMGET 25503 /* semget error */
#define ES_SEMCTL 25504 /* semctl error */

/*
 * 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()     */

#define IO_IDX      0x01    /* index file   */
#define IO_DAT      0x02    /* data file    */
#define IO_AUD      0x03    /* audit file   */
#define IO_LOK      0x04    /* lock file    */
#define IO_SEM      0x05    /* semaphore file */

/*
 * NOSHARE was needed as an attribute for global variables on VMS systems
 * It has been left here to make sure that it is defined for the
 * plethora of scattered references.
 */
#define NOSHARE

extern int iserrno;    /* isam error return code */
extern int iserrio;    /* system call error 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 status characters */
extern char isstat2;
extern char isstat3;
extern char isstat4;
extern char *isversnumber; /* C-ISAM version number */
extern char *iscopyright;  /* RDS copyright */
extern char *isserial;     /* C-ISAM software serial number */
extern int issingleuser;   /* set for single user access */
extern int is_nerr;        /* highest C-ISAM error code */
extern char *is_errlist[]; /* C-ISAM error messages */
/* error message usage:
 *     if (iserrno >= 100 && iserrno < is_nerr)
 *         printf("ISAM error %d: %s\n", iserrno, is_errlist[iserrno-100]);
 */

struct audhead
{
    char au_type[2];    /* audit record type aa,dd,rr,ww*/
    char au_time[4];    /* audit date-time */
    char au_procid[2];  /* process id number */
    char au_userid[2];  /* user id number */
    char au_recnum[4];  /* record number */
    char au_reclen[2];  /* audit record length beyond header */
};
#define AUDHEADSIZE 14    /* num of bytes in audit header */
#define VAUDHEADSIZE 16  /* VARLEN num of bytes in audit header */

#endif /* ISAM_INCL */

```

Figure B-1

Contents of isam.h File

decimal.h

You must include the file `decimal.h` in every program that uses the `DECIMALTYPE` data type. The header file defines the internal structure of `DECIMALTYPE` numbers. Your program accesses the internally stored `DECIMALTYPE` numbers only through the functions that are provided for this purpose. It should never access the internal structures directly. The explanation of this structure is provided here for reference only.

Memory Storage Structure

`DECIMALTYPE` numbers consist of an exponent and a mantissa (or fractional part) in base 100. In normalized form, the first digit of the mantissa must be greater than zero.

When used within a program, `DECIMALTYPE` numbers are stored in a C structure of the type shown in Figure B-2.

```
#ifndef DECSIZE
#define DECSIZE 16
#define DECUNKNOWN -2

struct decimal
{
    short dec_exp; /* exponent base 100 */
    short dec_pos; /* sign: 1=pos, 0=neg, -1=null */
    short dec_ndgts; /* number of significant digits */
    char dec_dgts[DECSIZE]; /* actual digits base 100 */
};
typedef struct decimal dec_t;
```

Figure B-2

Structure of a decimal or dec_t Data Type

The `dec_t` structure has four parts.

dec_exp	holds the exponent of the normalized <code>DECIMALTYPE</code> number. This exponent represents a power of 100.
dec_pos	holds the sign of the <code>DECIMALTYPE</code> number (1 when the number is zero or greater, and 0 when less than zero).
dec_ndgts	contains the number of base 100 significant digits of the <code>DECIMALTYPE</code> number.
dec_dgts	is a character array that holds the significant digits of the normalized <code>DECIMALTYPE</code> number (dec_dgts[0] != 0). Each character in the array is a one-byte binary number in base 100. The number of significant digits in dec_dgts is contained in dec_ndgts .

All operations on DECIMALTYPE numbers take place through the routines provided in **C-ISAM** that are described in Chapter 3, “Data Types.” Any other operations, modifications, or use of **dec_t** structures can produce unpredictable results.

File Storage Structure

When DECIMALTYPE numbers are stored in files, they are compressed or packed, as shown here:

First byte:

top 1 bit is the sign of the number

on = the number is positive

off = the number is negative

low 7 bits are the exponent in excess of 64

Remaining bytes:

base 100 digits (in 100 complement format for negative numbers)

The length in bytes of the packed DECIMALTYPE number is 1 plus the number of base 100 digits. It can vary from 2 to 17 bytes. This format permits sorts of DECIMALTYPE numbers using a simple unsigned byte-by-byte compari-

son. Zero is represented as 80,00,00,... (in hexadecimal). Figure B-3 shows the contents of the header file **decimal.h** that you must include in every program that uses the DECIMALTYPE data type.

```

#ifndef _DECIMAL_H
#define _DECIMAL_H

/*
 * Unpacked Format (format for program usage)
 *
 *   Signed exponent "dec_exp" ranging from -64 to +63
 *   Separate sign of mantissa "dec_pos"
 *   Base 100 digits (range 0 - 99) with decimal point
 *   immediately to the left of first digit.
 */

#define DECSIZE 16
#define DECUNKNOWN -2

struct decimal
{
    short dec_exp;           /* exponent base 100          */
    short dec_pos;           /* sign: 1=pos, 0=neg, -1=null */
    short dec_ndgts;         /* number of significant digits */
    char dec_dgts[DECSIZE]; /* actual digits base 100      */
};

typedef struct decimal dec_t;

/*
 * A decimal null will be represented internally by setting dec_pos
 * equal to DECPOSNULL
 */

#define DECPOSNULL          (-1)

/*
 * DECLEN calculates mininum number of bytes
 * necessary to hold a decimal(m,n)
 * where m = total # significant digits and
 *       n = significant digits to right of decimal
 */

#define DECLEN(m,n)         (((m)+(n)&1)+3)/2)
#define DECLLENGTH(len)    DECLEN(PRECTOT(len),PRECDEC(len))

/*
 * DECPREC calculates a default precision given
 * number of bytes used to store number
 */

#define DECPREC(size)       (((size-1)<<9)+2)

/* macros to look at and make encoded decimal precision

```

```

*
* PRECTOT(x)          return total precision (digits total)
* PRECDEC(x)          return decimal precision (digits to right)
* PRECMAKE(x,y)       make precision from total and decimal
*/

#define PRECTOT(x)      (((x)>>8) & 0xff)
#define PRECDEC(x)      ((x) & 0xff)
#define PRECMAKE(x,y)   (((x)<<8) + (y))

/*
* Packed Format (format in records in files)
*
*   First byte =
*       top 1 bit = sign 0=neg, 1=pos
*       low 7 bits = Exponent in excess 64 format
*   Rest of bytes = base 100 digits in 100 complement format
*   Notes -- This format sorts numerically with just a
*             simple byte by byte unsigned comparison.
*             Zero is represented as 80,00,00,... (hex).
*             Negative numbers have the exponent complemented
*             and the base 100 digits in 100's complement
*/

#endif /* _DECIMAL_H */

```

Figure B-3

Contents of decimal.h File

Error Codes

Four bytes, **isstat1**, **isstat2**, **isstat3**, and **isstat4**, return status information after **C-ISAM** calls. These bytes are primarily used by COBOL programs that use **C-ISAM** files. **isstat1** holds general status information, such as the success or failure of a **C-ISAM** call; **isstat2** contains more specific information that has meaning based on the status code in **isstat1**.

The following figure lists the values of **isstat1**:

isstat1 Value	Description
0	Successful Completion
1	End of File
2	Invalid Key
3	System Error
9	User Defined Errors

A large, bold, black serif capital letter 'C' is positioned on the right side of the page, partially overlapping a gray vertical bar.

The following figure shows the values of **isstat2** in conjunction with **isstat1**:

isstat1	isstat2	Indication
0 - 9	0	No further information is available.
0	2	Duplicate key found. After a READ, 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 WRITE or REWRITE, 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 COBOL program has changed the primary key value between the successful execution of a READ statement and the execution of the next REWRITE 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.

The following figure explains the combinations of **isstat3** and **isstat4**:

isstat3	isstat4	Indication
0	0	Successful completion; no further information is available.
0	2	Successful completion; duplicate key found. After a READ, 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 WRITE or REWRITE, this indicates that the record just written created a duplicate key value for at least one alternative record key for which duplicates are allowed.
1	0	Beginning or end of file was reached without successful completion
2	2	An attempt was 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	The filename specified in the isopen() function does not exist.
3	7	The mode parameter specified in the isopen() function is not allowed for the file.
3	9	There is a conflict between the fixed file attributes and the mode parameter specified in the isopen() function.
4	2	An attempt was made to close a file that was not open.
4	3	This call requires a current record. Either there is no current record, or the current record has been deleted.
4	4	An attempt was made to WRITE or REWRITE a record that is larger or smaller than is allowed for the file.

isstat3	isstat4	Indication
4	6	A READ with ISNEXT was attempted and there is no valid next record, either because no current record is defined or because the previous READ encountered an end condition.
4	7	A READ or isstart() was attempted on a file not opened with mode ISINPUT or ISINOUT.
4	8	A WRITE or iswrcurr() was attempted on a file not opened with mode ISOUTPUT or ISINOUT.
4	9	A DELETE, REWRITE, isdelrec() , isdelcurr() , isrewrec() , or isrewcurr() was attempted on a file not opened with mode ISINOUT.
9		Implementor-defined errors; the value of isstat4 is defined by the implementor.

The following figure shows the relationships between the **isstat** variables and the **ISAM** error codes. For other errors that do not support **isstat3** and **isstat4**, **isstat3** equals **isstat1** and **isstat4** equals **isstat2**.

Name	Number	Description	isstat1	isstat2	isstat3	isstat4
EDUPL	100	An attempt was made to add a duplicate value to an index via iswrite , isrewrite , isrewcurr , or isaddindex .	2	2	2	2
ENOTOPEN	101	An attempt was made to perform some operation on a C-ISAM file that was not previously opened using the isopen call.	9	0	4 4 4 4 4	2 7 8 9 0
EBADARG	102	One of the arguments of the C-ISAM call is not within the range of acceptable values for that argument.	9	0	3 3 4 9	7 9 4
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	9	0
ETOOMANY	104	The maximum number of files that can be open at one time would be exceeded if this request were processed.	9	0	9	0
EBADFILE	105	The format of the C-ISAM file has been corrupted.	9	0	9	0
ENOTEXCL	106	To add or delete an index, the file must have been opened with exclusive access.	9		9	

Name	Number	Description	isstat1	isstat2	isstat3	isstat4
ELOCKED	107	The record or file requested by this call cannot be accessed because another user has locked it.	9		9	
EKEXISTS	108	An attempt was made to add an index that has been defined previously.	9		9	
EPRIMKEY	109	An attempt was made to delete the primary key value. The primary key may not be deleted by the isdelindex call.	9		9	
EENDFILE	110	The beginning or end of file was reached.	1	0	1 4	0 6
ENOREC	111	No record could be found that contained the requested value in the specified position.	2	3	2	3
ENOCURR	112	This call must operate on the current record. The current record is not defined.	2	1	4 4	3 6
EFLOCKED	113	Another user has locked the file exclusively.	9		9	
EFNAME	114	The filename is too long.	9		9	0
ENOLOK	115	The lock file cannot be created.	9	0		
EBADMEM	116	Adequate memory cannot be allocated.	9		9	
EBADCOLL	117	Bad custom collating.	9	0		
ELOGREAD	118	Cannot read log file record.	9	0		

Name	Number	Description	isstat1	isstat2	isstat3	isstat4
EBADLOG	119	Record format of transaction log file cannot be recognized.	9	0		
ELOGOPEN	120	Cannot open transaction log file.	9	0		
ELOGWRIT	121	Cannot write to transaction log file.	9	0		
ENOTRANS	122	Not in transaction.	9	0		
ENOBEGIN	124	Beginning of transaction not found.	9	0		
ENONFS	125	Cannot use Network File Server.	9	0		
EBADROWID	126	Bad record number.	9	0		
ENOPRIM	127	No primary key.	9	0		
ENOLOG	128	No logging.	9	0		
EUSER	129	Too many users.	9	0		
ENOFREE	131	No free disk space.	9	0		
EROWSIZE	132	Record too long.	9	0		
EAUDIT	133	Audit trail exists.	9	0		
ENOLOCKS	134	No more locks.	9	0		
EDEMO	150	Demo limits have been exceeded.	9	0		
ENOMANU	153	Must be in ISMANULOCK mode	9	0		
EBADFORMAT	171	Incompatible file format	9	0		

File Formats

C-ISAM uses the following four kinds of file formats:

- Index file formats
- Data file formats
- Audit trail file formats
- Transaction file formats

The following sections present the formats of these nodes. The relationships between the nodes are discussed in Chapter 2, “Indexing.” This section describes all the file formats. You can use this section as a complete reference.

Index File Formats

C-ISAM index files (.idx) contain the following nodes:

- Dictionary node
- Key description node
- Remainder storage node
- B+ tree node
- Free-list node
- Audit trail node

The dictionary node has two fields to support variable-length records. The remainder storage node is used exclusively for variable-length records. The next sections describe these and other nodes in their entirety.

Dictionary Node

Byte Offset	Number of Bytes	Item	Value
0	2	validation	FE53
2	1	number of reserved bytes at start of index node	2
3	1	number of reserved bytes at end of index node	2
4	1	number of reserved bytes per key entry. Includes record number.	4
5	1	reserved	4
6	2	index file node length - 1.	(511 or 1023)
8	2	number of keys	
10	2	reserved	
12	1	file version number	
13	2	data record length in bytes	
15	4	index node number of first key description	
19	6	reserved	
25	4	index node number of free data record list	
29	4	index node number of free index node list	
33	4	record number of last record in data file	
37	4	index node number of last node in index file	
41	4	transaction number	
45	4	unique id	
49	4	pointer to audit trail information	
53	2	maximum data record length	
55	4	free group 0 hash pointer	

(1 of 2)

Byte Offset	Number of Bytes	Item	Value
59	4	free group 1 hash pointer	
63	4	free group 2 hash pointer	
67	4	free group 3 hash pointer	
71	4	free group 4 hash pointer	

(2 of 2)

Key Description Node

	Byte Offset	Number of Bytes	Item	Value	
	0	2	Number of bytes used in this node		
	2	4	Index node for continuation of key descriptions		
	6	2	length of description		
	8	4	Index node number of root		
	12	1	Compression flags		
These three items repeat for each part of the key	13	2	Length of key part 1 (top bit = duplicates)		These six items repeat for each key
	15	2	Position in data record		
	17	1	Data type parameter		
	n-2	1	Flag	FF	
	n-1	1	End of key description node	7E	

Remainder Storage Node

Byte Offset	Number of Bytes	Item	Value
0	2	Reserved	
2	2	Constant number	7E26
4	4	Forward pointer in hashed remainder storage page free list	
8	4	Backward pointer in hashed remainder storage page free list	
12	2	Free space available in this remainder storage page	4
14	2	Offset to free space in this remainder storage page	
16	4	Remainder pointer to next remainder space, if any.	
20	1	Flags	
21	1	Number of slots allocated	
22	1	Hash group for free list use	
23	varies	Data storage space	
	varies	Free space	
	4	Slot table, lowest entry	
	4	Slot table entry	
		
n-6	4	Slot table, highest entry	
n-2	1	Type	7C
n-1	1	Reserved	

B+ Tree Node

Byte Offset	Number of Bytes	Item	Value
0	2	Number of bytes used in this node	
2	1	Count of leading bytes, if compressed	
3	1	Count of trailing blanks, if compressed	
4	k	Key (may be compressed)	
4+k	2	For duplicate key, if compressed	
6+k	4	Pointer to data record (top bit may be duplicates flag)	
n-2	1	Index tree number	
n-1	1	Level in tree	0 = leaf node

Repeats for each key entry

Free-list Node

Byte Offset	Number of Bytes	Item	Value
0	2	Number of bytes used in this node (n)	
2	4	Count of leading bytes, if compressed	
6	n-8	Count of trailing blanks, if compressed	
n-2	1	Indicates data or index file	FF = data file FE = index file
n-1	1	End of list node flag	7F

Audit Trail Node

Byte Offset	Number of Bytes	Item	Value
0	2	Number of bytes used in this node (n)	
2	2	Flags	0 = audit trail is on 1 = audit trail is off
4	64	Audit trail pathname	
.	.	.	
n-1	1	End of list node flag	7D

Data File Format

Data files (**.dat**) contain only fixed-length data records, a flag at the end of each record, and if the data has a variable-length portion, two additional fields describing the length and placement of the variable-length portion.

If the flag is equal to zero (ASCII null), the record is deleted. Figure D-1 shows the data file format.

Figure D-1
Data file (.dat) format

Byte Offset	Number of Bytes	Item
0	r	Record with length r
r	1	Delete flag
r+1	2	The length of the valid data in the remainder portion; can be less than space allocated.
r+3	4	The first byte is the slot number (where first part of remainder is stored); the last 3 bytes are the remainder node number.

Audit Trail File Format

The audit trail file contains records consisting of a fixed-length header and an image of a data record. If the audit trail is associated with a file that contains variable-length records, it contains a two-byte entry that indicates the actual length of the data record. (This entry is not used in audit trails for fixed-length record files.) The following figure shows the format for audit trail files for variable-length records.

Figure D-2
Audit Trail Records

Byte Offset	Number of Bytes	Item	Value
0	2	audit trail record type	aa = record added dd = record deleted rr = record before update ww = record after update
2	4	time	
6	2	process identification number	
8	2	user identification	
10	4	data file record number	
14	2	actual length of variable-length data in bytes	
16	r	image of data record	
r+16			

If the operation is a rewrite, both the before and after images will be recorded in the audit trail file. The before image is listed first as an **rr** type, and it is followed by the after image as a **ww** type. Both have the same record number.

Transaction File Formats

Transaction file records contain a fixed-length header and other information, which depends upon the transaction type. Figure D-3 shows the format of the header.

Figure D-3
Transaction Record Header Format

Byte Offset	Number of Bytes	Item
0	2	Length of the log record
2	2	Transaction type
4	2	Transaction identification
6	2	User identification
8	L	Transaction time, L is the size of a long int. Measured as the number of seconds since midnight, 1970-01-01.
8 + L	8	Reserved
16 + L		

The transaction log file format header is the same for variable-length records as for fixed-length records. The following example lists all the transaction types:

```
/* record header definition */

#define LG_LEN 0/* current record length */
#define LG_TYPE LG_LEN+INTSIZE/* log record type */
#define LG_XID LG_TYPE+2/* transaction id */
#define LG_USER LG_XID+INTSIZE/* user name */
#define LG_TIME LG_USER+2/* transaction time */
#define LG_PREV LG_TIME+LONGSIZE/* previous log record */
#define LG_PREVLEN LG_PREV+LONGSIZE/* previous log length */

/* BEGIN, COMMIT, and ROLLBACK WORK record definition */
#define LG_TXSIZE LG_PREVLEN+INTSIZE+INTSIZE
/* record size */

/* build file record definition */
#define LG_FMODE LG_PREVLEN+INTSIZE/* build mode */
#define LG_RECLEN LG_FMODE+INTSIZE
/* minimum record length */
#define LG_MAXLEN LG_RECLEN+INTSIZE/* max rec length or zero */
#define LG_KFLAGS LG_MAXLEN+INTSIZE/* key flag */
#define LG_NPARTS LG_KFLAGS+INTSIZE/* number of key parts */
#define LG_KLEN LG_NPARTS+INTSIZE/* total key length */

/* erase file record definition */
```

Transaction File Formats

```
#define LG_FNAME LG_PREVLEN+INTSIZE/* directory path name */

/* rename file record definition */
#define LG_OLEN LG_PREVLEN+INTSIZE/* length of old filename */
#define LG_NLEN LG_OLEN+INTSIZE/* length of new filename */
#define LG_ONAME LG_NLEN+INTSIZE/* old filename */

/* open and close file record definition */
#define LG_ISFD LG_PREVLEN+INTSIZE/* isfd of file */
#define LG_VARLEN LG_ISFD+INTSIZE/* VARLEN flag of file */
#define LG_FPATH LG_VARLEN+INTSIZE/* directory path name */

/* create and drop index */
#define LG_IFLAGS LG_ISFD+INTSIZE/* key flags */
#define LG_INPARTS LG_IFLAGS+INTSIZE
/* number of key parts */
#define LG_IKLEN LG_INPARTS+INTSIZE/* total key length */

/* set unique id */
#define LG_UNIQID LG_ISFD+INTSIZE/* new unique id */

/* before or after image record definition */
#define LG_RECNO LG_ISFD+INTSIZE/* record number */
#define LG_IMGLEN LG_RECNO+LONGSIZE/* record image length */
#define LG_RECORD LG_IMGLEN+INTSIZE/* record data */

/* update image (before and after together) */
#define LG_BEFLLEN LG_RECNO+LONGSIZE
/* length of before image*/
#define LG_AFTLEN LG_BEFLLEN+INTSIZE/* length of after image*/
#define LG_BUPDATE LG_AFTLEN+INTSIZE
/* before image for update*/
/* (followed by the afterimage) */

/* savepoint record */
#define LG_SAVEPT LG_PREVLEN+INTSIZE/* savepoint number */
#define LG_SSIZE LG_SAVEPT+INTSIZE/* record size */

/* log memo record */
#define LG_LOCATION LG_PREVLEN+INTSIZE
#define LG_ISERRNO LG_LOCATION+LONGSIZE
#define LG_ERRNO LG_ISERRNO+INTSIZE
#define LG_ISERRIO LG_ERRNO+INTSIZE
#define LG_ISSTAT1 LG_ISERRIO+INTSIZE
#define LG_ISSTAT2 LG_ISSTAT1+1
#define LG_ISSTAT3 LG_ISSTAT2+1
#define LG_ISSTAT4 LG_ISSTAT3+1
```

```

#define LG_TEXT LG_ISSTAT4+1

#define LG_PAGESIZE 4096/* default log buff size */

/* log record types */
#define LG_ERROR 0/* log read or write error */
#define LG_BEGWORK 1/* BEGIN WORK */
#define LG_COMWORK 2/* COMMIT WORK */
#define LG_ROLWORK 3/* ROLLBACK WORK */
#define LG_DELETE 4/* deleted record */
#define LG_INSERT 5/* newly inserted record */
#define LG_UPDATE 6/* updated record */
#define LG_VERSION 7/* version */
#define LG_SVPOINT 8/* savepoint */
#define LG_FOPEN 9/* open file */
#define LG_FCLOSE 10/* close file */
#define LG_CKPOINT 11/* checkpoint */
#define LG_BUILD 12/* build new file */
#define LG_ERASE 13/* erase old file */
#define LG_RFORWARD 14/* ROLLFORWARD */
#define LG_CINDEX 15/* create index */
#define LG_DINDEX 16/* drop index */
#define LG_EOF 17/* end of log file */
#define LG_RENAME 18/* rename file */
#define LG_SETUNIQID 19/* set unique id */
#define LG_UNIQUEID 20/* get unique id */
#define LG_RBSVPT 21/* rollback to savepoint */
#define LG_CLUSIDX 22 /* create cluster index */
#define LG_MEMO 23 /* log file memo record */

#define TRUE 1
#define FALSE 0

#define NOPNFL 16

struct txlist
{
    int tx_xid; /* transaction id */
    struct xrloc *tx_nextrec; /* next log rec in transaction */
    struct txlist *tx_next; /* next transaction */
};

struct xrloc
{
    int xr_logtype; /* log record type */
    int xr_size; /* log record size */
    long xr_loc; /* location in log file */
    struct xrloc *xr_next; /* next log rec in transaction */
};

```




System Administration

Overview

This appendix discusses installation issues and system administration facilities that are available with **C-ISAM**. You should use this appendix in conjunction with the installation instructions that come with **C-ISAM**.

Installation

The following sections identify the files that are included with your **C-ISAM** system and explain how to set the ISAM-BUFS parameter for buffered input and output.

Files

Your installation media for the **C-ISAM** system contains several program files that are installed by the commands in **installisam**. (Refer to the installation instructions that come with the product for exact instructions on how to run these commands.)

The files that you need for programs that use **C-ISAM** files are as follows:

isam.h	must be included in each program.
decimal.h	must be included in all programs that reference the Decimal data type. (See Chapter 3, “Data Types.”)

E

libisam.a	is used whenever you compile a program that uses C-ISAM files. (See Chapter 1, “How to Use C-ISAM,” for compilation instructions.)
bcheck	can be used to check the integrity of a C-ISAM file. (See Appendix A, “The <i>bcheck</i> Utility.”)

Several sample programs also come with **C-ISAM**. You can compile and execute them to demonstrate that the files are correctly installed.

Buffers

C-ISAM uses buffering by the operating system to reduce the number of disk I/O operations required during execution of function calls. In addition to operating system buffers, **C-ISAM** maintains its own buffer pool to reduce the number of times that it calls the operating system to perform I/O. These internal buffers, therefore, further reduce overhead during **C-ISAM** calls. The parameter **ISAMBUFS** allows you to specify the number of internal buffers that are available to **C-ISAM**.

As a rule of thumb, you should allocate four buffers for every index that is in use at any one time. You must allocate a minimum of four buffers (total). The default **ISAMBUFS** value is 16.

If you are using the Bourne or Korn shell, enter the following commands:

```
ISAMBUFS=xx
export ISAMBUFS
```

If you are using the C shell, enter the following command:

```
setenv ISAMBUFS xx
```

In all cases, **xx** is the number of buffers you want to use, for example 4, 16, or some other number.

Locking System

Depending on the operating environment, **C-ISAM** implements locking in one of two ways. On most UNIX platforms, **C-ISAM** uses a kernel locking call to optimize the locking of files and records. Where the operating system does

not have a reliable and efficient locking-system call, **C-ISAM** creates and manipulates a system of files to track record locks in the absence of a locking call.

Transaction Logging and Recovery

You can use the transaction log file to write a program that recovers **C-ISAM** files. The program opens the log file and issues the **isrecover** call, as follows:

```
islogopen(logfile);  
isrecover();  
islogclose();
```

Ordinarily, your program would include error checking in addition to the **islogopen**, **isrecover**, and **islogclose** function calls.

Before you execute this program, you must restore the **C-ISAM** files that you want to recover from their backup media.

All programs that access recoverable **C-ISAM** files must have the same log file; otherwise, transaction recovery will not succeed. If you discover that a program made unlogged changes to a **C-ISAM** file or that different log files are being used concurrently, take the following actions:

1. Stop all programs that are using the **C-ISAM** file.
2. Make a backup copy of the **C-ISAM** file.
3. Restart all programs using the same *new* log file.

If you discover after recovery becomes necessary that unlogged changes were made to a **C-ISAM** file or that different log files are being used concurrently, you cannot guarantee integrity by using **isrecover**.

Index

A

- Access method
 - definition of Intro-3
 - implementation 2-13
 - indexed sequential 1-9
- Access modes 1-16, 1-29, 8-15
- Add a record
 - example 7-7
 - explanation 1-19
 - see also* Write a record
- Add an index
 - example 2-8, 7-6
 - explanation 2-8
 - isaddindex 2-8, 8-8
- Additional facilities
 - audit trail 6-6
 - file maintenance 6-3
 - force output 6-4
 - summary 6-13
- Audit trail
 - command modes 6-7
 - creating a new trail 8-11
 - example using isaudit 6-6
 - explanation 6-6
 - file format 6-8
 - isaudit 8-10
 - use of 6-6
 - with variable-length records 6-7

B

- bcheck
 - description and use of A-1
 - use 6-10
 - use in migrating files Intro-6

- use with variable-length record files 1-6
- Begin a transaction, `isbegin` 5-4, 8-13
- Block
 - definition of 2-22
 - see also* Index
- Buffers, `ISAMBUFS` parameter E-2
- Build a file
 - example 1-16, 7-5
 - explanation 1-13
 - `isbuild` 8-15
 - locking mode 4-6
 - record size 1-14
- B+ tree
 - adding to 2-16
 - delete from 2-21
 - growth of 2-17, 2-18
 - levels 2-15
 - maximum keys per node 2-15
 - nodes 2-13
 - organization 2-12
 - pointers 2-13
 - root 2-13
 - searching 2-15
 - sequential addition to 2-18
 - split 2-17
- Cluster index 6-9, 8-20
- Commit a transaction, `iscommit` 5-4, 8-22
- Compilation
 - header files 1-31
 - using `lint` 1-31
 - see also* System administration
- Compression of keys
 - see* Key
- Concurrency control
 - degree of concurrency 4-11
 - in transactions 5-8, 5-10
 - locking 4-3
- Conventions
 - typographical Intro-5
- Conversion functions
 - see* Format-conversion functions
- Create a file
 - see* Build a file
- Current index
 - see* Index
- Current record
 - definition of 1-19
 - set by `isread` 1-26, 8-43
 - set by `iswrcurr` 1-20, 8-65
 - set by `iswrite` 8-67

C

- C library functions
 - comparison to C-ISAM functions 1-8
 - `lseek` 1-8
 - `read` 1-8
 - `write` 1-8
- Call formats and descriptions 8-3
- Cancel a transaction, `isrollback` 5-4, 8-56
- Change a filename
 - audit trail 8-10
 - `isrename` 8-48
- Choosing an index
 - see* Selecting an in dex
- Close a file
 - audit trail 8-10
 - data file 1-28, 8-19
 - `isclose` 1-28, 8-19
 - `islogclose` 5-7, 8-38
 - transaction log file 5-7, 8-38
- Closing files
 - `iscleanup` 6-4, 8-18

D

- Data file
 - characteristics 1-31, 8-33
 - cluster 6-9, 8-20
 - organization 1-31
 - space utilization 1-32
- Data integrity
 - `bcheck` A-1
 - see also* Transaction
- Data record
 - adding 1-19
 - address of 1-5
 - customer record 3-6
 - declaration of 1-5
 - deleting 1-21
 - employee record 1-4, 1-13, 2-3, 3-3, 7-3
 - identifying a 1-18
 - in a C-ISAM file 1-4, 3-5
 - performance record 7-3
 - record layout 1-5, 3-6
 - reservation of space for 1-4, 3-6
 - summary of identification methods 1-19

- transfer to and from program 1-7
- updating 1-22
- Data representation
 - character data 3-8
 - comparison of C-ISAM to C language 3-5
 - DECIMALTYPE data 3-11
 - double-precision data 3-11
 - floating-point data 3-11
 - format-conversion 3-8
 - integer data 3-9
 - long integer data 3-9
 - machine independence 3-4
 - overview 1-6
- Data type
 - conversion functions 3-8
 - DECIMALTYPE 3-11
 - defined for keys 3-3
 - in variable-length record 3-7
 - introduction to data types 1-6
 - parameters 3-4
 - summary 3-16
 - see also* DECIMALTYPE data type
- Deadlock, definition 4-10
- decadd, syntax and use of 8-102
- deccmp, syntax and use of 8-104
- deccopy, syntax and use of 8-105
- deccvint, syntax and use of 8-93
- deccvlong, syntax and use of 8-95
- decdiv, syntax and use of 8-102
- dececv, syntax and use of 8-106
- decfcvt, syntax and use of 8-106
- DECIMALTYPE data type
 - accuracy 3-13
 - dec_t 3-11
 - defining Decimal data 3-11
 - sizing DECIMALTYPE numbers 3-12
 - see also* DECIMALTYPE functions
- DECIMALTYPE functions
 - decadd 8-102
 - deccmp 8-104
 - deccopy 8-105
 - deccvint 8-93
 - deccvlong 8-95
 - decdiv 8-102
 - dececv 8-106
 - decfcvt 8-106
 - decmul 8-102
 - decsb 8-102
 - dectoasc 8-91
 - dectodbl 8-101
 - dectoflt 8-99
 - dectoint 8-94
 - dectolong 8-97
 - decvasc 8-89
 - decvdbl 8-100
 - decvflt 8-98
 - lddecimal 3-13, 8-73
 - overview 3-14
 - stdecimal 3-13, 8-82
- decmul, syntax and use of 8-102
- decsb, syntax and use of 8-102
- dectoasc, syntax and use of 8-91
- dectodbl, syntax and use of 8-101
- dectoflt, syntax and use of 8-99
- dectoint, syntax and use of 8-94
- dectolong, syntax and use of 8-97
- decvasc, syntax and use of 8-89
- decvdbl, syntax and use of 8-100
- decvflt, syntax and use of 8-98
- dec_t
 - see* DECIMALTYPE data type 3-11
- Delete a file
 - see* Erase a file
- Delete a record
 - current record 1-22, 8-24
 - example 1-21, 7-10
 - isdelcurr 1-22, 8-24
 - isdelete 1-21, 8-25
 - isdelrec 1-22, 8-29
 - using the primary key 1-21, 8-25
 - using the record number 1-22, 8-29
- Delete an index
 - explanation 2-9
 - isdelindex 2-9, 8-27
- dictinfo
 - definition of 2-11
 - see also* Index
- Dictionary block
 - see* Index
- Dictionary format D-1
- di_nkeys variable 8-34
- di_recsz variable 8-34
- Documentation notes Intro-5
- Documentation, other useful Intro-4
- Duplicate key
 - compression 2-26

purpose 1-12
see also Key

E

End a transaction

iscommit 5-4, 8-22
isrollback 5-4, 8-56

Erase a file

audit trail 8-31
data file 6-4
iserase 6-4, 8-31
.lok lock file 8-31

Error handling

C-ISAM error codes C-1
end of file 1-26
example 4-12
in example programs 7-4
locked records 4-11
overview 1-17
record not found 1-24
return codes 1-17
use of iserrno 1-17
values for iserrno C-1

Error messages Intro-4

Example programs

build a file 7-5
chaining 7-17
random update 7-10
record definitions in 7-3
sequential processing 7-14
to add indexes 7-6
to add records 7-7
using transactions 7-22

F

fcntl() locking Intro-6, 4-10, 8-44

Field

conversion between program and
data record 1-6
declaration using pointer 1-5, 3-7
definition of 1-3
key 1-11
offset 1-4, 3-7

File

definition of 1-3
maintenance 6-9
see also Operating system files

File descriptor

returned by isbuild 1-13

returned by isopen 1-29
use of 1-13

File-level locking

explanation 4-6
see also Locking
see also Locking modes

Find a record

explanation 1-24
key value 1-10
see also Read a record
see also Selecting an index

Flush buffers, isflush 6-4, 8-32

Format

dictionary D-1

Format-conversion functions

character data 3-8
double-precision data 3-10
explanation 3-8
floating-point data 3-10
integer data 3-8
introduction to 1-6
ldchar 3-8, 8-70
lddbl 3-10, 8-71
lddblnull 3-10, 8-72
lddecimal 3-13, 8-73
ldfloat 3-10, 8-75
ldftnull 3-10, 8-76
ldint 3-8, 8-77
ldlong 3-9, 8-78
long integer data 3-8
stchar 3-8, 8-79
stdbl 3-10, 8-80
stdbnull 3-10, 8-81
stddecimal 3-13, 8-82
stfloat 3-10, 8-84
stftnull 3-10, 8-85
stint 3-8, 8-86
stlong 3-9, 8-87
see also DECIMALTYPE functions

Free-list block

see Index

Function list

format-conversion functions 8-4
functions to implement locking 8-4
functions to implement transactions
8-4
functions to manipulate
DECIMALTYPE data 3-15, 8-5
functions to manipulate files 8-3
functions to manipulate indexes 8-3
functions to manipulate records 8-3

functions, additional 8-4
Function return codes
 see Error handling

H

How to use C-ISAM 1-3

I

Identifying records
 by key value 1-18
 by record number 1-19
 current record 1-18
 summary of methods 1-19
Index
 add an index 2-8
 characteristics 2-11, 8-33
 cluster 6-9, 8-20
 current 1-26
 DECIMALTYPE data in 3-11
 defining a key for 1-15, 2-3
 definition in C-ISAM 2-4
 dictionary node 2-22
 explanation 2-9
 file organization 2-22
 file organization and variable-length
 records 1-6
 free-list node 2-23
 identify an index 2-5, 2-12
 implementation 2-13
 isaddindex 2-8, 8-8
 isdelindex 2-9, 8-27
 key description node 2-23
 maximum number of parts 8-8
 organization 2-12
 performance 2-23
 physical order 2-10
 primary 1-12, 2-10
 record number order 2-10
Indexed access, overview 1-9
Indexed sequential access method
 flexibility 1-9
 overview 1-9
Informix products
 application development tools Intro-4
isaddindex
 example 7-6
 explanation 2-8
 syntax and use of 8-8

ISAMBUFS E-2
isaudit
 command modes 6-7
 explanation 6-6
 syntax and use of 8-10
ISAUTOLOCK lock mode 8-15, 8-40
isbegin
 explanation 5-4
 syntax and use of 8-13
 see also Transaction
isbuild
 example 1-16, 7-5
 syntax and use of 8-15
iscleanup 1-30, 6-4
 syntax and use of 8-18
isclose
 caution 8-19
 syntax and use of 8-19
iscluster
 explanation 6-9
 regenerating indexes 6-10
 syntax and use of 8-20
iscommit
 explanation 5-4
 syntax and use of 8-22
 see also Transaction
ISCURR locator mode 1-26, 1-27
isdelcurr
 example 7-10
 explanation 1-22
 syntax and use of 8-24
isdelete
 example 7-10
 explanation 1-21
 syntax and use of 8-25
isdelindex
 explanation 2-9
 syntax and use of 8-27
isdelrec
 explanation 1-22
 syntax and use of 8-29
ISEQUAL locator mode 1-26, 1-27
iserase
 explanation 6-4
 syntax and use of 8-31
iserrno
 description of 1-17
 end of file 1-26
 locked records 4-11

- record not found 1-24
- use of 1-17
- values of C-1
- ISEXCLLOCK 1-16
- ISEXCLLOCK lock mode 8-15, 8-40
- ISFIRST locator mode 1-26, 1-27
- ISFIXLEN mode 8-16, 8-40
- isflush
 - explanation 6-4
 - syntax and use of 8-32
- ISGREAT locator mode 1-26, 1-27
- ISGTEQ locator mode 1-26, 1-27
- isindexinfo
 - example 1-31, 2-12
 - syntax and use of 8-33
- ISINOUT access mode 8-15, 8-32, 8-40
- ISINPUT access mode 8-15, 8-40
- ISKEEPLOCK 4-9
- ISLAST locator mode 1-26, 1-27
- ISLCKW 8-43
- islock
 - explanation 4-8
 - see also* isunlock
 - syntax and use of 8-36
- islogclose
 - explanation 5-7
 - syntax and use of 8-38
 - see also* Transaction
- islogopen
 - caution 8-39
 - explanation 5-7
 - syntax and use of 8-39
 - see also* Transaction
- ISMANULOCK lock mode 8-15, 8-36, 8-40
- ISNEXT locator mode 1-26
- ISNOLOG mode 8-16
- isopen
 - explanation 1-28
 - ISTRANS option 5-5
 - syntax and use of 8-40
 - see also* Access Modes
 - see also* Locking
 - see also* Transaction
- ISOUTPUT access mode 8-15, 8-32, 8-40
- ISPREV locator mode 1-26
- isread
 - example 1-24, 7-10, 7-14
 - explanation 1-24
 - syntax and use of 8-42
 - see also* Locking
- isreclen
 - with variable-length files 1-13
- isreclen global variable
 - opening a file 1-30
 - set by isindexinfo 8-34
- isrecnum
 - explanation 1-19
 - finding records by record number 1-28
 - set by isdelcurr 8-24
 - set by isdelete 8-25
 - set by isdelrec 8-29
 - see also* Record number
- isrecover
 - explanation 5-7
 - syntax and use of 8-46
 - see also* System Administration
 - see also* Transaction
- isrelease
 - explanation 4-9
 - syntax and use of 8-47
 - see also* Locking
- isrename
 - explanation 6-3
 - syntax and use of 8-48
- isrewcurr
 - explanation 1-23
 - syntax and use of 8-50
- isrewrec
 - explanation 1-24
 - syntax and use of 8-52
- isrewrite
 - example 7-10
 - explanation 1-23
 - syntax and use 8-54
- isrollback
 - see* Transaction
 - explanation 5-4
 - syntax and use of 8-56
 - with ISTRANS mode 8-16
- issetunique
 - explanation 6-5

 syntax and use of 8-58
 see also isuniqueid
ISSKIPLOCK 1-25
isstart
 example 7-10, 7-14, 7-17
 explanation 1-26
 syntax and use of 8-60
ISTRANS mode 8-16, 8-40
isuniqueid
 explanation 6-5
 syntax and use of 8-63
 see also issetunique
isunlock
 explanation 4-8
 syntax and use of 8-64
 see also islock
ISVARLEN mode 1-30, 8-16, 8-40
ISWAIT 8-43
iswrcurr
 example 1-20
 syntax and use of 8-65
iswrite
 example 1-20, 7-7
 syntax and use 8-67

K

Kernel locking 4-10
Key
 choice of 1-11
 compression 2-6, 2-24
 data type definition 3-3
 defining a key 1-14
 definition in C-ISAM 1-11, 2-3
 definition of Intro-3, 1-9
 descending order 2-7
 duplicate 1-12, 2-6
 flags 2-5, 2-6
 key description structure 1-11, 1-15, 2-5
 maximum size 2-7, 8-8
 number of parts 2-5, 2-7
 overview of usage 1-10
 packing density 2-23
 primary 1-12, 1-23, 2-10
 unique 1-12, 2-5, 2-6
 value 1-18
Key description block
 see Index

Key description structure
 defining a key 2-5
 definition of 2-6
 overview 1-15
 see also Key
keydesc
 defining a key 2-5
 defining a primary key 1-15
 definition of 2-6
 see also Key
keypart
 defining a key 2-5
 definition of 2-7
 see also Key
Keys in C-ISAM Files 1-10
Keyword
 definition of 1-9
 see also Key
k_flags variable 2-6
k_nparts variable 1-28, 8-16

L

ldchar
 explanation 3-8
 syntax and use of 8-70
lddbl
 explanation 3-10
 syntax and use of 8-71
lddblnull
 explanation 3-10
 syntax and use of 8-72
lddecimal
 explanation 3-13
 syntax and use of 8-73
ldfloat
 explanation 3-10
 syntax and use of 8-75
ldfltnull
 explanation 3-10
 syntax and use of 8-76
ldint
 explanation 3-8
 syntax and use of 8-77
ldlong
 explanation 3-9
 syntax and use of 8-78
Leading character compression
 explanation 2-25

see also Key

Level
see B+ tree

libisam.a library E-2

Locking
 automatic locking with isread 4-9
 by transactions 5-9
 compatibility between versions Intro-6
 concurrency control 4-3
 degree of concurrency 4-11
 during add or delete of an index 4-7
 file-level 4-6, 8-36
 implementation E-2
 islock 4-8, 8-36
 ISLOCK option in isread 4-9, 8-42
 isrelease 4-9, 8-47
 ISSKIPLOCK option in isread 8-42
 isunlock 4-8, 8-64
 manual 4-7, 4-8
 overview 4-3
 record-level 4-8, 8-42, 8-43
 single user systems 4-6
 specifying no locking 4-6
 summary 4-12
 types of 4-6
 unlock file 8-64
 unlock records 8-47
 use of 4-6
see also Locking modes

Locking modes
 automatic record locking 4-9
 exclusive file locking 1-29, 2-9, 2-10, 4-6
 ISAUTOLOCK 4-9
 ISEXCLLOCK 1-29, 2-9, 2-10, 4-6
 ISMANULOCK 1-29, 4-6, 4-7, 4-9
 ISWAIT 4-10
 manual file locking 4-7
 manual record locking 4-9
 no locking 1-29, 4-6
 summary 4-12
 waiting for locks 4-10

M

Machine notes Intro-5

Manipulating records in C-ISAM files 1-18

MAXKEYSIZE
 explanation of 8-8

N

Node
see B+ tree

NPARTS 2-5, 8-8

O

Offset, defining a field 1-4, 3-7

On-line files Intro-5

Open a file
 access modes 1-29
 audit trail 6-7, 8-10
 data file 1-28, 8-40
 file descriptor 1-29
 islogopen 5-7, 8-39
 isopen 1-28, 8-40
 ISTRANS option 5-5
 locking mode 1-29, 4-6
 maximum number of open files 1-30
 of variable-length records 1-30
 transaction log file 5-7, 8-39
 with isbuild 1-16

Operating system files
 .dat 1-13
 .idx 1-13

Organization of C-ISAM files
 data file 1-32
 index file 2-22
 overview 1-13

P

Performance
 key compression 2-25
 key size 2-23, 2-24
 multiple indexes 2-27
 tree height 2-23

Physical order index 2-10

Pointer
 definition of 1-9
see also B+ tree

Pointer arithmetic, to define a field 1-5

Primary index
see Index

Primary key
see Key

Programs
see Example programs

R

Read a record
 automatic locking 4-9
 by record number 1-28, 8-43
 example 1-24, 7-17
 explanation 1-24
 isread 1-24, 8-42
 locking option 4-9, 8-42, 8-43
 retrieval modes, *see* Search modes
 search modes 1-25, 8-42
 summary of search modes 1-26
 see also isread

Record
 definition of 1-3
 length 1-4
 variable-length 1-4
 see also Data record

Record number
 definition of 1-19
 example of retrieval by 1-28
 isrecnum 1-19
 retrieval by 1-28, 2-10
 see also Data record

Record-level locking
 explanation 4-8
 see also Locking
 see also Locking modes

Recover transactions, isrecover 8-46

Recovery
 caution 5-7
 explanation 5-7
 rollforward 5-8
 transaction 8-46
 see also System administration
 see also Transaction

Release notes Intro-5

Release record locks
 see isrelease

Remove a file
 see Erase a file

Remove an index
 see isdelindex

Rename a file, isrename 6-3, 8-48

Reorganization
 data file 1-32
 index 2-21

Representation of data
 see Data representation

RESETLOCK environment variable
 description of Intro-6

Retrieval by record number
 using isread 1-28, 8-43
 using isstart 1-28, 8-61

Return codes
 see Error handling

Rewrite a record
 by record number 1-24, 8-52
 current record 1-23, 8-50
 example 1-23, 7-10
 identified by primary key 1-23, 8-54
 isrewcurr 1-23, 8-50
 isrewrec 1-24, 8-52
 isrewrite 1-23, 8-54

Roll back a transaction, isrollback 5-4, 8-56

Rollforward recovery
 see Recovery

Root
 see B+ tree

S

Selecting an index
 example 7-14
 explanation 1-26
 isstart 1-26, 8-60
 retrieval by record number 1-28
 starting position within 1-27, 8-60, 8-61

Sequential access
 example 1-25, 7-14
 overview 1-9
 see also isread
 see also isstart

Skipping locked records 1-25

stchar
 explanation 3-8
 syntax and use of 8-79

stdbl
 explanation 3-10
 syntax and use of 8-80

stdbnull
 explanation 3-10
 syntax and use of 8-81

stdecimal
 explanation 3-13
 syntax and use of 8-82

-
- stfloat
 - explanation 3-10
 - syntax and use of 8-84
 - stfltnull
 - explanation 3-10
 - syntax and use of 8-85
 - stint
 - explanation 3-8
 - syntax and use of 8-86
 - stlong
 - explanation 3-9
 - syntax and use of 8-87
 - System administration
 - files E-1
 - installation E-1
 - ISAMBUFS parameter E-2
 - locking E-2
 - transaction logging and recovery E-3
- ## T
- Trailing space compression
 - explanation 2-26
 - see also* Key
 - Trailing spaces 1-8
 - Transaction
 - begin 5-4, 8-13
 - cancel 5-4, 5-6, 8-56
 - caution during recovery 5-7
 - close log file 5-7, 8-38
 - commit 5-4, 8-22
 - concurrency control 5-10
 - concurrent execution of 5-8
 - create log file 5-7
 - data integrity 5-8
 - definition of 5-3
 - example 5-5, 7-22
 - implementing 5-4
 - isaddindex in 8-8
 - isaudit in 8-11
 - isbegin 5-4, 8-13
 - isbuild 8-16
 - iscommit 5-4, 8-22
 - islogclose 5-7, 8-38
 - islogopen 5-7, 8-39
 - isrecover 5-7, 8-46, E-3
 - isrollback 5-4, 8-56
 - ISTRANS option in isopen 5-5
 - locking 5-9
 - logging 5-7, E-3
 - management services 5-4
 - open log file 5-7, 8-39
 - purpose of 5-3
 - recoverable, rollback disallowed 5-6
 - recovery 5-7, 8-46, E-3
 - rollforward recovery 5-8
 - summary 5-11
 - unit of work 5-4
 - with variable-length records 5-6
 - see also* Recovery
 - Transaction management support
 - routines 5-3
 - Transfer of data, between program and C-ISAM record 1-7
 - Typographical conventions Intro-5
- ## U
- Unique identifier
 - explanation 6-5
 - issetunique 6-5, 8-58
 - isuniqueid 6-5, 8-63
 - Unique key
 - purpose 1-12
 - see also* Key
 - Unlock file
 - see* Locking
 - Unlock records
 - see* Locking
 - Update a record
 - example 7-10
 - explanation 1-22
 - see also* Rewrite a record
- ## V
- Variable-length records
 - building a file 1-13
 - data corruption 6-9
 - in transactions 5-6
 - programming with 1-5
 - with audit trails 6-7
- ## W
- Waiting for locks 8-43, 8-44
 - What is a C-ISAM file? 1-3
 - Why use transaction management? 5-3
 - Write a record
 - current record 1-20
 - example 7-7

iswcurr 1-20, 8-65
iswrite 1-20, 8-67

X

X/Open compatibility 4-10
