

C-ISAM Indexed Sequential Access Method

Programmer's Manual

Version 7.2
October 2001
Part No. 000-7897B

Note:
Before using this information and the product it supports, read the information in the appendix entitled "Notices."

This document contains proprietary information of IBM. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this manual should not be interpreted as such.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1996, 2001. All rights reserved.

US Government User Restricted Rights—Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Table of Contents

Introduction

About This Manual	4
Purpose of This Manual	4
Organization of This Manual	4
Types of Users	5
New Features of This Product	6
Conventions	6
Typographical Conventions	7
Icon Conventions	8
Command-Line Conventions	9
Sample-Code Conventions	12
Additional Documentation	12
Printed Documentation	13
On-Line Documentation	14
Related Reading	15
Compliance with Industry Standards	15
Informix Welcomes Your Comments	15

Chapter 1

How to Use C-ISAM

What Is a C-ISAM File?	1-3
Data Records in C-ISAM Files	1-5
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	1-11
Organization of C-ISAM Files	1-14
Building a C-ISAM File	1-14
C-ISAM Error Handling	1-19

Manipulating Records in C-ISAM Files	1-20
Identifying Records	1-20
Adding Records	1-22
Deleting Records	1-24
Updating Records	1-25
Finding Records	1-27
Opening and Closing Files	1-32
Opening a File in Exclusive Mode	1-33
Opening a Variable-Length File	1-33
Maximum Number of Open Files	1-34
Closing Fixed- and Variable-Length Files	1-34
Compiling Your C-ISAM Program	1-35
Running Your C-ISAM Program	1-36
Setting the INFORMIXDIR Environment Variable	1-36
C-ISAM Data-File Structure	1-37
Summary	1-38

Chapter 2

Indexing

Defining an Index	2-3
Key Structures	2-6
Manipulating Indexes	2-8
Adding Indexes	2-9
Deleting Indexes	2-10
Defining Record-Number Sequence	2-11
Determining Index Structures	2-12
B+ Tree Organization	2-14
Searching for a Record	2-17
Adding Keys	2-18
Removing Keys	2-22
Index-File Structure	2-23
Performance Considerations	2-24
Key Size and Tree Height	2-24
Key Compression	2-26
Multiple Indexes	2-30
Localized Indexes	2-31
Summary	2-31

Chapter 3	Data Types	
	Defining Data Types for Keys	3-3
	C-ISAM Computer-Independent Data Types	3-5
	Defining Data Records	3-7
	Data Types in Variable-Length Records	3-8
	C-ISAM Data Type Conversion Routines	3-9
	DECIMALTYPE Data Type	3-13
	Using DECIMALTYPE Data Type Numbers	3-13
	Summary	3-19
Chapter 4	Locking	
	Concurrency Control	4-3
	Types of Locking	4-7
	File-Level Locking	4-7
	Record-Level Locking	4-10
	Increasing Concurrency	4-12
	Error Handling	4-13
	Summary	4-14
Chapter 5	Transaction Management Support Routines	
	Why Use Transaction Management?	5-3
	Transaction-Management Services	5-4
	Implementing Transactions	5-4
	Transactions with Variable-Length Records	5-7
	Logging and Recovery	5-7
	Data Integrity	5-9
	Concurrent Execution of Transactions	5-9
	Summary	5-12
Chapter 6	Additional Facilities	
	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	6-10
	File Maintenance with Variable-Length Records	6-11
	If Data Files Are Corrupted.	6-11
	If Index Files Are Corrupted	6-11
	Summary	6-14

Chapter 7	Sample Programs Using C-ISAM Files	
	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	7-8
	Random Update	7-11
	Sequential Access	7-16
	Chaining	7-19
	Using Transactions	7-25
	Summary	7-29

Chapter 8	Call Formats and Descriptions	
	Functions for C-ISAM File Manipulation	8-9
	isaddindex	8-10
	isaudit	8-12
	isbegin	8-14
	isbuild	8-16
	iscleanup	8-19
	isclose	8-20
	iscluster	8-21
	iscommit	8-23
	isdelcurr	8-25
	isdelete	8-26
	isdelindex	8-28
	isdelrec	8-30
	iserase	8-31
	isflush	8-32
	isglsversion	8-33
	isindexinfo	8-35
	islanginfo	8-38
	islock	8-39
	islogclose	8-40
	islogopen	8-41
	isnlsversion	8-42
	isopen	8-44
	isread	8-46
	isrecover	8-50
	isrelease	8-51
	isrename	8-52
	isrewcurr	8-53
	isrewrec	8-55
	isrewrite	8-56
	isrollback	8-58
	issetunique	8-60

isstart	8-61
isuniqueid	8-64
isunlock	8-65
iswrcurr	8-66
iswrite	8-68
Format-Conversion and Manipulation Functions	8-70
ldchar	8-71
lddbl	8-72
lddblnull	8-73
lddecimal	8-74
ldfloat	8-76
ldftnull	8-77
ldint	8-78
ldlong	8-79
stchar	8-80
stdbl	8-81
stdbnull	8-82
stdecimal	8-83
stfloat	8-85
stftnull	8-86
stint	8-87
stlong	8-88
DECIMALTYPE Functions	8-89
decadd, decsub, decmul, and decdiv	8-90
deccmp	8-92
deccopy	8-93
deccvasc	8-94
deccvdbl	8-96
deccvfit	8-97
deccvint	8-98
deccvlong	8-99
dececv and decfcvt	8-100
dectoasc	8-102
dectodbl	8-104
dectofit	8-105
dectoint	8-106
dectolong	8-107
Summary	8-108

Appendix A	C-ISAM Utilities
Appendix B	The GLS Environment
Appendix C	Error Codes
Appendix D	File Formats
Appendix E	System Administration
Appendix F	Header Files
Appendix G	Notices
	Index

Introduction

About This Manual	4
Purpose of This Manual	4
Organization of This Manual	4
Types of Users	5
New Features of This Product	6
Conventions	6
Typographical Conventions	7
Icon Conventions	8
Comment Icons	8
Compliance Icons	8
Command-Line Conventions	9
Sample-Code Conventions	12
Additional Documentation	12
Printed Documentation	13
On-Line Documentation	14
Release Notes, Documentation Notes, Machine Notes	14
Related Reading	15
Compliance with Industry Standards	15
Informix Welcomes Your Comments	15

T

his chapter introduces the *C-ISAM Programmer's Manual*. Read this chapter for an overview of the information provided in this manual and for an understanding of the conventions used throughout this manual.

C-ISAM is an Indexed Sequential Access Method that is defined and implemented for the C language by Informix. 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

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.

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.

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.

About This Manual

The *C-ISAM Programmer's Manual* describes the C-ISAM functions and facilities. This manual assumes that you are familiar with the C-programming language.

Purpose of This Manual

This manual provides information about the C-ISAM library of C functions that you can use to create and manipulate indexed files.

Organization of This Manual

This manual includes the following chapters:

- This Introduction describes C-ISAM, explains how to use this manual, provides an overview of the manual, and describes the documentation conventions used.

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

- Chapter 1, "How to Use C-ISAM," explains how to create and manipulate C-ISAM files.
- Chapter 2, "Indexing," explains the organization and use of the indexes.
- Chapter 3, "Data Types," describes the data types that you can use in C-ISAM files and how they are handled in C-ISAM programs.

Chapters 4, 5, and 6 explain specialized features.

- Chapter 4, "Locking," describes file and record locking and how these are implemented.
- Chapter 5, "Transaction Management Support Routines," explains how to ensure data integrity using transaction management.
- Chapter 6, "Additional Facilities," 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, “Sample Programs Using C-ISAM Files,” contains several complete programs that use the C-ISAM functions described in earlier chapters.
- Chapter 8, “Call Formats and Descriptions,” 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.
- Appendix B describes the GLS environment and explains how to set a GLS locale to specify a collation sequence for a C-ISAM file.
- 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 calls.
- Appendix E explains how to set up your system to use C-ISAM.
- Appendix F contains the source code for the header files you need to include in C-ISAM programs.

Types of Users

This manual is written for all C-ISAM users and assumes familiarity with the C programming language and the standard C library of functions related to files and Input/Output operations.

GLS

New Features of This Product

The Introduction to each Version 7.2 product manual contains a list of new features for that product. A comprehensive list of all of the new features for Version 7.2 Informix products is in the Release Notes file called **SERVERS_7.2**.

This section highlights the major new features implemented in Version 7.2 of C-ISAM.

The Global Language Support (GLS) feature lets Informix Version 7.2 products handle different languages, collation sequences, and code sets. C-ISAM uses the GLS feature to support localized collation on index files. Localized collation involves the sorting of character data according to the order of characters in a real language.

GLS functionality supersedes the functionality of Native Language Support (NLS) and eliminates the need to distinguish between internationalized versions of Informix software. ♦

Conventions

This section describes the conventions that are used in this manual. By becoming familiar with these conventions, you will find it easier to gather information from this and other volumes in the documentation set.

The following conventions are covered:

- Typographical conventions
- Icon conventions
- Command-line conventions
- Sample-code conventions

Typographical Conventions

This 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 this manual.




Convention	Meaning
<i>italics</i>	Within text, new terms and emphasized words are printed in italics. Within syntax diagrams, values that you are to specify are printed in italics.
boldface	Identifiers (names of classes, objects, constants, events, functions, program variables, forms, labels, and reports), environment variables, database names, table names, column names, menu items, command names, and other similar terms are printed in boldface.
<code>monospace</code>	Information that the product displays and information that you enter are printed in a monospace typeface.
KEYWORD	All keywords appear in uppercase letters.
◆	This symbol indicates the end of product- or platform-specific information.

Icon Conventions

Throughout the documentation, you will find text that is identified by several different types of icons. This section describes these icons.



Comment Icons

Comment icons identify three types of information, as described in the following table. This information is always displayed in *italics*.

Icon	Description
	Identifies paragraphs that contain vital instructions, cautions, or critical information.
	Identifies paragraphs that contain significant information about the feature or operation that is being described.
	Identifies paragraphs that offer additional details or shortcuts for the functionality that is being described.

Compliance Icons

Compliance icons indicate paragraphs that provide guidelines for complying with a standard.

Icon	Description
	Identifies information that is specific to an application that uses Global Language Support.
	Identifies information that is specific to an application that uses an NLS index.

Command-Line Conventions

C-ISAM supports command-line options. You enter these commands at the operating-system prompt to perform certain functions with C-ISAM. Each valid command-line option is illustrated in a diagram in Appendix A.

This section defines and illustrates the format of the commands that are available in C-ISAM and other Informix products. These commands have their own conventions, which might include alternative forms of a command, required and optional parts of the command, and so forth.

Each diagram displays the sequences of required and optional elements that are valid in a command. A diagram begins at the upper left with a command. It ends at the upper right with a vertical line. Between these points, you can trace any path that does not stop or back up. Each path describes a valid form of the command. You must supply a value for words that are in italics.

You might encounter one or more of the following elements on a command-line path.

Element	Description
command	This required element is usually the product name or other short word that invokes the product or calls the compiler or preprocessor script for a compiled Informix product. It might appear alone or precede one or more options. You must spell a command exactly as shown and must use lowercase letters.
<i>variable</i>	A word in italics represents a value that you must supply, such as a database, file, or program name. A table following the diagram explains the value.
-flag	A flag is usually an abbreviation for a function, menu, or option name or for a compiler or preprocessor argument. You must enter a flag exactly as shown, including the preceding hyphen.
.ext	A filename extension, such as .sql or .cob , might follow a variable that represents a filename. Type this extension exactly as shown, immediately after the name of the file and a period. The extension might be optional in certain products.

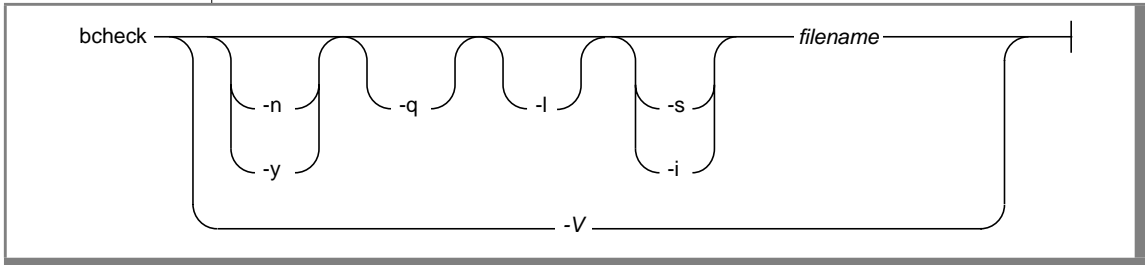
(1 of 2)

Element	Description
(.,;+*-/)	Punctuation and mathematical notations are literal symbols that you must enter exactly as shown.
' '	Single quotes are literal symbols that you must enter as shown.
<div>Privileges p. 5-17</div> <div>Privileges</div>	A reference in a box represents a subdiagram on the same page (if no page is supplied) or another page. Imagine that the subdiagram is spliced into the main diagram at this point.
— ALL —	A shaded option is the default. If you do not explicitly type the option, it will be in effect unless you choose another option.
→ →	Syntax enclosed in a pair of arrows indicates that this is a subdiagram.
—	The vertical line is a terminator and indicates that the statement is complete.
<div>IN</div> <div>NOT</div>	A branch below the main line indicates an option
<div>,</div> <div>variable</div>	A loop indicates a path that you can repeat. Punctuation along the top of the loop indicates the separator symbol for list items, as in this example.
<div>,</div> <div>3 size</div>	A gate ($\sqrt{3}$) on a path indicates that you can only use that path the indicated number of times, even if it is part of a larger loop. Here you can specify <i>size</i> no more than three times within this statement segment.

(2 of 2)

Figure 1 shows the flow of the **bcheck** utility command.

Figure 1
Elements of a Command-Line Diagram



To construct a correct command, start at the top left with the command **bcheck**. Then follow the diagram to the right, including the elements that you want. The elements in the diagram are case sensitive.

To read the example command-line diagram

1. Type the word **bcheck**.
2. Choose either the lower path or the upper path.
3. When you choose the lower path, you must type **-V**. That brings you to the terminator. That completes the **bcheck** command. Press RETURN to execute the command.
4. When you choose the upper path, take the following steps:
 - a. You can choose **-n** or **-y**, but not both.
 - b. You can choose **-q**.
 - c. You can choose **-l**.
 - d. You can choose **-s**.
 - e. You must supply a filename.

After you choose *filename*, you come to the terminator. Your command is complete.
5. Press ENTER to execute the command.

Sample-Code Conventions

Examples of C code occur throughout this manual. Except where noted, the code is specific to C-ISAM. For instance, you might see the code in the following example:

```
#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;
.
.
.
```

Dots in the example indicate that more code would be added in a full application, but it is not necessary to show it to describe the concept being discussed.

Additional Documentation

The C-ISAM documentation set includes printed manuals and on-line manuals.

This section describes the following pieces of the documentation set:

- Printed documentation
- On-line documentation
- Related reading

Printed Documentation

The following printed manuals are included in the *C-ISAM Programmer's Manual* documentation set:

- ***UNIX Products Installation Guide***. This guide contains instructions for installing Informix products on computers running the UNIX operating system. Keep it with your Informix software documentation for easy reference.
- ***Informix Error Messages***. When errors occur, you can look them up by number and learn their cause and solution in the *Informix Error Messages* manual. You can also look up the error messages in Appendix C of this manual.

On-Line Documentation

C-ISAM provides on-line documentation for release notes, documentation notes, and machine notes.

Release Notes, Documentation Notes, Machine Notes

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 C-ISAM sample programs, might supplement the information in this manual.

On-Line File	Purpose
Documentation notes	Describes features that are not covered in the manuals or that have been modified since publication. The file containing the Documentation Notes for C-ISAM is called ISAMDOC_7.2 .
Release notes	Describes feature differences from earlier versions of Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds. The file containing the Release Notes for Version 7.2 of Informix database server products is called SERVERS_7.2 .
Machine notes	Describes any special actions that are required to configure and use Informix products on your computer. Machine notes are named for the product that is described. Machine notes are named for the product described, for example, the Machine Notes file for C-ISAM is ISAM_7.2 .

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

Related Reading

If you have limited UNIX system experience, consult your operating system manual or a good introductory text before you read this manual. The following texts provide a good introduction to UNIX systems:

- *Introducing the UNIX System* by H. McGilton and R. Morgan (McGraw-Hill Book Company, 1983)
- *Learning the UNIX Operating System*, by G. Todino, J. Strang, and J. Peek (O'Reilly & Associates, 1993)
- *A Practical Guide to the UNIX System*, by M. Sobell (Benjamin/Cummings Publishing, 1989)
- *UNIX for People* by P. Birns, P. Brown, and J. Muster (Prentice-Hall, 1985)
- *UNIX System V: A Practical Guide* by M. Sobell (Benjamin/Cummings Publishing, 1995)

Compliance with Industry Standards

C-ISAM is built to conform to the guidelines put forth in the *X/Open Portability Guide, Issue 3 (XPG3)*. The GLS feature for Informix Version 7.2 products is based on the X/Open XPG4 specifications.

Informix Welcomes Your Comments

Please let us know what you like or dislike about our manuals. To help us with future versions of our manuals, please tell us about any corrections or clarifications that you would find useful. Write to us at the following address:

Informix Software, Inc.
SCT Technical Publications Department
4100 Bohannon Drive
Menlo Park, CA 94025

If you prefer to send electronic mail, our address is:

`doc@informix.com`

Or, send a facsimile to the Informix Technical Publications Department at:

415-926-6571

Please include the following information:

- The name and version of the manual that you are using
- Any comments that you have about the manual
- Your name, address, and phone number

We appreciate your feedback.

How to Use C-ISAM

What Is a C-ISAM File?	1-3
Data Records in C-ISAM Files.	1-5
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
Indexed Access	1-9
Sequential Access.	1-9
Flexibility	1-10
Keys in C-ISAM Files	1-10
Using Keys	1-11
Choosing a Key	1-11
Key Descriptions	1-12
Unique and Duplicate Keys	1-12
Primary Keys	1-13
Collation Sequences of Keys	1-13
Organization of C-ISAM Files	1-14
Building a C-ISAM File	1-14
Building a File With Fixed-Length Records	1-15
Building a Variable-Length File	1-19
C-ISAM Error Handling	1-19
Manipulating Records in C-ISAM Files	1-20
Identifying Records	1-20
Using the Key Value.	1-20
Using the Current Record	1-21
Using the Record Number.	1-21
Summary of Record Identification Methods	1-22
Adding Records	1-22

Deleting Records	1-24
Updating Records	1-25
Finding Records	1-27
Using the isstart Function	1-29
Finding Records by Record Number	1-31
Opening and Closing Files	1-32
Opening a File in Exclusive Mode	1-33
Opening a Variable-Length File	1-33
Maximum Number of Open Files	1-34
Closing Fixed- and Variable-Length Files	1-34
Compiling Your C-ISAM Program.	1-35
Running Your C-ISAM Program	1-36
Setting the INFORMIXDIR Environment Variable	1-36
C-ISAM Data-File Structure	1-37
Summary	1-38

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 and run your program and describes 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 store in your computer. For example, you might 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 store is called a *field*.

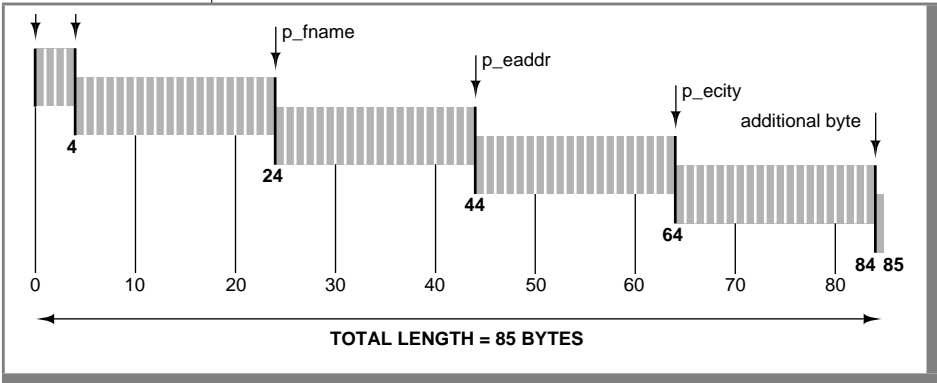
You might decide to keep an employee number, the first and last names, address, and city for each employee. This collection of fields is called a *record*. You must determine the data type and the length of each field.

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.

Figure 1-1
Employee Record

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

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 1 additional byte for a null character that indicates the end of the record. The easiest way to do this is to declare a character variable of the size that your record layout indicates plus 1 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 4-byte pointer to the variable-length portion of the record. The variable-length portion of the record is stored in the index file.

Warning: *It is important that you do not remove the index files (.idx) associated with variable-length records. When you remove the .idx files, no way exists of restoring the files and the variable-length data contained within them, other than restoring them from a backup.*

When 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 “File Maintenance with Variable-Length Records” on page 6-11.

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 computer. C-ISAM representation of these data types, however, is computer-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 that you need in programs that use C-ISAM, are in the header file **isam.h** that you must include in your programs. Appendix B contains a listing of **isam.h**.)

C-ISAM provides functions to convert between the internal representation of data on your computer and the way that C-ISAM stores the data. (See Figure 1-3 on page 1-7.) 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 computer and the computer-independent C-ISAM representation of the data. Chapter 3, “Data Types,” describes the conversion functions that you can use.

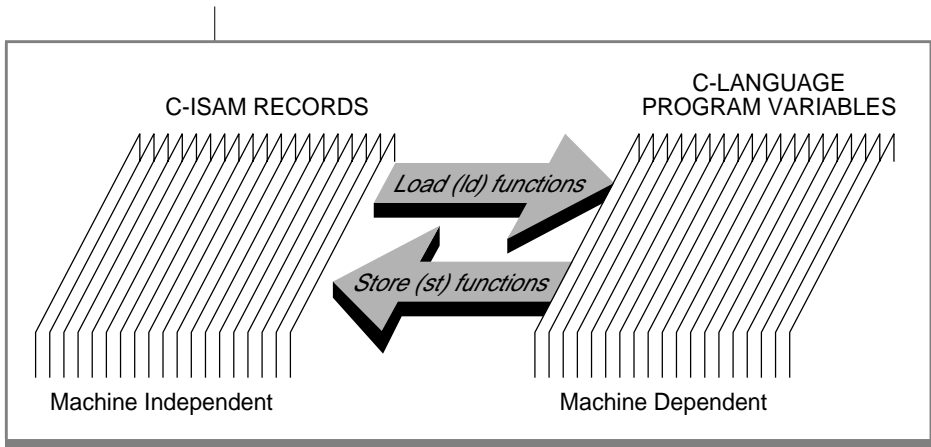


Figure 1-3
*Converting the
 Internal
 Representation of
 Data to the C-ISAM
 Representation of
 Data*

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.

Figure 1-4
Transferring Data Between Program Variables and a C-ISAM Data Record

```
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);
```

The function **stlong** takes the long integer **empno**, converts it into the C-ISAM computer-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 on page 1-4. The function 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 that is 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 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. When 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. When you add a section to a book, rearrange paragraphs or sections, remove a few pages, you must re-create your index because 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. When 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 “Indexed Access” on page 1-9, an entry in the index for this book is a keyword. With each keyword a pointer exists 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 might want to access records by employee number. This task 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. When 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. When you provide a search value of 100 and use the Employee Number index, the C-ISAM function locates the record that corresponds to Employee Number 100, regardless of where it is in the file.

Choosing a Key

You might also need to find specific records in the **employee** file by employee name. Once again, this task requires an index. The choice of the key, in this case, is a little more complex because 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 10 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. When 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 on page 1-4.

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 might 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. When you want to find or modify the record that belongs to a specific employee, this unique field saves the trouble of determining whether you have the correct record.

When 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 that you want. When it is not the correct one, you must search again to find others.

When you know that you have a unique field within your records, you can include this information in the key description. Then C-ISAM allows 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.

Sometimes you do not want to specify a key as unique. When you want an index on Employee Name, you might want to allow for duplicate keys in case two or more employees have the same name, for example, two R. Smith. When 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 more than one exists.

Primary Keys

When you create your C-ISAM file, you ordinarily specify the key and its description in the index. The keys in this index are called *primary keys*. This index is the *primary index*. Other nonprimary 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 “Manipulating Records in C-ISAM Files” on page 1-20.

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 discusses this option.

Collation Sequences of Keys

Each key has a *collation sequence*—the sequence in which the data is ordered. The default locale, U.S. English with the 8859-1 code set, specifies the collation sequence. When you do not want your index to use the default collation sequence, you can specify a different collation sequence. Indexes built with specific collation sequences are called *localized indexes*. When you are using a character field in the key that is built on a specific collation sequence, you need to declare the field as NCHARTYPE data type.

Once you have built an index with a specific collation sequence, you must always use that collation sequence when you access the data. All the indexes associated with a data file must use the same collation sequence.

For information about building and using localized indexes, see “Creating Localized Indexes” on page B-8. ♦

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 two indexes exist for the employee file, one on Employee Number and the other on Employee Name, only one data record exists 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 can build files with either fixed-length records or variable-length records. In either case, you use the **isbuild** function call to create a C-ISAM file. The process of building a file that contains variable-length records is very similar to the process of building files with fixed-length records. In the two sections that follow, both processes are described in detail.

Building a File With Fixed-Length Records

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

```
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 you can use to identify the C-ISAM file in other function calls.

The first argument of the function is the C-ISAM filename. 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.

Figure 1-5
Employee Record

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		

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 following parameters:

employee	is the name of the file that is being built, and the first parameter.
84	is the record size (not including the additional null terminator), 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 because 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 F for a complete listing of **isam.h**.

Figure 1-6
Key-Description Structure

```
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 */
};
```


You must declare and initialize a **keydesc** structure to define your key. At this point, consider only what is necessary to define the primary index that contains 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 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 on the lengths of the fields that precede the key field in the record. Because 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. Because 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.

When 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.

Figure 1-7
Creating a C-ISAM File

```
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);
}
.
.
```

The function returns a code. When 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. A return code that is less than zero indicates an error.

The file-opening modes are discussed in “Opening and Closing Files” on page 1-32. 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.

To build a 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 contains variable-length records.
3. Assign the second parameter the maximum length of the record, including the fixed- and variable-length parts. The smallest value you can use in ISRECLEN is 1.

The smallest variable-length record that you can use is 2 bytes; 1 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 1,284 bytes, a minimum record size of 84 bytes, and a variable-length portion of up to 1,200 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 8-16.

C-ISAM Error Handling

C-ISAM functions return an integer code. When this code is greater than or equal to zero, the function executed successfully. When 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 contains a description of all error conditions, their values, and mnemonics.

Figure 1-7 shows an example of the use of the **iserrno** variable. Check the return code of each C-ISAM call and take appropriate action based on 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 might 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. When 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, because Employee Number is the unique primary key. (See “Building a C-ISAM File” on page 1-14, for an example of how to build the **employee** file.)

When 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.

GLS

When your application uses an index with a specific collation sequence to find a record, you must set your environment to the locale that specifies the collation sequence of the index. If the collation sequence of the application accessing records differs from the index-collation sequence, the function called to locate the record (such as **isread**) fails. For a complete description of Global Language Support (GLS) locales, collation order, and localized indexes, refer to Appendix B. ♦

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 because the last record that you read becomes the *current record*.

When 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.

Method	Description
Key value	Uses an index to access the record.
Current record	Is either the last record read or, in certain cases that are discussed in the following sections, is set by another function.
Record number	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 assign values to the elements of your data-record structure with the data to be written to the file. When 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 is associated with your file.

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.

Figure 1-8
Adding Records to a C-ISAM File

```
#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);
    .
    .
    .

```

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: **isdelete**, **isdelcurr**, or **isdelrec**. All of these functions 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.

Figure 1-9
Deletion Using the Primary Key

```
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);
```

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.

The integer that receives the return code is **cc**. 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: **isrewrite**, **isrewcurr**, or **isrewrec**.

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, as shown in the following example:

```
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. The following example of a call rewrites the 404th record from the beginning of the file:

```
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 that belongs 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 that is 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.

When **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**. When 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. When the value of **iserrno** is **ENOREC**, a record matching the key cannot be found.

When **isread** finds a locked record, the current record pointer and the contents of the global variable **isrecnum** remain unchanged from the last **isread** call. When 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 can 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.

When 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. When you want 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.

When 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 ISCURRE (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 in the following list:

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 no exact match exists, 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. When 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 in “Building a C-ISAM File” on page 1-14 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. When 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 ISCURRE (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 finish using the file, close it with **isclose** as shown in the following example:

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

In this example, **fd** is the file descriptor that was returned when **isbuild** created the file.

When you close a C-ISAM file and want to use it again, you must open it with **isopen**. The following statement opens the file 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 . When 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 if you want to perform locking under the control of your program.

Figure 1-14 shows all the allowable access modes.

Figure 1-14
Access Modes for isopen and isbuild

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

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);
```

Refer to 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. When 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 8-44.

When 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. When 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;
/* Try opening file as FIXLEN */
fd = isopen(employee, mode);
if (fd < 0)
{
    mode += ISVARLEN;
    /*Try opening file as VARLEN */
    fd = isopen(employee, mode);
    if (fd < 0)
    {
        printf ("isopen failed");/* Open really failed */
        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 255 C-ISAM files open at any one time. An operating-system limit on the number of open files, however, might 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. When your program uses the DECIMALTYPE data type (see Chapter 3), you must also include the **decimal.h** header file. (Refer to Appendix F 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 identifies the files that are necessary to compile your programs.)

To compile your C-ISAM program, use a command line like the following example. This example assumes that you have used the default directories when you installed the C-ISAM files.

```
cc bld_file.c -lisam -o bld_file
```

When you use the **lint** utility, specify the C-ISAM library as shown in the following example:

```
lint bld_file.c -lisam
```

GLS

Running Your C-ISAM Program

Before you run a C-ISAM program, set the **INFORMIXDIR** environment variable to the directory where you have installed C-ISAM. If you do not set **INFORMIXDIR**, your program cannot access the GLS locale files that it uses for locale-sensitive processing. For a full description of the GLS environment, see Appendix B. ♦

Setting the INFORMIXDIR Environment Variable

The **INFORMIXDIR** environment variable specifies the directory that contains C-ISAM product files.

```
setenv INFORMIXDIR pathname
```

Element	Purpose	Key Considerations
<i>pathname</i>	Specifies the pathname of the directory where C-ISAM product files are located.	None.

The installation script that you use to install C-ISAM places the product files in either the default directory or a directory that you specify.

If you choose the default installation directory to install C-ISAM, set the **INFORMIXDIR** environment variable to **/usr** before you run a program. The following **setenv** command sets **INFORMIXDIR** to match the default installation directory:

```
setenv INFORMIXDIR /usr
```

If you choose a custom installation directory to install C-ISAM, make sure that **INFORMIXDIR** is set to that directory.

C-ISAM Data-File Structure

The file that contains 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 “Determining Index Structures” on page 2-12.

The data record has a 1-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 explains 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.
- Run a C-ISAM program.
- Determine the record length and number of records in a C-ISAM file.

Indexing

Defining an Index	2-3
Key Structures	2-6
Manipulating Indexes	2-8
Adding Indexes	2-9
Deleting Indexes	2-10
Defining Record-Number Sequence.	2-11
Determining Index Structures.	2-12
B+ Tree Organization	2-14
Searching for a Record	2-17
Adding Keys	2-18
Removing Keys.	2-22
Index-File Structure	2-23
Performance Considerations	2-24
Key Size and Tree Height	2-24
Key Compression	2-26
Leading-Character Compression	2-27
Trailing-Space Compression	2-28
Duplicate-Key and Maximum Compression	2-29
Multiple Indexes	2-30
Localized Indexes	2-31
Summary	2-31

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,” introduced you to C-ISAM files and keys, and showed 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 define indexes only for the fixed-length portion of a record. When you define indexes for the fixed-length portion of variable-length records, you follow the same procedure as for standard fixed-length records.

Figure 2-1
Employee Record

Description	Type	Length	Pointer	Offset in 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 2-2
Employee Record Illustration

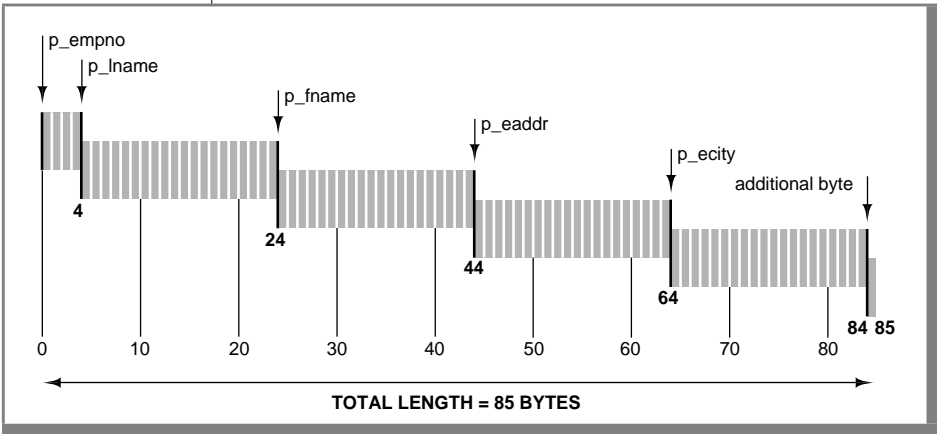


Figure 2-3 shows the code used to build this file.

```
#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;
.
.
.
    struct keydesc.key;
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 **keypart** 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 instance of the structure **keydesc**. Figure 2-4 shows this structure. You must use a separate instance 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 “Using the isstart Function” on page 1-29).

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 nested in **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; you cannot change it, but it might be different for different operating systems.

Because 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 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, because this part of the key starts at offset zero from the beginning of the record. You set **k_part[0].kp_leng** to LONGSIZE because this is the length of the data type in bytes. You set **k_part[0].kp_type** to LONGTYPE because 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 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 shown in the following list:

k_flags	sets one or more of the following flags that you can use 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.

“Key Compression” on page 2-26 describes compression techniques.

When you use two or more flags, add them together, as shown in the following example:

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

This example 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 “Defining Record-Number Sequence” on page 2-11.) 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 shown in the following example:

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

Manipulating Indexes

When you create a C-ISAM 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 nonprimary 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 because 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 because the **ISDESC** parameter is not added to either **kp_type**.

The lengths that you assign to the **nkey.k_part[0].kp_leng** and **nkey.k_part[1].kp_leng** attributes 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. You can do this by calling the **isopen** function as shown in Figure 2-5.

GLS

For information about adding an index that uses a specific collation sequence, see “Creating Localized Indexes” on page B-8. ♦

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 by passing the **ISEXCLLOCK** parameter to **isopen**. 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 might want to find records based on the relative location of the records from the beginning of the file. As explained in “Finding Records by Record Number” on page 1-31, 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 setting means that no primary key exists, and whenever you open the file, the record number defines the key order. When the file has other indexes, you can change the index by calling **isstart** with the appropriate **keydesc** structure.

You have 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 to this structure:

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

The **dictinfo** structure is defined in “The isam.h Header File” on page F-1. 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 shown in the following list:

- | | |
|--------------------|---|
| 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 “B+ Tree Organization” on page 2-14.) |
| di_nrecords | This field contains the number of data records in the file. |

The **isindexinfo** function also sets the global variable **isreclen** (defined in **isam.h**) 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** for 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 might change. C-ISAM functions use a key description, not an index number, to identify the index.

For information about using the **islanginfo** function to determine the locale that is associated with a localized index, see “Determining Index-Collation Sequence” on page B-11. ♦

GLS

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. 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 *node* is an ordered group of key values having a fixed number of elements. As shown in Figure 2-9, a node can contain space reserved for key values that have no value assigned.

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.

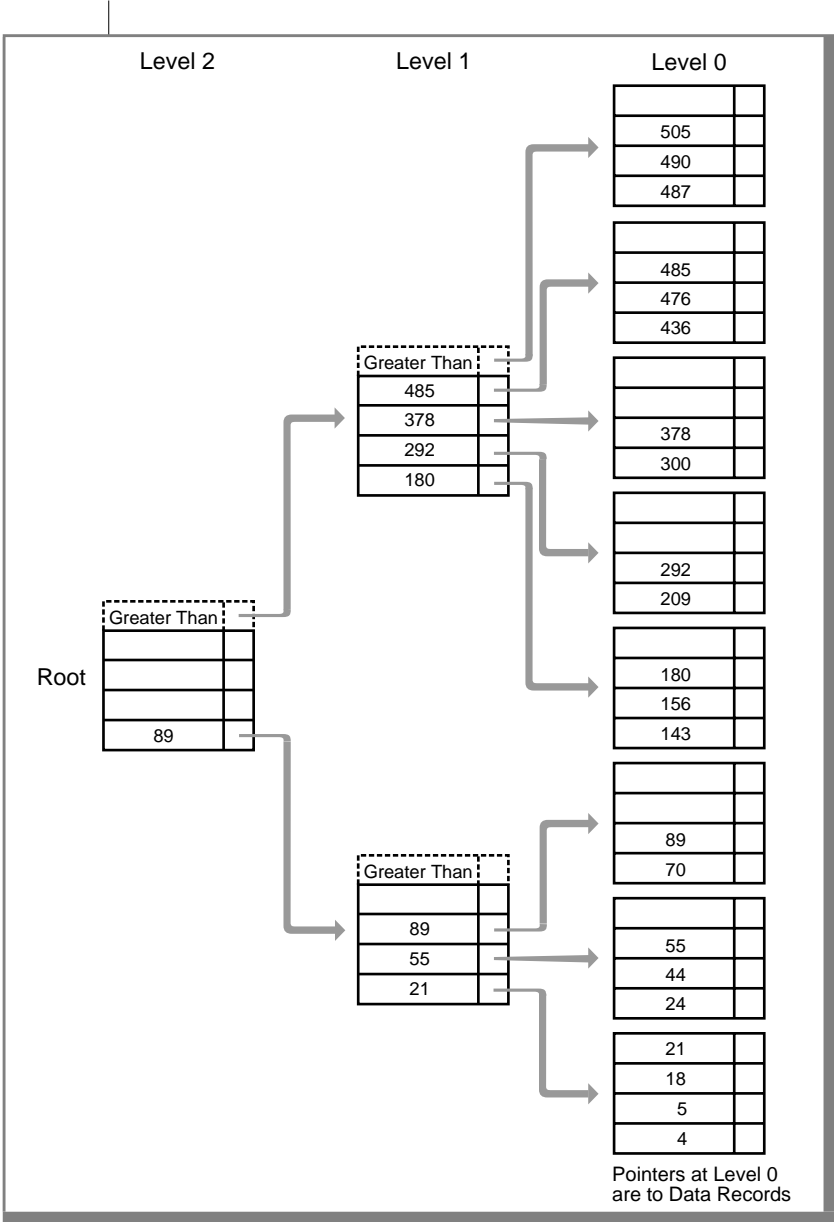


Figure 2-9
B+ Tree
Organization

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 might 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 on page 2-15 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 might 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. The root contains two pointers. One points to a node that contains keys with values less than or equal to 89. The other pointer is directed to a node that contains 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 2-9 on page 2-15 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, because 44 is less than 89. Next, the function examines the node on Level 1 that contains keys 21, 55, and 89. The function follows the pointer for key 55, because 44 is less than 55 but greater than 21. The Level 0 node contains keys 24, 44, and 55. Because 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 because 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 because 475 is less than 485 but greater than 378. At Level 0 the keys are 436, 476, and 485. Because 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, as shown in Figure 2-10. Each key entry points to a data record because 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.

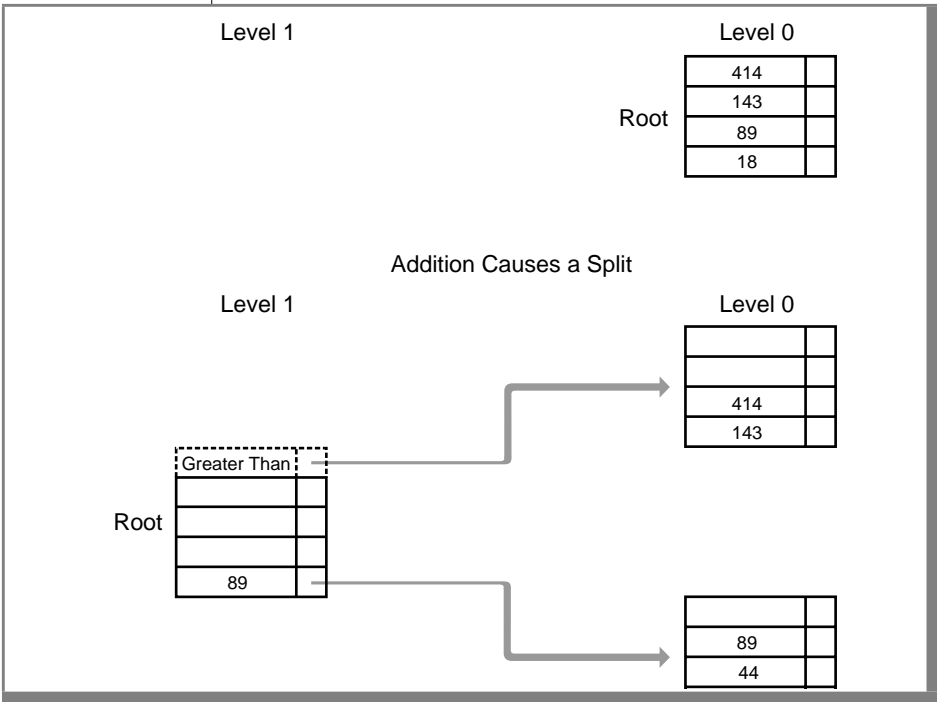


Figure 2-10
Growth of a B+ Tree

C-ISAM splits a node by selecting the median from a list that is composed of all of the values in the node plus the value of the key that is being added. 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.

Because 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. When 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.

When 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.

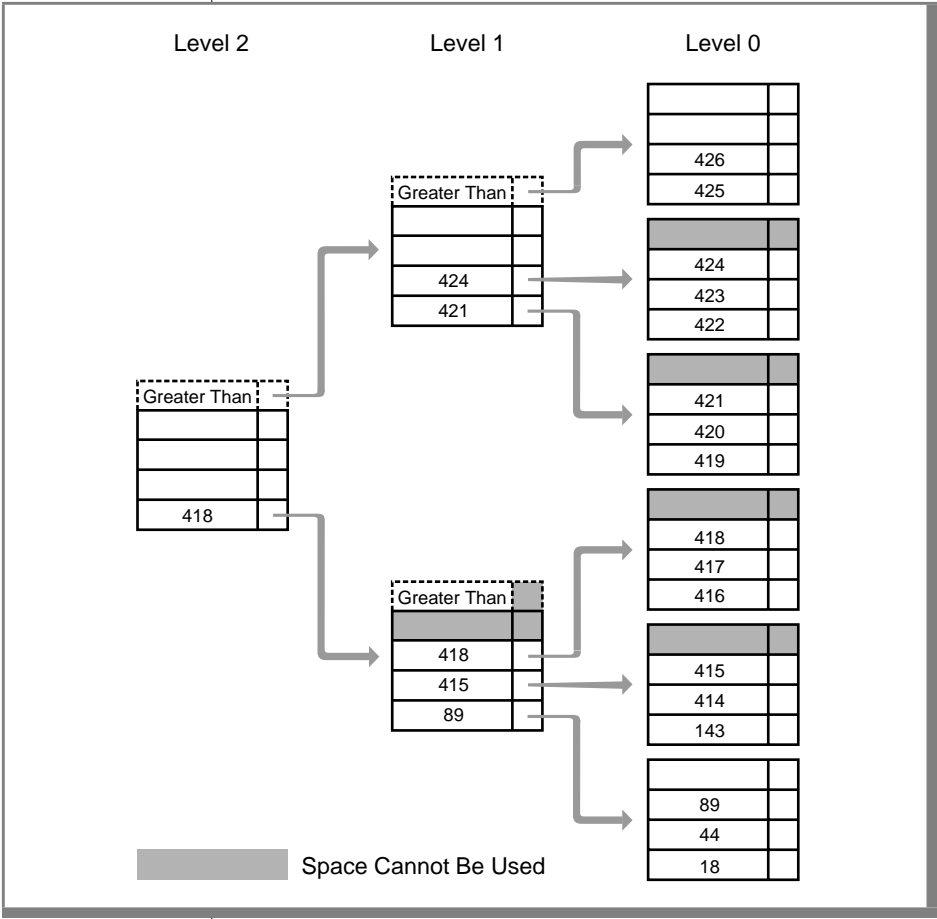


Figure 2-11
Wasted Space in B+ Trees

To avoid this problem, C-ISAM uses a different strategy. When 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*

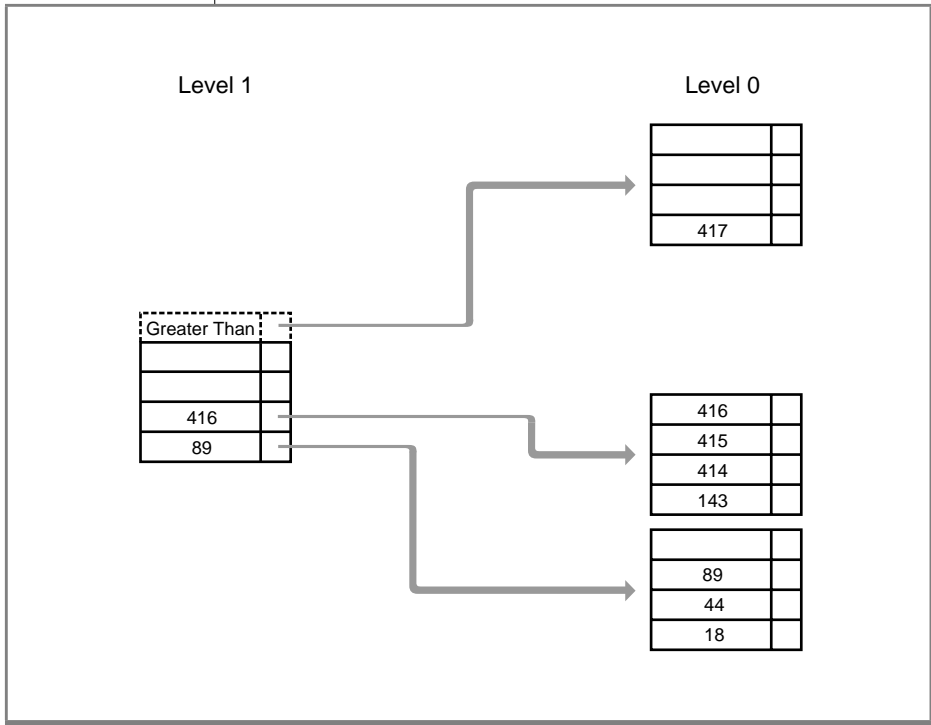


Figure 2-13 shows the effect of this strategy when key values 415 through 426 are added to this tree.

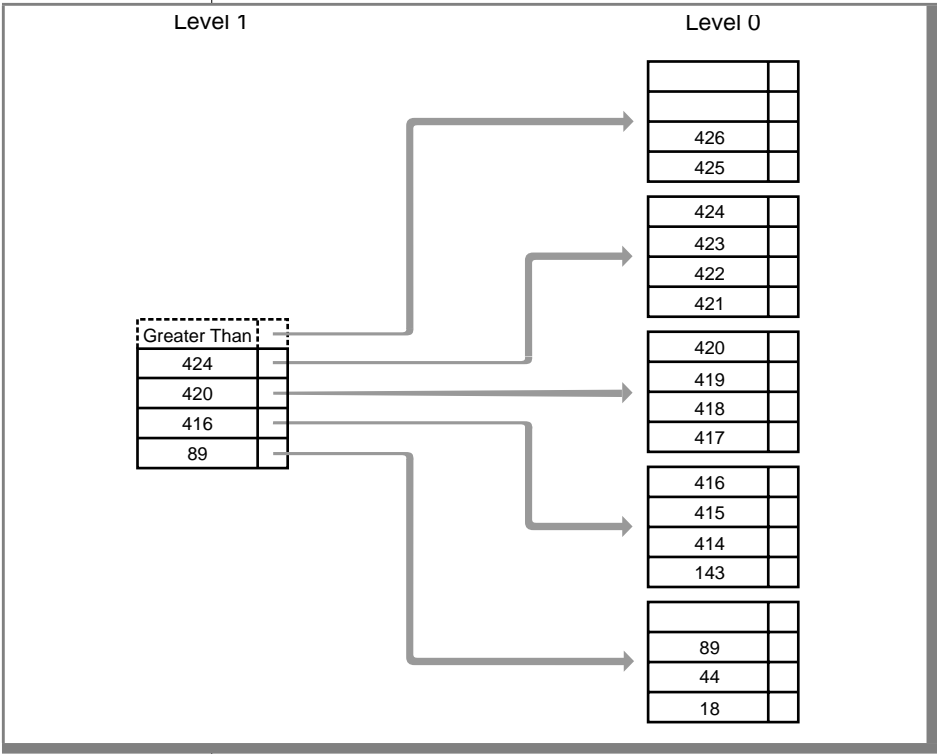


Figure 2-13
*Efficient Sequential
Addition of Keys*

Removing Keys

When you delete a record, C-ISAM removes the key from the index. When 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 that contain keys and pointers
- List nodes

Usually a one-to-one correspondence exists 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 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.

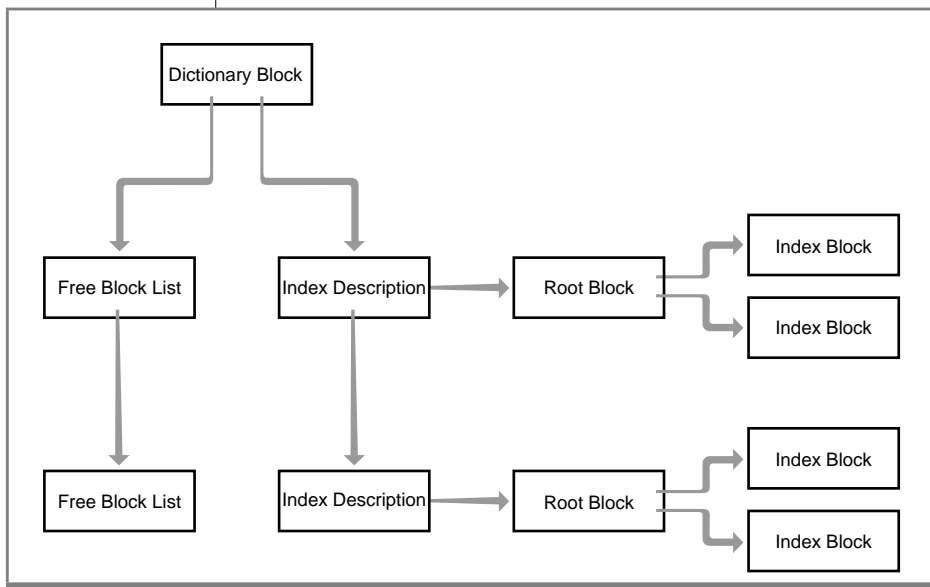


Figure 2-14
Index-File Structure

The dictionary block 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, the number of indexes, and whether an index has an NCHARTYPE key 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 on page 2-15 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 “B+ Tree Organization” on page 2-14 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 because 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. When 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 packing of nodes because, 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. When the tree in Figure 2-9 on page 2-15 grows another level, the file might hold 158 records, or $[(.75)(4+1)]^3 (.75)(4)$.

C-ISAM puts as many keys as possible into a node. More realistically, because the keys in Figure 2-9 are short integers requiring 6 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.

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. When 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 noncharacter keys or keys that are 8 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 1-byte overhead required to record the number of leading characters compressed is shown as a pound sign (#). The dots (.) represent spaces. When 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 1 byte at the beginning of the key.

1fcFigure 2-16 shows an example of this compression technique combined with leading-character compression (**k_flags**= TCOMPRESS + LCOMPRESS). The 1-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. When 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 2-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 nonduplicate 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. When 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 on page 2-15 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. When 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 because the operating-system buffers the most-recently used index blocks. Therefore, two fewer 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.

When 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.

When 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.

Localized Indexes

Access to data through a localized index is slower than through an index that does not use NCHARTYPE fields. It is more efficient to organize your data to separate NCHARTYPE data into different files from the bulk of the rest of the data, so that searches on the NCHARTYPE field will not slow down data access to the other fields.

When you do not need a localized collation sequence for a key, do not declare it as an NCHARTYPE. When you only need to search on the character field at certain times, do not include it as a key in the commonly used index. Instead, build a separate index that includes the NCHARTYPE key.

For information about collation order on C-ISAM indexes, see “Collation Order of Characters in a C-ISAM File” on page B-7. ♦

Summary

The principle features of C-ISAM indexes are shown in the following list:

- 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 can specify different orders within a key.
- C-ISAM does not impose a limit on the number of indexes that are allowed for a file.
- C-ISAM allows duplicate key values.
- You can 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

Defining Data Types for Keys	3-3
C-ISAM Computer-Independent Data Types	3-5
Defining Data Records	3-7
Data Types in Variable-Length Records	3-8
C-ISAM Data Type Conversion Routines	3-9
Character Data	3-9
Integer and Long Integer Data	3-10
Floating-Point and Double-Precision Data	3-11
DECIMALTYPE Data Type	3-13
Using DECIMALTYPE Data Type Numbers	3-13
DECIMALTYPE Data Type Declaration	3-13
Sizing DECIMALTYPE Numbers	3-14
Storing and Retrieving DECIMALTYPE Numbers	3-15
Manipulating DECIMALTYPE Numbers	3-17
Summary	3-19

C-ISAM data types provide computer 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 computer-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 for use with C-ISAM, you do not specify the data type or length of individual fields. C-ISAM needs type information only for keys. Consider the Employee record shown in Figure 3-1.

Figure 3-1
Employee Record

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	

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.

Figure 3-2
Data Type Parameters

C Language Data Type	Data Type Parameter	Data Type Mnemonic	Size Parameter (in bytes)	Size Mnemonic
char	0	CHARTYPE	—	—
int	1	INTTYPE	2	INTSIZE
long	2	LONGTYPE	4	LONGSIZE
double	3	DOUBLETTYPE	sizeof (double)	DOUBLESIZE
float	4	FLOATTYPE	sizeof (float)	FLOATSIZE
nchar [*]	7	NCHARTYPE	—	—

^{*}Indicates an Informix data type rather than a C data type.

Because **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 Computer-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 computer usually determines the length of **int** data types. If your computer has a 16-bit word length, an **int** is usually 16 bits long. If your computer 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 computers require that numbers start on a word boundary. If character data precedes numeric data, the numeric data can 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 computer architecture. This manner of storage eliminates any confusion surrounding computer-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.

Figure 3-4
C-ISAM Data Types

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	—	—	—
NCHARTYPE	char	—	—

C-ISAM integers always take 2 bytes, regardless of the internal representation of an integer on your computer.

C-ISAM does not change the representation of **float** and **double** data types. Consider using the C-ISAM DECIMALTYPE data type, described in “DECIMALTYPE Data Type” on page 3-13, as an alternative to FLOATTYPE and DOUBLETTYPE if you want complete computer independence.

Defining Data Records

Consider the record structure in Figure 3-5.

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

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	

You know the record size and the field offsets because you know the size of each field. (See Figure 3-2.) 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 computer. 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, define a **char** variable to receive records from the file and to set up records that are put into the file. The variable must be 1 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

Because 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 “C-ISAM Data Type Conversion Routines” on page 3-9 for more information about the **ld** and **st** functions.

C-ISAM Data Type Conversion Routines

Use C-ISAM functions to convert between the computer-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 computer-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 nonterminated 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 transfer data from the C language variable **cname** to the record **rec**, 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 computer-format integer from the data-file record
stint	stores a computer-format integer in a data-file record.
ldlong	returns a computer-format long integer from the data-file record.
stlong	stores a computer-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 computer-independent data types INTTYPE and LONGTYPE consist of 2-byte and 4-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 computer-format floating-point number from the data-file record.
stfloat	stores a computer-format floating-point number in a data-file record.
ldfltnull	returns a computer-format floating-point number from the data-file record and checks if it is null.
stfltnull	stores a computer-format floating-point number in a data-file record and checks if it is null.
lddbl	returns a computer-format double-precision number from a data-file record.
stdbl	stores a computer-format double-precision number in the data-file record.
lddblnull	returns a computer-format double-precision number from a data-file record, and checks if it is null.
stdblnull	returns a computer-format double-precision number in the data-file record, and checks if it is null.

DECIMALTYPE Data Type

The DECIMALTYPE data type is a computer-independent method for the representation of numbers of up to 32 significant digits, with or without a decimal point, and exponents in the range -130 to +124. 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. Routines also exist 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** header file. Appendix F contains a listing of the **decimal.h** header file.

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 on 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, six significant digits exist.

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 on 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, 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. (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 1 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 1 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.

Figure 3-11
Customer Record Using DECIMALTYPE Data Type

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	

The decimal point is implied; it is not physically present in either the **dec_t** structure or the data record. 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. The following 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 Amt. and Acct. Balance from Record */
lddecimal(&rec[26],4,&tramt);
lddecimal(&rec[30],8,&acctbal);
.
.
.
/* Store Transaction Amount and Account Balance in 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 the following arguments:

- The location where the DECIMALTYPE data starts in the data record. The location is determined by the offset in the record layout in Figure 3-11.
- The length of the DECIMALTYPE data, not the number of significant digits. See “Sizing DECIMALTYPE Numbers” on page 3-14 for a discussion on how to determine the size of a DECIMALTYPE number in a C-ISAM file.
- The address of the **dec_t** structure to receive the DECIMALTYPE number.

The **stdecimal** function has the following arguments:

- The **dec_t** structure that contains the DECIMALTYPE data
- The location in the record to receive the data
- 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, to add two DECIMALTYPE numbers you can use 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, as 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. Chapter 8 describes these function calls in detail.

Function Call	Description
stdecimal	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)

Summary

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

C-ISAM provides the following data types:

- CHARTYPE and NCHARTYPE are equivalent to the C language char data type. NCHARTYPE allows for specific collation (localized) sequencing.
- INTTYPE is a computer-independent integer that corresponds to the C language **int** data type.
- LONGTYPE is a computer-independent long integer that corresponds to the C language **long** integer data type.
- FLOATTYPE is a computer-dependent floating-point data type that corresponds to the C language **float** data type.
- DOUBLETTYPE is a computer-dependent double-precision data type that corresponds to the C language **double** data type.
- DECIMALTYPE is a computer-independent data type, which allows you to represent numbers of up to 32 significant digits with exponents in the range of -130 to +124.

Locking

Concurrency Control	4-3
Types of Locking	4-7
File-Level Locking.	4-7
Exclusive File Locking	4-8
Manual File Locking.	4-9
Record-Level Locking	4-10
Automatic Record Locking	4-10
Manual Record Locking	4-11
Waiting for Locks.	4-11
Increasing Concurrency	4-12
Error Handling	4-13
Summary	4-14

Y

ou 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 executing 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). Because the use 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.

Because 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 program 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. Because 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 one possible sequence of interleaved execution of the instructions in which the two programs do not interfere with each other.

Figure 4-1
Concurrent Execution of Programs

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

Figure 4-2
Concurrent Updates Without Locking

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

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.

Figure 4-3
Concurrent Updates with Locking

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		Add 200 (in memory)
8	2800		Writes Record and releases lock

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 ISSKIPLOCK 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.

“Increasing Concurrency” on page 4-12 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 because 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.

The code example in Figure 4-4 shows how 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 example 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. The code example in Figure 4-6 shows 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 does 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 “Data Integrity” on page 5-9 for more information.

With most implementations of C-ISAM, the operating system determines the maximum number of locked records that you can have.

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 might 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. Because no way exists for either process to get the lock it needs, both processes wait forever. Deadlocks are possible only 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 a conflict 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 because you locked the entire file, even though no actual conflict exists. Concurrency is unnecessarily impaired, in this case, because 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 on page 4-6 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 the function.

```
.
.
.
/* 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 might want to retry the **isread** call only a few times, rather than to retry for a long time.

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:

- ISEXCLLOCK 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

Why Use Transaction Management?	5-3
Transaction-Management Services	5-4
Implementing Transactions	5-4
Transactions with Variable-Length Records	5-7
Logging and Recovery	5-7
Data Integrity.	5-9
Concurrent Execution of Transactions	5-9
Locking	5-10
Concurrency Issues	5-11
Summary	5-12

At times, you will 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 to complete entirely or not at all.

Why Use Transaction Management?

Assume 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 might 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 on 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 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 indivisible 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 might 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 “Locking” on page 5-10.)

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 because 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 transaction-log formats that are used with variable-length records are different than those used with fixed-length records. All the transaction-log formats that are used in the transaction logs are listed in Appendix D.

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 that 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 shown in the following example:

```
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. Ensure that no one is using the C-ISAM files during the recovery.



Warning: *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, a transaction must contain all **isopen** calls that are intended to open the file in a write mode. The files must also be closed before the transaction is committed or rolled back. If you want a file to open in read-only mode and not 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 any cases exist 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 that 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 that are 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, and then Program A rewrites it, the Program B update is lost. This is shown in Figure 5-2.

Figure 5-2
Concurrent Updates Without Locking

Time	Amount Field in Record 1	Program A	Program B
0	2500		
1	2500	Reads Record 1	
2	2500		Reads Record 1

(1 of 2)

Time	Amount Field in Record 1	Program A	Program B
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

(2 of 2)

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 transaction does not modify the records in the file.

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

The number of record locks that can exist at any one time is operating-system dependent.

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.

Choose an appropriate strategy for handling situations where a C-ISAM call returns an indication that a record is locked. (See “Error Handling” on page 4-13 for a description of how locked records are identified.) The safest strategy is to roll back the transaction. This strategy guarantees that transactions 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.

Figure 5-3
Conflict Resolution with Transactions

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	Adds 100 (in memory)	
4	2600	Writes Record 1	
5	2600	Commits	Retries
6	2600		Reads and locks Record 1
7	2600		Adds 200 (in memory)
8	2800		Writes Record 1
9	2800		Commits

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; 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 indivisible unit of work that consists of one or more C-ISAM function calls, operating on one or more files. The following functions implement transactions.

Function	Description
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

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	6-10
File Maintenance with Variable-Length Records	6-11
If Data Files Are Corrupted	6-11
If Index Files Are Corrupted	6-11
Summary	6-14

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 filenames
- 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 constitute 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**.

The following call renames **employee.dat** to **personnel.dat** and **employee.idx** to **personnel.idx**:

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

Any other files that are 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 constitute the C-ISAM file. The following example removes the files **personnel.dat** and **personnel.idx**:

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

This function also removes the audit-trail file for the personnel file if one exists. See “Audit-Trail Facility” on page 6-6. It does not remove transaction-log files.

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

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);
```

The file descriptor, **fd**, was returned when the file was opened or built.

If you have a multiuser 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 that are 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 as unique, you can add a **long** integer field to the records to serve as a unique key. 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. The function call is shown in the following example:

```
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 that is specified for the key. If, for example, the first 4 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. To start the value 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. 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 that transactions provide. 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 functions for transaction management, you can use an audit-trail file. If you use both support functions and audit-trail files, 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 = 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);

```

Figure 6-1
Using the *isaudit*
Function Call

The **isaudit** function calls in Figure 6-1 perform different tasks depending on the third argument, the mode. The following list describes the action that **isaudit** takes, based on 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 character is equal to 1, changes to the C-ISAM file are recorded.
AUDSTART	starts the audit trail. 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 2-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 shown in the following list:

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 variable that contains the time in UNIX format.
au_procid	is an INTTYPE variable that contains the process identification number.
au_userid	is an INTTYPE variable that contains the user identification code.
au_recnum	is a LONGTYPE variable that contains the number of the record that is added, deleted, or modified.
au_reclen	is a LONGTYPE variable 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 feature 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. Although the keys in an index node are physically adjacent, no guarantee exists 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 clustering 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. A call to **iscluster** 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 re-create an index file simply by knowing the key descriptions and some dictionary information. With variable-length records, you can re-create the index portion of the **.idx** files with the same information, but you cannot re-create 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 that the parameters specify, re-creates 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 **bcheck** utility or the **iscluster** function do not repair the variable-length data. 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 must restore all the files, both **.dat** and **.idx**, from a backup.

Figure 6-3 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.

Figure 6-3
Program to Create newfile from oldfile

```
#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;
}

```

Appendix D describes the format of the index files.

Summary

C-ISAM provides the following additional functions.

Function	Description
isrename	Changes the name of a C-ISAM file
iserase	Removes a C-ISAM file
isflush	Forces output to a C-ISAM file that is opened exclusively or is on a single computer 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

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	7-8
Random Update	7-11
Sequential Access	7-16
Chaining	7-19
Using Transactions	7-25
Summary	7-29

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.

Figure 7-1
Employee 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

The **perform** file holds information that pertains 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.

Figure 7-2
Performance File Layout

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

You must allocate 1 more byte for C-ISAM records in memory. Because 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 one of the following status codes 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. Appendix C lists and describes the values that are returned in **iserrno**.

The sample programs that follow do not always illustrate adequate error checking. (This omission is designed 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.

Figure 7-3
Creating C-ISAM Files with bld_file.c

```
#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);
}
```

The primary key for the **employee** file has one part, the Employee Number. This primary key is a long integer beginning at offset 0, the start of the record. It is 4 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, 4 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 6 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.

When 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 practice is appropriate when file insertions, deletions, or updates are a major activity because extra indexes slow down these operations.

A modified version of this program, which adds a localized index, is described in “An Example of Adding a Localized Index” on page B-9. ♦

Figure 7-4
Adding Additional Indexes with add_idx.c

```
#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", iserr
no);
        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);
}
```

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.

Figure 7-5
Adding Records to C-ISAM Files with add_rcrd.c

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

```

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. You do not have to use **isstart** with each **isread** if you use the entire key to locate a record.

Figure 7-6
Random Update of C-ISAM Files with upd_file.c

```
#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();
        updatemp();
    }
    cmd = getinstr();
}

isclose(fdemploy);
isclose(fdperform);
}
updatemp()
{
    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,"%1s",instr);
if (strcmp(instr,"y")==0)
    return (DELETE);
else
    {
        printf("Update? (y/n): ");
        fgets(line,80,stdin);
        sscanf(line,"%1s",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 returns */
        *dest++ = *src++;
    while (i++ <= num)
        /* or nulls; pad remaining */
        *dest++ = ' ';
    /* characters in blanks */
}

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

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.

Figure 7-7
Sequential Processing of a C-ISAM File with sqntl_rd.c

```
#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);
}

```

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.

Figure 7-8
Sample Performance Data

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

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 on 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 that **isstart** found.

To obtain the most-recent record for Employee 1 in Figure 7-8

1. Call **isstart** with the ISGTEQ mode and a key that contains Employee 1 and Review Date 999999.
The **isstart** function positions at Employee 2, Review Date 760301, because 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 that **isstart** found.

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 and then finding subsequent records with the ISNEXT mode in the **isread** function call.

Figure 7-9

Chaining to the Last Record in a List with chaining.c

```
#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;

    memcpy(perfrec, operfrec, 4); /* get employee id no. */
    memcpy("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++ = ' ';
}

```

Using Transactions

Figure 7-10 shows a sample program that has been modified to define C-ISAM operations as transactions. (Figure 7-5 on page 7-8 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 that are 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**.

Figure 7-10

Adding Records Inside a Transaction with transctn.c

```

#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", iserrn
o);
        scleanup();
        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+ISTR
ANS);
        if (cc < SUCCESS)
        {
            isrollback();
            break; }

        fdperform = cc = isopen("perform", ISMANULOCK+ISOUTPUT+ISTR
```

```

ANS);
    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 */
        {
            srollback();
        }
    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 returns */
        /* or nulls
        *dest++ = *src++;
    pad remaining */
    while (i++ <= num) /* characters in blanks */
        *dest++ = ' ';
}

```

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

Functions for C-ISAM File Manipulation	8-9
isaddindex	8-10
isaudit	8-12
isbegin	8-14
isbuild	8-16
iscleanup	8-19
isclose	8-20
iscluster	8-21
iscommit	8-23
isdelcurr	8-25
isdelete	8-26
isdelindex	8-28
isdelrec	8-30
iserase	8-31
isflush	8-32
isglsversion	8-33
isindexinfo	8-35
islanginfo	8-38
islock	8-39
islogclose	8-40
islogopen	8-41
isnlsversion	8-42
isopen	8-44
isread	8-46
isrecover	8-50
isrelease	8-51
isrename	8-52
isrewcurr	8-53
isrewrec	8-55
isrewrite	8-56
isrollback	8-58

issetunique	8-60
isstart	8-61
isuniqueid	8-64
isunlock	8-65
iswrcurr	8-66
iswrite	8-68
Format-Conversion and Manipulation Functions	8-70
ldchar	8-71
lddbl	8-72
lddblnull	8-73
lddecimal	8-74
ldfloat	8-76
ldftnull	8-77
ldint	8-78
ldlong	8-79
stchar	8-80
stdbl	8-81
stdblnull	8-82
stdecimal	8-83
stfloat	8-85
stftnull	8-86
stint	8-87
stlong	8-88
DECIMALTYPE Functions	8-89
decadd, decsub, decmul, and decdiv	8-90
deccmp.	8-92
deccopy	8-93
deccvasc	8-94
deccvdbl	8-96
deccvflt	8-97
deccvint	8-98
deccvlong	8-99
dececv and decfcvt	8-100
dectoasc	8-102
dectodbl	8-104
dectoft.	8-105
dectoint	8-106
dectolong	8-107
Summary	8-108

This chapter describes all the functions that are available as part of C-ISAM. These functions 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
- Determine GLS-related information about a file
- Perform other functions that are associated with maintaining C-ISAM files

The following functions allow you to manipulate files and indexes.

Function	Description
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

(1 of 2)

Function	Description
iserase	Removes a C-ISAM file
isaddindex	Adds an index to a file
isdelindex	Removes an index from a file

(2 of 2)

The following functions allow you to manipulate C-ISAM records.

Function	Description
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.

Function	Description
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.

Function	Description
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 functions allow you to determine GLS-related and (for backward compatibility) NLS-related information.

Function	Description
isglsversion	Determines if a localized collation has been associated with a file
islanginfo	Returns the collation string associated with a localized index file
isnlsversion	Determines if a localized collation has been associated with a file



The following additional functions are also available with C-ISAM.

Function	Description
isaudit	Maintains an audit trail
isuniqueid	Determines the last unique ID for a record

Function	Description
issetunique	Sets the starting unique ID
isindexinfo	Determines the characteristics of a file and its indexes
isflush	Forces output to a C-ISAM file

(2 of 2)

The following functions convert between computer-dependent representation of numbers and the C-ISAM counterparts.

Function	Description
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 computer-dependent integer
stint	Converts a computer-dependent integer to a C-ISAM integer
ldlong	Converts a C-ISAM long integer to a computer-dependent long integer
stlong	Converts a computer-dependent long integer to a C-ISAM long integer
ldfloat	Converts a C-ISAM floating-point number to a computer-dependent floating-point number
stfloat	Converts a computer-dependent floating-point number to a C-ISAM floating-point number
ldftnull	Converts a C-ISAM floating-point number to a computer-dependent floating-point number and checks if it is null
stftnull	Converts a computer-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 computer-dependent double-precision number

(1 of 2)

Function	Description
stdbl	Converts a computer-dependent double-precision number to a C-ISAM double-precision number
lddblnull	Converts a C-ISAM double-precision number to a computer-dependent double-precision number and checks if it is null
stdbnull	Converts a computer-dependent double-precision number to a C-ISAM double-precision number and checks if it is null

(2 of 2)

The following functions allow you to manipulate the C-ISAM DECIMALTYPE data type. They are listed here in logical order; the reference pages for these functions are in alphabetical order according to function name.

Function	Description
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 computer-dependent integer into a DECIMALTYPE number
dectoint	Converts a DECIMALTYPE number into a computer-dependent integer
deccvlong	Converts a computer-dependent long integer into a DECIMALTYPE number
dectolong	Converts a DECIMALTYPE number into a computer-dependent long integer
deccvflt	Converts a computer-dependent floating-point number into a DECIMALTYPE number
dectoflt	Converts a DECIMALTYPE number into a computer-dependent floating-point number

(1 of 2)

Function	Description
deccvdbl	Converts a computer-dependent double-precision number into a DECIMALTYPE number
dectodbl	Converts a DECIMALTYPE number into a computer-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	Converts a DECIMALTYPE value to an ASCII string
decfcvt	Converts a DECIMALTYPE value to an ASCII string

(2 of 2)

Functions for C-ISAM File Manipulation

This section describes the following functions in alphabetical order.

isaddindex	islogclose
isaudit	islogopen
isbegin	isnlsversion
isbuild	isopen
iscleanup	isread
isclose	isrecover
iscluster	isrelease
iscommit	isrename
isdelcurr	isrewcurr
isdelete	isrewrec
isdelindex	isrewrite
isdelrec	isrollback
iserase	issetunique
isflush	isstart
isglsversion	isuniqueid
isindexinfo	isunlock
islanginfo	iswrcurr

isaddindex

Use **isaddindex** to add an index to a C-ISAM file.

Syntax

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

isfd is the file descriptor returned by **isopen** or **isbuild**.

keydesc is a pointer to a key-description structure.

Usage

The C-ISAM file must be opened for exclusive access (ISEXCLLOCK), and it must be open for both input and output (ISINOUT).

C-ISAM does not limit the number of indexes that you can add. However, you can define indexes only 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 8-16 for more information on the *keydesc* parameter.

The maximum number of parts that you can define for an index is NPARTS. The **isam.h** file contains the definition of NPARTS, which is usually NPARTS equals 8. The **isam.h** file also contains the definition of MAXKEYSIZE (the maximum key size). MAXKEYSIZE is usually 120 bytes.

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

Use **isaudit** to perform operations that involve an audit-trail file. You can start or stop recording changes to a C-ISAM file, or you can set the name of an audit-trail file. You can also determine whether the audit trail is on or off.

Syntax

```
int isaudit(isfd, filename, mode)
    int isfd;
    char *filename;
    int mode;
```

<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>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>	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.
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.								

Usage

When the mode equals AUDINFO, the function sets the first byte of the *filename* parameter (*filename*[0]) as follows:

- If the audit trail is off, the first byte is set to 0 (ASCII null).
- If the audit trail is on, the first byte is set to 1.

When you stop the audit trail, it is not erased. Further changes to the C-ISAM file, however, are not recorded. 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.

C-ISAM retains the name in the index (.idx) file when you set the audit-trail filename. You can create a new audit-trail file, either by removing the old file or by setting a new filename. The audit-trail filename can be any operating-system filename or pathname.

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.”

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

Use **isbegin** to define the beginning of the transaction.

Syntax

```
int isbegin( )
```

Usage

If you are using a log file, you must call **isbegin** before you open the file for a read-only (ISINPUT) operation.

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

Use **isbuild** to create a C-ISAM file.

Syntax

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

<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.

Usage

If you do not use ISFIXLEN or ISVARLEN, the record length defaults to fixed length. 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.

The **isbuild** function creates two operating-system files with the names *filename.dat* and *filename.idx*. These files are treated together as one logical C-ISAM file. The *filename* parameter should contain a null-terminated character string that is at least four characters shorter than the longest legal operating-system filename and no longer than 10 characters. The function returns an integer file descriptor that identifies the file.

The file is left open with the access and locking modes that are set in the mode parameter.

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.

You can add indexes later by using **isaddindex**.

If you open 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.

If you open 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 of subsequent C-ISAM calls on the file. If you use ISNOLOG, the **isbuild** call will still be logged. If you use ISNOLOG, be sure that all future **isopen** calls for this file also specify ISNOLOG.

If you use the ISTRANS parameter, you must have already called **islogopen**.

The **isbuild** function cannot be rolled back.

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+ISEXCLOCK))<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

Use **iscleanup** to close all of the C-ISAM files that your program opened.

Syntax

```
int iscleanup()
```

Usage

Make it standard practice to call **iscleanup** before you exit a C-ISAM program.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
cc = iscleanup();
```

isclose

Use **isclose** to close a C-ISAM file.

Syntax

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

isfd is the file descriptor returned by **isopen** or **isbuild**.

Usage

Outside the scope of a transaction, a call to **isclose** releases any locks that your program holds on the table. Within a transaction, the locks are held until the transaction is committed by an **iscommit** or rolled back by **isrollback**.



Warning: *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

Use **iscluster** to change the physical order of a C-ISAM file to key sequence.

Syntax

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

isfd is the file descriptor of the file that you want to modify.
keydesc is a pointer to the key-description structure that specifies the new physical order for the file.

Usage

The C-ISAM file must be opened for exclusive access; the file cannot have an audit trail at the time that you use the function.

The function copies the records of the file to a new file and returns a new file descriptor that must be used with the new file. The records in the new file are physically in the order that the key defines. 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.

The **iscluster** function re-creates all indexes. You can use any index to specify the physical order of the file. 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.

The **iscluster** call cannot be rolled back within a transaction but can be recovered.

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

Use **iscommit** to end a transaction and release all locks.

Syntax

```
int iscommit( )
```

Usage

All changes to the C-ISAM files within the transaction occur as calls are made. The **iscommit** function marks the transaction as completed in the log file so that the changes are rolled forward when the file must be recovered. The function releases any locks that the transaction holds.

Calling **iscommit** without a prior call to **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

Use **isdelcurr** to delete the current record from the C-ISAM file.

Syntax

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

isfd is the file descriptor returned by **isopen** or **isbuild**.

Usage

The **isdelcurr** function removes the key from each existing index. The function is useful when, for example, you want to delete the most-recent record read with **isread**.

The **isrecnum** global variable is set to the record number of the deleted record.

The current record is undefined because 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

Use **isdelete** to delete a record using the primary key.

Syntax

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

isfd is the file descriptor returned by **isopen** or **isbuild**.
record contains a key value in the position defined for the primary key.

Usage

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. If the primary index is not unique, use **isread** to find the record and **isdelcurr** to delete it.

You cannot use this function with files that are created with INFORMIX-4GL, INFORMIX-SQL, or an SQL API, such as INFORMIX-ESQL/C, because the C-ISAM files that constitute SQL databases do not contain primary indexes. Use **isdelcurr** instead.

The function removes the key of the record from each index.

The **isdelete** function does not change the current record.

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

Use **isdelindex** to remove an existing index.

Syntax

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

isfd is the file descriptor returned by **isopen** or **isbuild**.

keydesc is a pointer to a key-description structure.

Usage

You must open the C-ISAM file for exclusive access.

You can use **isdelindex** to delete any index except the primary index. The key-description structure identifies the index that you want to delete.

The **isdelindex** call cannot be rolled back within a transaction but can be recovered.

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

Use **isdelrec** to delete a record using the record number.

Syntax

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

isfd is the file descriptor returned by **isopen** or **isbuild**.

recnum is the record number of the data-file record.

Usage

The **isdelrec** function uses the record number to find the record that you want to delete. Use this function 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. The **isdelrec** function does not change the current record position.

The **isdelrec** function removes the key from each index.

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

Use **iserase** to remove the operating-system files that constitute the C-ISAM file.

Syntax

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

filename is the C-ISAM file that you want to delete.

Usage

Do not use a filename extension with the *filename* argument. The function deletes *filename.idx* and *filename.dat* (and the audit-trail file, if it exists).

You must close the file that you want to delete before you call **iserase**.

The **iserase** function cannot be rolled back within a transaction but can be recovered.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

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

isflush

Use **isflush** to immediately flush any buffered index pages to the operating system.

Syntax

```
int isflush(isfd)
int isfd;
```

isfd is the file descriptor returned by **isopen** or **isbuild**.

Usage

Ordinarily, C-ISAM flushes data to the operating system after each function call. 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 that are opened for exclusive access. Periodic calls to **isflush** protect you against substantial loss of data during a system crash.

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);
```


isglsversion

Use **isglsversion** to determine whether a localized collation sequence is associated with a C-ISAM file.

In C-ISAM, Version 7.2, the **isglsversion** function replaces the **isnlsversion** function.

Syntax

```
int isglsversion(name)
    name *char;
```

name is the specification of an existing C-ISAM file.

Usage

When **isglsversion** is true, you can use **islanginfo** to determine the name of the collation sequence. For more information, see “Determining Index-Collation Sequence” on page B-11.

Return Codes

- | | |
|----|--|
| -1 | Error; iserrno contains the error code. |
| 0 | No localized collation sequence is associated with the file. |
| 1 | A localized collation sequence is associated with the file. |

Example

```
#include <isam.h>
#define SUCCESS 0

int cc;

main()
{
    /* determine if file was built with localized collation*/
    cc = isglsversion("employee");

    if (cc < SUCCESS)
    {
        printf("isglsversion error %d for employee file\n",
            iserrno);
        exit(1);
    }
    if (cc == 1)
    {
        printf("\nemployee file HAS LOCALIZED indexes \n");
        /*use islanginfo to get the value of lang used for index */
        printf("The LANG of the index is %s\n",
            islanginfo("employee") );
        exit(1);
    }
    if (cc == 0) printf("\nemployee file DOES NOT HAVE LOCALIZED
        indexes \n");
}
```



isindexinfo

Use **isindexinfo** to determine information about the structure and indexes of a C-ISAM file.

Syntax

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

isfd is the file descriptor returned by **isopen** or **isbuild**.

buffer is a pointer to a structure.

number is either an index number or zero.

Usage

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.

To get general information, including the number of indexes, index-node size, and data-record size, you call **isindexinfo** with *number* set to 0 and with a *buffer* of structure type **dictinfo**.

Indexes are associated with numbers, starting with 1. The primary index is always index 1. 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**.

When 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, when the file has variable-length records, the **di_nkeys** and **di_recsz** variables pointed to by *buffer* contain information that is specific to the variable-length records as shown in the following list:

- **di_nkeys**

When the file supports variable-length records, the significant bit is set. The remaining bits indicate the number of indexes that are defined for the file.

- **di_recsz**

This field contains the maximum record size in bytes.

See “Determining Index Structures” on page 2-12 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 shown in the following example:

```
#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 shown in the following example:

```
#include <isam.h>
struct dictinfo info;
struct keydesckdesc;
.
.
.
/* 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--);
.
.
.
}
```

islanginfo

Use **islanginfo** to determine what, if any, localized collation sequence was associated with a C-ISAM index file.

Syntax

```
char * islanginfo(name)
char * name ;
```

name is the name of an existing C-ISAM file.

Usage

If a localized collation is associated with the file, the collation specification string is returned. If not, null is returned. For more information about using collation sequences, see “Determining Index-Collation Sequence” on page B-11.

Use **islanginfo** after calling **isglsversion**; **isglsversion** determines if a localized collation sequence is associated with an index file.

Example

```
#include <isam.h>

/* determine if file was built with localized collation*/
cc = isglsversion("employee");
if (cc < SUCCESS)
{
    printf("isglsversion error %d for employee file\n",
        iserrno);
    exit(1);
}
if (cc == 1)
{
    printf("\nemployee file HAS LOCALIZED indexes \n");
    /* use islanginfo to get the value of lang used for index */
    printf("The LANG of the index is %s\n", islanginfo("employee") );
    exit(1);
}
if (cc == 0) printf("\nemployee file DOES NOT HAVE LOCALIZED indexes \n");
}
```

◆

islock

Use **islock** to lock the entire C-ISAM file.

Syntax

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

isfd is the file descriptor of the file you want to lock that is returned by **isopen** or **isbuild**.

Usage

You must open the file with the ISMANULOCK mode.

Other programs can read records but they cannot update records.

You can release the lock with **isunlock**. 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

Use **islogclose** to close the transaction-log file.

Syntax

```
int islogclose()
```

Usage

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

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

```
int islogopen(logname) char *logname;
```

logname is a pointer to the filename string.

Usage

The log file must already exist to use **islogopen**.



Warning: *If the log file does not exist, C-ISAM function calls still work, but no log-file records are saved and recovery is impossible.*

Once you call **islogopen**, you can use ISTRANS mode in an **isbuild** or **isopen** function call.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

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

isnlversion

The **isnlversion** function determines whether a localized collation sequence is associated with a C-ISAM file. This function is retained in C-ISAM, Version 7.2, for backward compatibility with existing Version 6.0 and 7.1 C-ISAM programs.

Use the **isglversion** function for programs that you write with C-ISAM, Version 7.2.

Syntax

```
int isnlversion(name)
    name *char;
```

name is the specification of an existing C-ISAM file.

Usage

When **isnlversion** is true, you can use **islanginfo** to determine the name of the collation sequence. For more information, see “Determining Index-Collation Sequence” on page B-11.

Return Codes

- | | |
|----|--|
| -1 | Error; iserrno contains the error code. |
| 0 | No localized collation sequence is associated with the file. |
| 1 | A localized collation sequence is associated with the file. |

Example

```
#include <isam.h>
#define SUCCESS 0

int cc;

main()
{
    /* determine if file was built with localized collation*/
    cc = isnlsversion("employee");

    if (cc < SUCCESS)
    {
        printf("isnlsversion error %d for employee file\n",
iserrno);
        exit(1);
    }
    if (cc == 1)
    {
        printf("\nemployee file HAS localized indexes \n");
        /*use islanginfo to get the value of lang used for index */
        printf("The LANG of the index is %s\n",
islanginfo("employee") );
        exit(1);
    }
    if (cc == 0) printf("\nemployee file DOES NOT HAVE localized
indexes \n");
}
```



isopen

Use **isopen** to open a C-ISAM file for processing.

Syntax

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

filename is the name of the file.

mode 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.



Warning: *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**.*

Usage

The **isopen** function returns the file descriptor that you must use in subsequent operations on the C-ISAM file. 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.

The *filename* parameter must contain a null-terminated string without an extension, which is the filename of the C-ISAM file to be processed.

When you use the ISVARLEN parameter with the function call, the global integer **isreclen** is set to the maximum record length for the file. 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.

If you use the ISTRANS parameter, you must have already called **islogopen**.

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

Use **isread** to read records sequentially or randomly, as indicated by the *mode* parameter.

Syntax

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

<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. Optionally, you can add one or more of the following locking options to the search mode: <table> <tr> <td>ISLOCK</td><td>locks the record.</td></tr> <tr> <td>ISSKIPLOCK</td><td>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.</td></tr> <tr> <td>ISWAIT</td><td>causes the process to wait for a locked record to become free.</td></tr> <tr> <td>ISLCKW</td><td>is the same as ISLOCK+ISWAIT.</td></tr> </table> </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. Optionally, you can add one or more of the following locking options to the search mode: <table> <tr> <td>ISLOCK</td><td>locks the record.</td></tr> <tr> <td>ISSKIPLOCK</td><td>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.</td></tr> <tr> <td>ISWAIT</td><td>causes the process to wait for a locked record to become free.</td></tr> <tr> <td>ISLCKW</td><td>is the same as ISLOCK+ISWAIT.</td></tr> </table>	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.
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: <table> <tr> <td>ISLOCK</td><td>locks the record.</td></tr> <tr> <td>ISSKIPLOCK</td><td>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.</td></tr> <tr> <td>ISWAIT</td><td>causes the process to wait for a locked record to become free.</td></tr> <tr> <td>ISLCKW</td><td>is the same as ISLOCK+ISWAIT.</td></tr> </table>	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.																
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.																								

Usage

Place the search value in the *record* in the appropriate position for the key. If the search is successful, **isread** fills the remainder of the *record* with the returned record. The record becomes the current record for the file.

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 that are returned in the record buffer. The contents of the buffer beyond the value of **isreclen** are undefined.

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.

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**, **isrollback**, or **isclose** when you call **isclose** outside of a transaction.

A Dirty Read occurs when you use **isread** without ISLOCK, and the **isread** gets a record that is locked by another process, which you can read. However, if you use **isread** with ISLOCK and the **isread** encounters a record that another process has locked, then ELOCKED is returned (unless you set ISWAIT).

When 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.

When you use **isread** with ISCURRE or ISNEXT after you add a record with **iswrite**, **isread** returns the record that you just added. If you use **isread** with ISCURRE or ISNEXT after you make an **isstart** call, **isread** returns the starting record in either case.

When 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**.

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. Issue another **isread** (ISNEXT) call to skip to the next record.

When 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.

You can use ISWAIT and ISLCKW only if your version of C-ISAM uses the **fcntl()** call for record locking.

When **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**.
- Once an **isread** call returns EENDFILE, the current record position is undefined. If you make another **isread**(ISNEXT) call, the ENOCURR code is returned.
- If you use **isread**(ISEQUAL) and no records match the record that you are looking for, ENOREC is returned. The entries of the file are searched sequentially, starting with the record that is indicated by the value in **isrecnum** until the end of the file is reached.

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

Use **isrecover** along with the log file to redo all committed transactions in a copy of the C-ISAM file.

Syntax

```
int isrecover( )
```

Usage

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. The log file must already be open by a call to **islogopen**.

No one should use the C-ISAM files before the function finishes executing.

***Important:** 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

Use **isrelease** to unlock records that are locked by calls to **isread** with the ISLOCK option.

Syntax

```
int isrelease(isfd)
int isfd;
```

isfd is the file descriptor returned by **isopen** or **isbuild**.

Usage

The **isrelease** function unlocks all records that your program locked in the C-ISAM file. A call to **isrelease** during a transaction only releases unmodified records.

If you used an **isstart** call with the ISKEEPLOCK option, you must use **isrelease** to unlock the record.

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

Use **isrename** to change the name of a C-ISAM file.

Syntax

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

oldname is the file that you want to rename.

newname is the name of the new file.

Usage

Do not specify a filename extension for the C-ISAM file.

The **isrename** function renames the **.dat** and **.idx** files but does not change the name of audit-trail files or transaction-log files because their names are not logically tied to the C-ISAM filename.

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. When you use a relative pathname, keep in mind the current working directory of the program.

The **isrename** function cannot be rolled back within a transaction but can be recovered.

Return Codes

-1 Error; **iserrno** contains the error code

0 Successful

Example

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

isrewcurr

Use **isrewcurr** to modify or update fields in the current record.

Syntax

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

isfd is the file descriptor returned by **isopen** or **isbuild**.

record contains the complete record including updated fields.

Usage

When 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.

When you change a key field, C-ISAM updates the index entry. You can change the value of the primary-key field.

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);
```

When 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 1,084 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

Use **isrewrec** to update a record that is identified by its record number.

Syntax

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

isfd is the file descriptor returned by **isopen** or **isbuild**.
recnum is the record number.
record contains the complete record including updated fields.

Usage

When 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.

If you change a key field, C-ISAM updates the index entry. You can change the value of the primary-key field.

The function sets **isrecnum** to the record number of the record. 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

Use **isrewrite** to rewrite the nonprimary key fields of a record in a C-ISAM file.

Syntax

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

isfd is the file descriptor returned by **isopen** or **isbuild**.
record contains the complete record including the primary key and the updated fields.

Usage

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.

The primary key in the *record* identifies the record that you want to rewrite.

The primary index must be unique.

You cannot change the value of the primary-key field. When you change a key field in a nonprimary index, the function updates the index.

You cannot use this function with files that are created with INFORMIX-4GL, INFORMIX-SQL, or an SQL API such as INFORMIX-ESQL/C, because the C-ISAM files that constitute SQL databases do not contain primary indexes. Use **isrewcurr** or **isrewrec** instead.

C-ISAM does not change the current record position.

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

Use **isrollback** to cancel the effect of C-ISAM calls since the last call to **isbegin**.

Syntax

```
int isrollback( )
```

Usage

The **isrollback** function returns any modified records to their original unmodified state.

You must include the ISTRANS parameter as part of the mode in the **isopen** function to effect the reversal of modified records.

You cannot roll back the following functions:

isrename	iserase
isaudit	isrename
isbuild	issetunique
iscluster	isuniqueid
isdelindex	

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

Use **issetunique** to set the value of the internally stored unique identifier.

Syntax

```
int issetunique(isfd, uniqueid)
    int isfd;
    long uniqueid;
```

isfd is the file descriptor returned by **isopen** or **isbuild**.
uniqueid is a **long** integer that specifies the new unique identifier.

Usage

A *uniqueid* is maintained for each C-ISAM file. You can use **issetunique** when you need a unique primary key value for a record, and no other part of the record is suitable.

When the value of the *uniqueid* is less than the current unique identifier, the function does not change the value. You can use **isuniqueid** to determine the greatest *uniqueid*.

The **issetunique** call cannot be rolled back within a transaction but can be recovered.

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

Use **isstart** to select the index and the starting point in the index for subsequent calls to **isread**.

Syntax

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

<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: <table><tr><td>ISFIRST</td><td>finds the first record by positioning the starting point just before the first record.</td></tr><tr><td>ISLAST</td><td>finds the last record by positioning the starting point just before the last record.</td></tr><tr><td>ISEQUAL</td><td>finds the record equal to the search value.</td></tr><tr><td>ISGREAT</td><td>finds the first record greater than the search value.</td></tr><tr><td>ISGTEQ</td><td>finds the first record greater than or equal to the search value.</td></tr><tr><td>ISKEEPLOCK</td><td>causes isstart to keep locks held on any record in automatic locking-mode.</td></tr></table>	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.
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.												

Usage

The **isstart** function selects the index that you want to use for subsequent calls to **isread** but does not read a record in the C-ISAM file.

The key-description structure that defines the index that you want to use is *keydesc*.

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**.

To locate a record using the entire key, set the *length* to either 0 or the length of the entire key.

To locate a record using only part of the key, set the *length* to 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.

If the *mode* is ISFIRST or ISLAST, **isstart** ignores the contents of *record* and *length*.

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*.

You can use **isstart** to specify retrieval by record number when you use a key-description structure with **k_nparts**= 0. 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.

The function sets **isrecnum** to the starting record number. The contents of the current record do not change.

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.

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

Use **isuniqueid** to return a **long** integer that is guaranteed to be unique for the C-ISAM file.

Syntax

```
int isuniqueid(isfd, uniqueid)
    int isfd;
    long *uniqueid;
```

isfd is the file descriptor returned by **isopen** or **isbuild**.

uniqueid is a pointer to the **long** integer that receives the unique identifier.

Usage

The value returned by **isuniqueid** is serially incremented with each call.

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.

You must place *uniqueid* in the data record.

The **isuniqueid** call cannot be rolled back within a transaction but can be recovered.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

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

isunlock

Use **isunlock** to remove a lock on a file.

Syntax

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

isfd is the file descriptor returned by **isopen** or **isbuild**.

Important: 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

Use **iswrcurr** to write a record and make it the current record.

Syntax

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

isfd is the file descriptor returned by **isopen** or **isbuild**.

record is a pointer to the record that you want to write.

Usage

When 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.

Each index receives a key for the record. The function sets **isrecnum** to this record, which 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

Use **iswrite** to write a record to a C-ISAM file.

Syntax

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

isfd is the file descriptor returned by **isopen** or **isbuild**.
record is a pointer to the record that you want to write.

Usage

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.

Each index receives a key for the record. The current record does not change.

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 computer-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 in the following table allow you to convert between computer-dependent C language data types and the C-ISAM equivalents. They are defined on the following pages in alphabetical order.

ldchar	stchar
lddbl	stdbl
lddblnull	stdbnull
lddecimal	stdecimal
ldfloat	stfloat
ldftnull	stftnull
ldint	stint
ldlong	stlong

ldchar

Use **ldchar** to convert a character string in a C-ISAM data record to a null-terminated string.

Syntax

```
void ldchar(fstr,length,cstr);
char *fstr;
int length;
char *cstr;
```

fstr is a pointer to the starting byte of a C-ISAM character string.
length is the length of the C-ISAM character string.
cstr is the destination string in memory.

Usage

C-ISAM does not terminate a character string with a null character. Instead, it pads the string with trailing spaces. The **ldchar** function removes trailing spaces and places a null byte after the last nonblank 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

Use **lddbl** to return a computer-dependent, double-precision floating-point number from a C-ISAM DOUBLETTYPE format.

Syntax

```
double double lddbl(p)
char *p;
```

p is a pointer to the starting byte of a C-ISAM DOUBLETTYPE number.



Important: A C-ISAM DOUBLETTYPE number has the same format as a C **double** number, except that a C-ISAM number might 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

Use **lddblnull** to return a computer-dependent double-precision floating-point number from a C-ISAM DOUBLETTYPE format and simultaneously test if the value is `null`.

Syntax

```
double lddblnull(p, nullflag)
    char *p;
    short *nullflag;
```

p is a pointer to the starting byte of a C-ISAM DOUBLETTYPE number.

nullflag is a pointer to the null code.

Usage

A C-ISAM DOUBLETTYPE number has the same format as a C **double** number, except that a C-ISAM number might not be aligned on a word boundary.

When the value of the DOUBLETTYPE number is `null`, **lddblnull** sets **nullflag* to 1 and returns a 0.

When the value of the DOUBLETTYPE number is not `null`, **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],nflg);
acctbal = lddblnull(&rec[30],nflg2);
```

ldecimal

Use **ldecimal** to return a DECIMALTYPE number in a **dec_t** structure from a C-ISAM data record.

Syntax

```
int ldecimal (cp,len,decp)
    char *cp;
    int len;
    dec_t *decp;
```

cp is a pointer to the position in the data record where the decimal data starts.

len is the length of the decimal data in the data record.

decp is the **dec_t** structure that receives the decimal data.

Usage

DECIMALTYPE data is stored in a packed format within the C-ISAM file. DECIMALTYPE data must be transferred into a **dec_t** structure before the program can manipulate it.

The length parameter *len* specifies the length of the packed data and is between 2 and 17 bytes, inclusive. The packed length is the sum of the following three items:

1. The number of significant digits to the left of the decimal point, divided by 2 and rounded up
2. The number of significant digits to the right of the decimal point, divided by 2 and rounded up
3. Plus 1 byte

See “Sizing DECIMALTYPE Numbers” on page 3-14 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

Use **ldfloat** to return a computer-dependent floating-point number from a C-ISAM FLOATTYPE format.

Syntax

```
double ldfloat(p)
    char *p;
```

p is a pointer to the C-ISAM FLOATTYPE format number.

Usage

A C-ISAM FLOATTYPE number has the same format as a C **float** number, except that a C-ISAM number might not be aligned on a word boundary.

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

Use **ldfltnull** to return a computer-dependent floating-point number from a C-ISAM FLOATTYPE format and simultaneously test if the value is `null`.

Syntax

```
double ldfltnull(p,nullflag)
    char *p;
    short *nullflag;
```

p is a pointer to the starting byte of the C-ISAM FLOATTYPE format number.

nullflag is a pointer to the null code.

Usage

A C-ISAM FLOATTYPE number has the same format as a C **float** number, except that a C-ISAM number might not be aligned on a word boundary. Floating-point numbers are returned as double-precision floating point numbers.

When the value of the FLOATTYPE is `null`, the **ldfltnull** sets **nullflag* to 1, and returns a 0.

When the value of the FLOATTYPE is not `null`, the **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],nflg);
acctbal = lddblnull(&rec[30],nflg2);
```

ldint

Use **ldint** to return a computer-dependent integer from a C-ISAM INTTYPE format.

Syntax

```
short ldint(p)
char *p;
```

p is a pointer to a C-ISAM integer.

Usage

C-ISAM stores an INTTYPE integer as a 2-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

Use **ldlong** to return a computer-dependent **long** integer from a C-ISAM LONGTYPE format.

Syntax

```
long ldlong(p)
char *p;
```

p is a pointer to the C-ISAM LONGTYPE number.

Usage

C-ISAM stores a LONGTYPE integer as a 4-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

Use **stchar** to store a character string in a C-ISAM data record.

Syntax

```
void stchar(cstr,fstr,length);
char *cstr;
char *fstr;
int length;
```

<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.

Usage

C-ISAM does not terminate a character string with a null character; instead it pads the string with trailing spaces. The **stchar** 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

Use **stdbl** to store a computer-dependent double-precision number in a C-ISAM DOUBLETTYPE format.

Syntax

```
void stdbl(d,p)
    double d;
    char *p;
```

d is the double-precision number to be stored.

p is the pointer to the C-ISAM DOUBLETTYPE format that receives the number.

Usage

A C-ISAM DOUBLETTYPE number has the same format as a C **double** number, except that a C-ISAM number might 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]);
```

stdbnull

Use **stdbnull** to store a computer-dependent double-precision number or a null in a C-ISAM DOUBLETTYPE format.

Syntax

```
void stdbnull(d,p,nullflag)
    double d;
    char *p;
    short nullflag;
```

d is the double-precision number to be stored.
p is the pointer to the C-ISAM DOUBLETTYPE format that receives the number.
nullflag is the null code.

Usage

A C-ISAM DOUBLETTYPE number has the same format as a C **double** number, except that a C-ISAM number might not be aligned on a word boundary.

When you set *nullflag* to 1, a C-ISAM null is stored. When *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]);
stdbnull(acctbal,&rec[30],nflag);
```

stdecimal

Use **stdecimal** to store a DECIMALTYPE number in a **dec_t** structure into a C-ISAM record in packed format.

Syntax

```
void stdecimal (decp,cp,len)
    dec_t *decp;
    char *cp;
    int len;
```

decp is the **dec_t** structure that contains the decimal data.
cp is a pointer to the position in the data record where the decimal data starts.
len is the length of the decimal data in the data record.

Usage

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.

The length parameter *len* specifies the length of the packed data and is between 2 and 17 bytes, inclusive. The packed length is the sum of the following three items:

1. The number of significant digits to the left of the decimal point, divided by 2 and rounded up
2. The number of significant digits to the right of the decimal point, divided by 2 and rounded up
3. Plus 1 byte

For more information, see “Sizing DECIMALTYPE Numbers” on page 3-14.

Example

```
char rec[39]; /* C-ISAM Data Record */
.
.
.
/* Store Transaction Amount and Account Balance
   into Record */
stddecimal(&tramt,&rec[26],4);
stddecimal(&acctbal,&rec[30],8);
```

stfloat

Use **stfloat** to store a computer-dependent floating-point number in a C-ISAM FLOATTYPE number.

Syntax

```
void stfloat(f,p)
    float f;
    char *p;
```

- f* is the floating-point number to be stored in C-ISAM FLOATTYPE format.
- p* is the pointer to the C-ISAM FLOATTYPE format to receive the number.

Usage

A C-ISAM FLOATTYPE number has the same format as a C **float** number, except that a C-ISAM number might 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

Use **stfltnull** to store a computer-dependent floating-point number or a null in a C-ISAM FLOATTYPE number.

Syntax

```
void stfltnull(f,p,nullflag)
    float f;
    char *p;
    short nullflag;
```

- f* is the floating-point number to be stored in C-ISAM FLOATTYPE format.
- p* is the pointer to the C-ISAM FLOATTYPE format that receives the number.
- nullflag* is the null code.

Usage

A C-ISAM FLOATTYPE number has the same format as a C **float** number, except that a C-ISAM number might not be aligned on a word boundary.

When *nullflag* is set to 1, a C-ISAM null is stored; if *nullflag* is set to 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

Use **stint** to store a computer-dependent short integer in a C-ISAM INTTYPE number.

Syntax

```
void stint(i,p)
    short i;
    char *p;
```

i is the computer-dependent short integer to be stored.

p is a pointer to the C-ISAM INTTYPE number that receives the integer.

Usage

C-ISAM stores an INTTYPE integer as a 2-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

Use **stlong** to store a computer-dependent **long** integer in a C-ISAM LONGTYPE format.

Syntax

```
void stlong(l,p)
    long l;
    char *p;
```

l is the computer-dependent **long** integer.

p is the pointer to the C-ISAM LONGTYPE format that receives the number.

Usage

C-ISAM stores a LONGTYPE integer as a 4-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

dececvl and decfcvt

decadd, decsub, decmul, and decdiv

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

```
int decadd(n1, n2, result)/* result = n1 + n2 */
    dec_t *n1;
    dec_t *n2;
    dec_t *result;

int decsub(n1, n2, result)/* result = n1 - n2 */
    dec_t *n1;
    dec_t *n2;
    dec_t *result;

int decmul(n1, n2, result)/* result = n1 * n2 */
    dec_t *n1;
    dec_t *n2;
    dec_t *result;

int decdiv(n1, n2, result)/* result = n1 / n2 */
    dec_t *n1;
    dec_t *n2;
    dec_t *result;
```

<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.

Usage

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

Use **deccmp** to compare two DECIMALTYPE numbers.

Syntax

```
int deccmp(n1, n2)
    dec_t *n1;
    dec_t *n2;
```

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

Use **deccopy** to copy one **dec_t** structure to another.

Syntax

```
void deccopy(n1, n2)
    dec_t *n1;
    dec_t *n2;
```

n1 is a pointer to the source **dec_t** structure.

n2 is a pointer to the destination **dec_t** structure.

deccvasc

Use **deccvasc** to convert a value held as printable characters in a C **char** type into a DECIMALTYPE number.

Syntax

```
int deccvasc(cp, len, np)
char *cp;
int len;
dec_t *np;
```

cp points to a string that holds the value that you want to convert.
len is the length of the string.
np is a pointer to a **dec_t** structure that receives the result of the conversion.

Usage

The **deccvasc** function ignores leading spaces in the character string. The character string can have a leading plus (+) or minus (-) sign, a decimal point (.), and numbers to the right of the decimal point. The character string can contain an exponent preceded by either *e* or *E*. The exponent can 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);
```

deccvdbl

Use **deccvdbl** to convert a C type **double** into a DECIMALTYPE number.

Syntax

```
int deccvdbl(dbl, np)
double dbl;
dec_t *np;
```

dbl is a double-precision floating-point number.

np 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);
```

deccvflt

Use **deccvflt** to convert a C type **float** into a DECIMALTYPE number.

Syntax

```
int deccvflt(flt, np)
float flt;
dec_t *np;
```

flt is a floating-point number.

np 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);
```

deccvint

Use **deccvint** to convert a C type **short** into a DECIMALTYPE number.

Syntax

```
int deccvint(integer, np)
    int integer;
    dec_t *np;
```

integer is the integer that you want to convert.

np 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);
```

deccvlong

Use **deccvlong** to convert a C type **long** value into a DECIMALTYPE number.

Syntax

```
int deccvlong(lng, np)
long lng;
dec_t *np;
```

lng is a pointer to a long integer.

np 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);
```

dececv and decfcvt

These functions convert a DECIMALTYPE value to an ASCII string. The **dececv** function is the DECIMALTYPE equivalent of UNIX **ecvt(3)**, and **decfcvt** is the DECIMALTYPE equivalent of UNIX **fcvt(3)**.

Syntax

```
char *dececv(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;
```

<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 dececv , 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.

Usage

The **dececv** 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.

The low-order digit of the DECIMALTYPE number is rounded.

The **decfcvt** function is identical to **dececv**, 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*:

```
dececvf(4)  = "1235"          *decpt = 5  
dececvf(10) = "1234567000"   *decpt = 5
```

```
decfcvf(1)  = "123457"        *decpt = 5  
decfcvf(3)  = "12345670"      *decpt = 5
```

In this example, let *np* point to .001234:

```
dececvf(4)  = "1234"          *decpt = -2  
dececvf(10) = "1234000000"    *decpt = -2
```

```
decfcvf(1)  = ""              *decpt = -2  
decfcvf(3)  = "1"             *decpt = -2
```

dectoasc

Use **dectoasc** to convert a DECIMALTYPE number to a printable ASCII string.

Syntax

```
int dectoasc(np, cp, len, right)
    dec_t *np;
    char *cp;
    int len;
    int right;
```

<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 that indicates the number of decimal places to the right of the decimal point.

Usage

If *right* equals -1, the number of decimal places is determined by the decimal value of **np*.

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.

The string returned by **dectoasc** is *not* null terminated.

Return Codes

-1	Error; iserrno contains the error code
0	Successful

Example

```
#include <decimal.h>

char input[80];
char output[17];
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);

/* Null terminate the output string */
output[16] = '\0';

/* Print the value just entered */
printf("You just entered %s", output);
```

dectodbl

Use **dectodbl** to convert a DECIMALTYPE number into a C type **double**.

Syntax

```
int dectodbl(np, dblp)
    dec_t *np;
    double *dblp;
```

np is a pointer to a decimal structure.

dblp is a pointer to a double-precision floating-point number that receives the result of the conversion.

Usage

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);
```

dectoflt

Use **dectoflt** to convert a DECIMALTYPE number into a C type **float**.

Syntax

```
int dectoflt(np, fltp)
    dec_t *np;
    float *fltp;
```

np is a pointer to a decimal structure.

fltp is a pointer to a floating-point number to receive the result of the conversion. 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);
```

dectoint

Use **dectoint** to convert a DECIMALTYPE number into a C **int** type.

Syntax

```
int dectoint(np, ip)
    dec_t *np;
    int *ip;
```

np is a pointer to a decimal structure whose value is converted to an integer.

ip 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);
```

dectolong

Use **dectolong** to convert a DECIMALTYPE into a C type **long**.

Syntax

```
int dectolong(np, lngp)
    dec_t *np;
    long *lngp;
```

np is a pointer to a decimal structure.

lngp is a pointer to a long integer 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);
```

Summary

This chapter describes all the functions that are available as part of C-ISAM, including:

- file-manipulation functions.
- format-conversion and format-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 that are associated with maintaining C-ISAM files

The format-conversion functions allow you to convert between computer-dependent representation of numbers and their 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.

C-ISAM Utilities

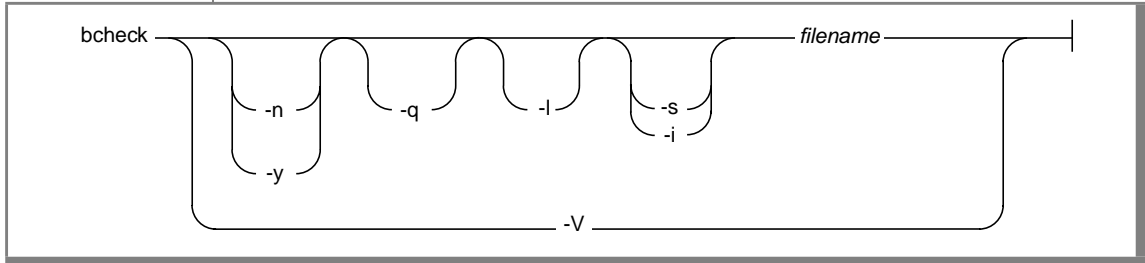
The **bcheck** Utility

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.



Element	Purpose	Key Considerations
-i	Checks index file only.	References: For information about checking indexes, see “Checking Indexes with the -i Option” on page A-3.
-l	Lists entries in B+ trees.	None.
-n	Responds negatively to all prompts.	Additional information: When you know in advance that all your responses to bcheck prompts are negative, specify the -n option. References: For information about not specifying the -n option, see “Choosing Not to Specify the -n or -y Option” on page A-3.
-q	Suppresses printing of the program banner.	None.
-s	Converts an index file from its existing node size to the current computer hardware node size.	References: For information about resizing the keys within an index, see “Resizing Nodes and Indexes with the -s Option” on page A-4.
-V	Displays software version information.	None.
-y	Responds affirmatively to all prompts.	Restrictions: Do not use the -y option with bcheck when you are checking the files for the first time. References: For information about not specifying the -y option, see “Choosing Not to Specify the -n or -y Option” on page A-3.
<i>filename</i>	Specifies the name of the index file that you want checked without the .idx extension.	None.

GLS



Unless you use the **-n** or **-y** option, **bcheck** is interactive, waiting for you to respond to each error that it finds.

For information about using **bcheck** to check indexes built on a specific collation sequence, see “Checking Localized Indexes with the bcheck Utility” on page B-12. ♦

***Warning:** Use the **-y** option with caution. Do not run **bcheck** using the **-y** option if you are checking the files for the first time.*

Choosing Not to Specify the -n or -y Option

When you do not know your responses to **bcheck** prompts in advance, you can choose to not specify the **-n** or **-y** option. However, when you do not specify the **-y** or **-n** options, responding to prompts can take a long time.

When you do not specify the **-n** or **-y** option, **bcheck** executes and prompts you interactively. The prompts from **bcheck** request confirmation that you want to re-create the index when **bcheck** finds bad entries. To repair indexes, **bcheck** reads all data from the **.dat** file and re-creates the index in the **.idx** file.

Checking Indexes with the -i Option

If you use the **-i** option with fixed-length records, **bcheck** checks the index information in the index files for consistency with the data files.

If you use the **-i** option with variable-length records, **bcheck** checks the entire contents of the index file for free space as well as for consistency. The **bcheck** utility also checks 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 with the -s Option

The **-s** option resets the **NODESIZE** parameter from the existing value to the current machine node size. Use this option after you move a table to a computer with a different node size. If you are running an application, an error message that indicates a wrong node size alerts you to node-size problems.

The **-s** option does not change any of the characteristics of the index keys themselves or 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 receive the following two messages with relevant values in addition to 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. Because 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 must delete corrupted records with your own C-ISAM program. See “File Maintenance with Variable-Length Records” on page 6-11 for more information about retrieving data from corrupted **.idx** files.

Examples 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 7.2
Copyright (C) 1981-1996 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
```

The following example shows a sample run where **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 7.2
Copyright (C) 1981-1996 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

ERROR: 3 bad data record(s)
Delete index ? no
```

Examples Using bcheck

```
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
```

Because **bcheck** finds errors, you must delete and rebuild the corrupted indexes. The **-y** option is used to answer yes to all questions that **bcheck** asks, as shown in the following example:

```
BCHECK C-ISAM B-tree Checker version 7.2
Copyright (C) 1981-1996 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
```

The GLS Environment

This appendix provides information about the Global Language Support (GLS) environment. The GLS locales and files are described in the following topics:

- GLS locales
- Locale names
- Locale files
- Code-set files
- The Informix **registry** file

This appendix also includes the following topics, which describe how to use C-ISAM in the GLS environment:

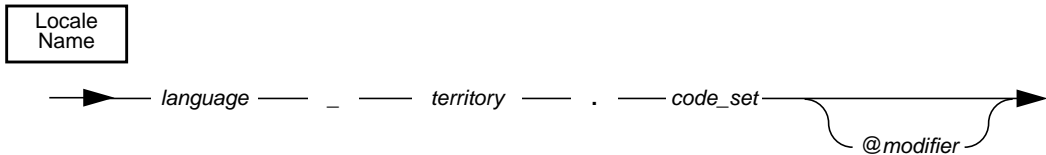
- How to specify a locale
- How to set the **CLIENT_LOCALE** environment variable
- Collation orders for C-ISAM files
- How to add a localized index
- How to determine an index-collation sequence
- How to use the **bcheck** utility to check localized indexes
- The **glfiles** utility
- How to generate a list of GLS locales and code-set files

GLS Locales

With the GLS feature, C-ISAM can support different languages, code sets, and collation sequences. All culture-specific information is combined in a single environment, which is called a *GLS locale*. A locale provides the information that C-ISAM uses for sorting (collation) and comparing data in a file. To use a specific collation sequence on a C-ISAM file, you must set a GLS locale in your runtime environment.

Locale Names

The *locale name* identifies the particular GLS locale file that your C-ISAM program uses at runtime. The *locale* represents the basic language and cultural conventions that are relevant to the processing of data for a given language and territory. You use a locale name to set the **CLIENT_LOCALE** environment variable when you want to specify a collation sequence on a C-ISAM file. A GLS locale name has the following form.



Element	Purpose	Key Considerations
<i>code_set</i>	Specifies the name of the computer code set that the locale uses.	None.
<i>language</i>	Specifies the two-character name that represents the language for a specific locale.	None.
<i>modifier</i>	Specifies an optional locale modifier that has a maximum of four alphanumeric characters.	Additional information: This specification modifies the default cultural-convention settings that are implied in the <i>language_territory</i> setting. For example, you might set <i>@modifier</i> to specify a locale that contains dictionary or telephone-book sorting order.
<i>territory</i>	Specifies a two-character name that represents the cultural conventions. For example, <i>territory</i> might specify the Swiss version of the French, German, or Italian language.	None.

The default locale, U.S. English with the 8859-1 code set, has the following name:

```
en_us.8859-1
```

where **en** indicates the English language, **us** indicates the United States territory, and **8859-1** indicates the name of the ISO8859-1 code set. An example of a locale name for a French-Canadian locale follows:

```
fr_ca.ISO8859-1
```

The following example specifies this same locale with a custom collation sequence for dictionary sorting:

```
fr_ca.ISO8859-1@dict
```

Locale Files

The *locale file* defines a GLS locale. It describes the basic language and cultural conventions that are relevant to the processing of data for a given language and territory. By default, C-ISAM uses the locale file that **en_us.8859-1** specifies for locale-sensitive processing. When you build an index with a specific collation sequence, you set the **CLIENT_LOCALE** environment variable to a locale name that identifies the locale file that a C-ISAM program uses at runtime.

Each locale file has the following two forms:

- A source locale file is an ASCII file that contains the six locale categories. This file has the **.lc** file extension and serves as documentation for the corresponding object file.
- An *object* locale file is used by Informix products at runtime to obtain locale information quickly. Object locale files have the **.lco** file extension.

GLS locale files reside in subdirectories of the **\$INFORMIXDIR/gls/lcX** directory, where **\$INFORMIXDIR** is the directory in which your Informix product is installed and **X** is the version number for the locale object format. These subdirectories have names of the format *lg_tr*, where *lg* is the two-character language name, and *tr* is the two-character territory name. The object and source versions of locale files available for a particular language and territory reside in these locale subdirectories.

Code-Set Files

A given language has a *character set* of one or more natural-language alphabets together with additional symbols for digits, punctuation, and diacritical marks. Each character set has at least one *code set*, which maps its characters to unique bit patterns. ASCII, ISO 8859-1, and EBCDIC are examples of code sets for the English language.

The number of unique characters in the language determines the amount of storage that each code-set character requires. Because a single byte can store values in the range 0 to 255, it can uniquely identify 256 characters. Most Western languages have fewer than 256 characters and therefore have code sets made up of single-byte characters. When an application handles data in such code sets, it can assume that one character is always stored in 1 byte.

A C-ISAM *code-set file* (also called a *charmap* file) defines a code set for use by locale files. A GLS locale includes the appropriate code-set file for the code set that it supports.

Each code-set file has the following two forms:

- The *source* code-set file is a text file that describes the characters of a character set. This file has a **.cm** extension and serves as documentation for the corresponding object file.
- The *object* code-set file is used to create locale object files. Object code-set conversion files have a **.cmo** file extension.

C-ISAM includes only the source version of code-set files (**.cm**). Use these files as on-line documentation for the locales that use them.

C-ISAM only uses the object code-set conversion files (**.cmo**) to compile locale files (**.lc**). The object code-set conversion files are not needed at runtime.

GLS code-set files reside in the **\$INFORMIXDIR/gls/cmZ** directory, where **\$INFORMIXDIR** is the directory in which C-ISAM is installed and **Z** is the version number for the code-set object format.

The Informix registry File

The Informix Code-Set Name Mapping File, which is called **registry**, is a text file that associates code-set names and synonyms with their code-set numbers. C-ISAM uses the **registry** file to map a locale specification to a filename. For example, when you set `CLIENT_LOCALE = en_us.8859-1`, the **registry** file converts this value to **en_us/0333.lco**, which is the actual filename of the locale.

Refer to the comments at the top of the **registry** file for information about the file format and search algorithm that Informix products use to convert code-set names to code-set numbers.

The **registry** file is located in the **\$INFORMIXDIR/gls/cmZ** directory, where **\$INFORMIXDIR** is the directory in which C-ISAM is installed and **Z** is the version number for the code-set object format.

C-ISAM in the GLS Environment

The previous section describes the GLS environment. This section shows you how to specify GLS locales with C-ISAM, how to create an index with a specific collation sequence, and how to use the **bcheck** utility to check an index built on a specific collation sequence.

Specifying a Locale

By default, C-ISAM uses the locale **en_us.8859-1**. However, when you want to use a customized version of U.S. English, or British English, or another language, you must set the locale that specifies the collation sequence that you want to use. You can use the **glfiles** utility to generate a list of the locales that C-ISAM supports. The **glfiles** utility is described on page B-12.

Important: You must always set the **INFORMIXDIR** environment variable before you run a C-ISAM program. C-ISAM uses **INFORMIXDIR** to locate the locale that it uses for locale-sensitive processing.

To specify a nondefault locale

1. Set the **INFORMIXDIR** environment variable to the path where C-ISAM is installed. (See “Setting the INFORMIXDIR Environment Variable” on page 1-36.)
2. Determine the locale name for the locale that you want C-ISAM to use.

For information about determining the locale that is associated with an existing index, see “Determining Index-Collation Sequence” on page B-11. To generate a list of the GLS locales that C-ISAM supports, see “Generating a List of GLS Locales” on page B-14.

3. Set the **CLIENT_LOCALE** environment variable to the locale name that specifies the locale that you want to use.

Important: Set the **INFORMIXDIR** and **CLIENT_LOCALE** environment variables before you run a C-ISAM program that uses an index with a specific collation sequence.



Setting the *CLIENT_LOCALE* Environment Variable

The **CLIENT_LOCALE** environment variable specifies the locale that C-ISAM uses to determine the collation sequence of an index.

CLIENT_LOCALE

Locale
Name
p. B-2

For example, to specify a German locale with a custom collation sequence for phone-book sorting, you might set **CLIENT_LOCALE** as follows:

```
setenv CLIENT_LOCALE de_de.IS08859-1@phon
```

When you do not explicitly set **CLIENT_LOCALE**, C-ISAM uses the default locale, **en_us.8859-1**.

Collation Order of Characters in a C-ISAM File

C-ISAM uses a GLS locale to control the behavior of collation and string comparisons. Collation involves the sorting of character data that C-ISAM stores. A GLS locale contains a **COLLATION** category, which lists all characters, in the order in which they sort. C-ISAM, Version 7.2, supports the following two methods of sorting character data:

- *Code-set order* refers to the intrinsic order of characters within a code set.

The order of the character codes in the code-set file determines the sort order. For example, in the ASCII code set, A=65 and B=66. A always sorts before B because 65 is less than 66.

- *Localized order* refers to an order of the characters that relates to a real language.

The order of the characters in the **COLLATION** category of the locale file determines the sort order. For example, even though the character Å might have a code-set code of 133, the locale file would list this character after A and before B (A=65, Å=133, B=66).

The collation order that a C-ISAM file uses depends on the data type of the key that the index uses. The following table summarizes these collation orders.

Data Types	Collation Order
CHARTYPE	Code-set order
NCHARTYPE	Localized order

The difference in collation order is the only distinction between the CHARTYPE and NCHARTYPE data types. An index that uses a specific collation sequence is referred to as a *localized index*. A localized index is always built on an NCHARTYPE character field.

NLS

Index files created with earlier versions of C-ISAM (before Version 7.2), use the same collation orders as C-ISAM, Version 7.2, as follows:

- Code-set order for indexes that designate character fields as CHARTYPE
- Localized order for indexes that designate character fields as NCHARTYPE.



Tip: In the *C-ISAM Programmer's Manual, Version 6.0 and 7.1*, indexes built on an NCHARTYPE character field are referred to as *NLS indexes*. ♦

Creating Localized Indexes

Localized indexes are indexes that use a specific collation sequence, usually something other than U.S.English. When you create an index on an NCHARTYPE character field, the index keys are stored in localized collation order as opposed to code-set order.

You can build a localized index into a file using the **isbuild** function or you can add a localized index later using the **isaddindex** function, just as with a nonlocalized index.



To create a localized index

1. Set the **CLIENT_LOCALE** environment variable to the locale that specifies the collation sequence on which the index is built.
2. Designate the key that the index is built on as **NCHARTYPE**.
3. Call the **isbuild** or **isaddindex** function.

***Important:** All the indexes associated with a file must use the same collation sequence.*

An Example of Adding a Localized Index

To create an additional localized index on a file that uses a French locale with a specific collation sequence for phone-book sorting, you might set the locale environment variable as follows:

```
setenv CLIENT_LOCALE fr_fr.8859-1@phon
```

Figure B-1 shows the definition of a key structure for building an additional localized index on the **employé** file, a French version of the **employee** file described in Chapter 1. To create the localized index, you must set the `nlkey.k_part[0].kp_type` member of the **keydesc** structure to **NCHARTYPE** for this index (**nlkey**), and call **isaddindex** to create the index. This code is a variation of the code in the **add_idx.c** demonstration program.

```
#include <isam.h>
:
:
struct keydesc nlnkey;
int fdemploy;

/*This program adds a secondary index for the last name field
 * in the employ   file, with a localized collation sequence.*/

main()
{
int cc;

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

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

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

Figure B-1
*Adding a Localized
Index to a C-ISAM
File*



Important: If you set **CLIENT_LOCALE** to an invalid locale or collation sequence, C-ISAM uses default settings and might build the index with the wrong locale.

Determining Index-Collation Sequence

You can use the **islanginfo** function to determine what locale is associated with a localized index. For a description of this function, see “islanginfo” on page 8-38.

The program shown in Figure B-2 calls the **isglsversion** function to check the **employee** file and determine if any of the indexes are localized indexes. If **isglsversion** is true, the program calls the **islanginfo** function to identify the locale that C-ISAM uses to build the index. To run a C-ISAM program that modifies records that are associated with the localized index, set the **CLIENT_LOCALE** environment variable to the locale that specifies the collation sequence of the index.

```
#include <isam.h>
#define SUCCESS 0

int cc;

main()
{
    printf("\nThis program determines if the employee\n");
    printf("file has localized indexes associated with it. \n\n");

    /* determine if file was built with a specific collation
    sequence*/

    cc = isglsversion("employee");
    if (cc < SUCCESS)
    {
        printf("isglsversion error %d for employee file\n",
            iserrno);
        exit(1);
    }
    if (cc == 1)
    {
        printf("\nemployee file HAS LOCALIZED indexes \n");
        /* use islanginfo to get the value of lang for index */
        printf("The LANG of the index is %s\n",
            islanginfo("employee") );
        exit(1);
    }
    if (cc == 0)
        printf("\nemployee file DOES NOT HAVE LOCALIZED indexes
        \n");

    /* isclose(fdemploy); */
}
```

Figure B-2
Determining
Whether a Key
Includes
NCHARTYPE and
Identifying Its
Locale

NLS

*For backward compatibility with C-ISAM, Version 6.x and 7.x programs, C-ISAM Version 7.2 continues to support the **isnlsversion** function. However, you can use the **isglsversion** function to check whether a file has a localized index associated with it (either GLS or NLS). ♦*

Checking Localized Indexes with the bcheck Utility

You can use the **bcheck** utility on files that use localized indexes. The **bcheck** utility verifies that your locale is the same as that of the index file.

To run the bcheck utility on a localized index

1. Set the **INFORMIXDIR** environment variable to the directory where C-ISAM is installed.
2. Set the **CLIENT_LOCALE** environment variable to the locale that specifies the collation sequence that you want to use with the index.
3. Construct a **bcheck** command with the utility options that you want.

NLS

When using **bcheck** to check files that use localized indexes, the **CLIENT_LOCALE** environment variable must be set to the locale that was used to build the index. For information on using the **islanginfo** function to identify the locale associated with an index, see “Determining Index-Collation Sequence” on page B-11. ♦

If you use **bcheck** on localized indexes without first setting **INFORMIXDIR** and **CLIENT_LOCALE**, **bcheck** will not run and displays an error message.

The glfiles Utility

You can use the **glfiles** utility to find out what GLS-related files are available with C-ISAM, Version 7.2. To comply with DOS 8.3 naming conventions, C-ISAM uses condensed filenames to store GLS locales. These filenames, therefore, do not match the locale names that you use to set **CLIENT_LOCALE**. Use the **glfiles** utility to generate a more readable list of the following GLS-related files:

- The GLS locales that are available on your system
- The Informix code-set files that are available on your system

Before you run **glfiles**, take the following steps:

1. Set the **INFORMIXDIR** environment variable to the directory in which you have installed C-ISAM.
If you do not set **INFORMIXDIR**, **glfiles** checks the **/usr/informix** directory for the GLS files.
2. Change directories to the directory where you want the **glfiles** output files to reside.
The utility creates the GLS file listings in the current directory.

glfiles

-lc

-cm

Element	Purpose	Key Considerations
-lc	Creates a file that lists the names of the locales that are available on your system.	References: For information about the file that the glfiles utility creates when you use the -lc option, see “Generating a List of GLS Locales” on page B-14.
-cm	Creates a file that lists the names of the code sets that are available on your system.	References: For information about the file that the glfiles utility creates when you use the -cm option, see “Generating a List of Code-Set Files” on page B-16.

If you specify no options, the command is interpreted as **glfiles -lc** and **glfiles** creates a file that lists the names of the locales that are available on your system.

Generating a List of GLS Locales

When you run **glfiles** with the **-lc** option, the utility creates a file that lists the available GLS locales. For each **lcX** subdirectory in **\$INFORMIXDIR/gls**, **glfiles** creates a file in the current directory called **lcX.txt**, where **X** is the version number of the locale object file format. The **lcX.txt** file lists the locales in alphabetical order, sorted on the name of the GLS object locale file.

Figure B-3 shows a sample file, **lc9.txt**, with the format for GLS locales.

```
Filename: lc9/cs_cs/0354.lco
Language: unknown
Territory: unknown
Modifier: unknown
Code Set: 852
Locale Name: cs_cs.852
.
.
.
Filename: lc9/en_us/0333.lco
Language: English
Territory: United States
Modifier:
Code Set: 8859-1
Locale Name: en_us.8859-1
.
.
.
Filename: lc8/ja_jp/e006.lco
Language: Japanese
Territory: Japan
Code Set: sjis
Locale Name: ja_jp.sjis
```

Figure B-3
*Sample Format for
GLS Locales*

NLS

The **lcX.txt** file also contains GLS locales that are compatible with many operating-system locales. Figure B-4 shows the sample format in **lc9.txt** for these locales. C-ISAM provides these compatible locales for backward compatibility with existing C-ISAM files that use localized indexes. These compatible locales match the locale names that were used to specify the collation sequence on a localized index. ♦

```
Filename: lc9/os/C
Locale Name: C

Filename: lc9/os/POSIX
Locale Name: POSIX

Filename: lc9/os/de
Locale Name: de

Filename: lc9/os/en_US
Locale Name: en_US

Filename: lc9/os/fr
Locale Name: fr
.
.
.
```

Figure B-4
*Sample Format for
GLS Locales
Compatible with
Operating-System
Locales*

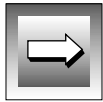
Examine the **lcX.txt** file(s) to determine the GLS locales supported by the **\$INFORMIXDIR/gls/lcX** directory on your system.

Generating a List of Code-Set Files

When you run **glfiles** with the **-cm** option, the utility creates a file that lists the available code-set files. For each **cmZ** subdirectory in **\$INFORMIXDIR/gls**, **glfiles** creates a file in the current directory called **cmZ.txt**, where **Z** is the version number of the code-set object file format. The **cmZ.txt** file lists the code sets in alphabetical order, sorted on the name of the GLS object code-set file.

Important: *C-ISAM contains only the source versions of code-set files.*

Figure B-5 shows a sample file, **cm3.txt**, with the format for code sets.



```
Filename: cm3/0333.cm
Code Set: 8859-1

Filename: cm3/0352.cm
Code Set: 850
```

```
.
```

Figure B-5
*Sample Format for
C-ISAM
Code-Set Files*

Examine the **cmZ.txt** file to determine the code sets supported by the **\$INFORMIXDIR/gls/cmZ** directory on your system.

Error Codes

This appendix lists error codes and status information returned by C-ISAM calls.

Status Information Bytes

Four bytes, **isstat1**, **isstat2**, **isstat3**, and **isstat4**, return status information after C-ISAM calls. These bytes are used primarily 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**. Figure C-1 lists the values of **isstat1**.

Figure C-1
isstat1 Values

isstat1 Value	Description
0	Successful completion
1	End of file
2	Invalid key
3	System error
9	User-defined errors

Figure C-2 shows the values of **isstat2** based on the **isstat1** status code.

Figure C-2
isstat2 Values Based on isstat1 Value

isstat1 Value	isstat2 Value	Indication
0 - 9	0	No further information is available.
0	2	Duplicate key found. After a read, this value 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 value 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 isstat2 is defined by the user.

Figure C-3 explains the combinations of **isstat3** and **isstat4** values.

Figure C-3
Combinations of isstat3 and isstat4 Values

isstat3 Value	isstat4 Value	Indication
0	0	Successful completion; no further information is available.
0	2	Successful completion; duplicate key found. After a read, this value 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 value 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.
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.

(1 of 2)

isstat3 Value	isstat4 Value	Indication
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.

(2 of 2)

Status Codes and ISAM Errors

Figure C-4 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**.

Figure C-4
ISAM Errors and isstat Values

Name	Number	Description	isstat1 Value	isstat2 Value	isstat3 Value	isstat4 Value
EDUPL	100	An attempt was made to add a duplicate value to an index with 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	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	
ELOCKED	107	The record or file requested by this call cannot be accessed because another user has locked it.	9		9	

(1 of 3)

Name	Number	Description	isstat1 Value	isstat2 Value	isstat3 Value	isstat4 Value
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 cannot 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		
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		

(2 of 3)

Name	Number	Description	isstat1 Value	isstat2 Value	isstat3 Value	isstat4 Value
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		

(3 of 3)

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 that C-ISAM index files contain. 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 following sections describe these and other nodes.

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	1	Localized index flag, true or false	
20	5	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 next record in data file	
37	4	Index-node number of next node in index file	
41	4	Transaction number	
45	4	Unique id	
49	4	Pointer to audit-trail information	
53	2	Locking method	

(1 of 2)

Byte Offset	Number of Bytes	Item	Value
55	4	Free group 0 hash pointer	
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	
75	36	Collation sequence used if file has localized index	

(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	
13	2	Length of key part 1 (top bit = duplicates)	
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


Repeats for each key

Repeats for each part of the key

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 (might be compressed)	 Repeats for each key entry
4+k	2	For duplicate key (if compressed)	
6+k	4	Pointer to data record (top bit might duplicate flag)	
N-2	1	Index tree number	
N-1	1	Level in tree	0 = leaf node

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 that describe the length and placement of the variable-length portion.

If the flag is equal to 0 (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 that consist 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 2-byte entry that indicates the actual length of the data record. (This entry is not used in audit trails for fixed-length record files.) Figure D-2 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-images and after-images are 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 on 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, 1/1/70
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_LEN0/* current record length */
#define LG_TYPELG_LEN+INTSIZE/* log record type */
#define LG_XIDLG_TYPE+2/* transaction id */
#define LG_USERLG_XID+INTSIZE/* user name */
#define LG_TIMELG_USER+2/* transaction time */
#define LG_PREVLG_TIME+LONGSIZE/* previous log record */
#define LG_PREVLENLG_PREV+LONGSIZE/* previous log length */

/* BEGIN, COMMIT, and ROLLBACK WORK record definition */
#define LG_TXSIZELG_PREVLEN+INTSIZE+INTSIZE
/* record size */

/* build file record definition */
#define LG_FMODELG_PREVLEN+INTSIZE/* build mode */
```

```
#define LG_RECLENLG_FMODE+INTSIZE/* minimum record length */
#define LG_MAXLENLG_RECLEN+INTSIZE/* max rec length or
zero */
#define LG_KFLAGSLG_MAXLEN+INTSIZE/* key flag */
#define LG_NPARTSLG_KFLAGS+INTSIZE/* number of key parts */
#define LG_KLENLG_NPARTS+INTSIZE/* total key length */

/* erase file record definition */
#define LG_FNAMELG_PREVLEN+INTSIZE/* directory path name */

/* rename file record definition */
#define LG_OLENLG_PREVLEN+INTSIZE/* length of old filename */
#define LG_NLENLG_OLEN+INTSIZE/* length of new filename */
#define LG_ONAMELG_NLEN+INTSIZE/* old filename */

/* open and close file record definition */
#define LG_ISFDLG_PREVLEN+INTSIZE/* isfd of file */
#define LG_VARLENLG_ISFD+INTSIZE/* VARLEN flag of file */
#define LG_FPATHLG_VARLEN+INTSIZE/* directory path name */

/* create and drop index */
#define LG_IFLAGSLG_ISFD+INTSIZE/* key flags */
#define LG_INPARTSLG_IFLAGS+INTSIZE/* number of key parts */
#define LG_IKLENLG_INPARTS+INTSIZE/* total key length */

/* set unique id */
#define LG_UNIQIDLG_ISFD+INTSIZE/* new unique id */

/* before or after image record definition */
#define LG_RECNOLG_ISFD+INTSIZE/* record number */
#define LG_IMGLENLG_RECNO+LONGSIZE/* record image length */
#define LG_RECORDLG_IMGLEN+INTSIZE/* record data */

/* update image (before and after together) */
#define LG_BEFLNLG_RECNO+LONGSIZE/* length of before image*/
#define LG_AFTLENLG_BEFLN+INTSIZE/* length of after image*/
#define LG_BUPDTELG_AFTLEN+INTSIZE
/* before image for update*/
/* (followed by the afterimage) */

/* savepoint record */
#define LG_SAVEPTLG_PREVLEN+INTSIZE/* savepoint number */
#define LG_SIZELG_SAVEPT+INTSIZE/* record size */

/* log memo record */
#define LG_LOCATIONLG_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_ERROR0/* log read or write error */
#define LG_BEGWORK1/* BEGIN WORK */
#define LG_COMWORK2/* COMMIT WORK */
#define LG_ROLWORK3/* ROLLBACK WORK */
#define LG_DELETE4/* deleted record */
#define LG_INSERT5/* newly inserted record */
#define LG_UPDATE6/* updated record */
#define LG_VERSION7/* version */
#define LG_SVPOINT8/* savepoint */
#define LG_FOPEN9/* open file */
#define LG_FCLOSE10/* close file */
#define LG_CKPOINT11/* checkpoint */
#define LG_BUILD12/* build new file */
#define LG_ERASE13/* erase old file */
#define LG_RFORWARD14/* ROLLFORWARD */
#define LG_CINDEX15/* create index */
#define LG_DINDEX16/* drop index */
#define LG_EOF17/* end of log file */
#define LG_RENAME18/* rename file */
#define LG_SETUNIQID19/* set unique id */
#define LG_UNIQUEID20/* get unique id */
#define LG_RBSVPT21/* rollback to savepoint */
#define LG_CLUSIDX 22 /* create cluster index */
#define LG_MEMO 23 /* log file memo record */

#define TRUE1
#define FALSE0

#define NOPNFL 16

/* "smart" recovery flags */
#define R_NOTRANS 0x01 /*recover records outside
of transx */
#define R_FERRORS 0x02 /* file errors */
^L
/*
* recovery opens
* iopens is used with is logging
* during recovery, it keeps a list of all files open by all
processes
*/

```

```
struct iropen
{
    short iro_risfd; /* recovery open isfd */
    short iro_cnt; /* recovery - # of txs using */
    short iro_flag; /* flags - see below */
    char *iro_path; /* dir path for file */
    struct iropen *iro_next; /* next in list */
};

/* define for iro_flag
 */
#define IRO_CURRTX 001 /* is file open for curr tx */
#define IRO_VARLEN 002 /* file has var len records */

struct txlist
{
    int tx_xid; /* transaction id */
    int tx_flag; /* transaction flags */
    struct xrloc *tx_nextrec; /* next log rec in transaction */
    struct txlist *tx_next; /* next transaction */
    struct txo *tx_opens; /* open list */
};

/* values for tx_flag */

#define TX_BEGWORK 0x1 /* begin work found for transx */

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 */
};

struct txo
{
    int txo_ofd; /* original transactions fd */
    struct iropen *txo_ro; /* pointer into openefile */
    struct txo *txo_next;
};
```

System Administration

This appendix discusses the following topics:

- Installation issues
- Removing GLS files to save disk space
- Migrating C-ISAM files
- System administration facilities

Use this appendix 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 ISAMBUFS parameter for buffered input and output.

Files

Your installation media for the C-ISAM system contains several program files that the commands in **installisam** installs. (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 described in the following table:

File	Description
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.”)
libisam.a	Is used whenever you compile a program that uses C-ISAM files. (See “Compiling Your C-ISAM Program” on page 1-35 for compilation instructions.)
.lco (locale object files)	Are used for locale-sensitive processing at runtime.

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 operating-system buffers to reduce the number of disk I/O operations that are required during the 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 C-ISAM 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.

The size of each buffer is 1 kilobyte. The default ISAMBUFS value is 16. Typically, 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 total number of buffers that you can allocate is the number equal to the maximum small integer on your system.

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 that you want to use (for example 4, 16, or some other number).

GLS

Saving Disk Space

C-ISAM automatically installs GLS locale and code-set files that allow you to build indexes for different collation sequences. The locale and code-set files represent the particular languages and territories and define different collation sequences that you can use. The sections that follow describe the GLS files that you might remove to increase your available disk space.

Removing Locale Files

To save disk space, you might want to keep only the locale files that you intend to use. You can safely remove the following GLS files from your C-ISAM product:

- Locale source files (.lc)
- Locale object files (.lco)

These files can be removed from the subdirectories of **\$INFORMIXDIR/gls/lcX** for the locales that you do not intend to use.

Warning: Do not remove the object locale file for the U.S. ASCII English locale, **\$INFORMIXDIR/gls/lcX/en_us/0333.lco**. In addition, do not remove the Informix Code-Set Name Mapping file, **registry**. C-ISAM uses these files for the language processing of all locales.

Because C-ISAM does not access source versions of locale files, you can safely remove them. However, these files do provide useful on-line documentation for the supported locales. If you have enough disk space, Informix recommends that you keep these source files for the GLS locales that are supported.

For more information about the GLS locale files, see “Locale Files” on page B-4.



Removing Code-Set Files

C-ISAM Version 7.2 includes only the source version of code-set files (.cm). C-ISAM provides these files as on-line documentation for the locales that use them. Because C-ISAM does not access source code-set files, you can safely remove them. However, if you have enough disk space, Informix recommends that you keep these source files for the GLS locales that are supported.

For more information on code-set files, see “Code-Set Files” on page B-4. ♦

Migrating C-ISAM Files

When you migrate C-ISAM applications, be aware of changes in the way you specify locale information.

Migrating Version 6.0 and 7.1 Applications

C-ISAM, Version 7.2, files are compatible with Version 6.0, or 7.1 applications. However, to manipulate files that use localized indexes, you must link your application with C-ISAM, Version 7.2, libraries and set the locale environment before you run the application.

To migrate a 6.0 or 7.1 application that uses a localized index

1. Link the application with C-ISAM, Version 7.2, libraries. For example, if you installed C-ISAM using the default directories, you might use the following command line:

```
cc filename.c -lisam
```
2. Set the **INFORMIXDIR** environment variable to the directory where C-ISAM is installed.
3. Set the **CLIENT_LOCALE** environment variable to the locale that specifies the collation sequence.

When you set **INFORMIXDIR** and **CLIENT_LOCALE**, C-ISAM can locate and use the locale that it needs for locale-sensitive processing. A Version 6.0 or 7.1 application can still call the **setlocale** function. The **setlocale** function might affect operating-system functions such as **isalpha**, but it does not affect the behavior of C-ISAM, Version 7.2, processing. ♦

Migrating Version 6.0 and 7.1 Files with Localized Indexes

C-ISAM, Version 6.0 or 7.1, files are compatible with C-ISAM, Version 7.2, applications.

To run a 7.2 application that uses a localized Index

1. Set the **INFORMIXDIR** environment variable to the directory where C-ISAM is installed.
2. If necessary, use the **islanginfo** function to determine the name of the locale that is associated with the localized index.
3. Set the **CLIENT_LOCALE** environment variable to the value that **LC_COLLATE** specified when the localized index was built.

Version 7.2 applications do not require a call to the **setlocale()** function to specify a custom collation order. C-ISAM uses the value of **CLIENT_LOCALE** to determine the locale that specifies the collation order. ♦

Transaction Logging and Recovery

You can use the transaction-log file to write a program that recovers C-ISAM files. Your program must open the log file and issue 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 backup media.

All programs that access recoverable C-ISAM files must have the same log file; otherwise, transaction recovery does 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. Use the same new log file to restart all programs.

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, C-ISAM cannot guarantee integrity.

Determining Version and Serial Number

You can determine the version and serial number of your C-ISAM product from two global variables defined in **isam.h**. The variable **isversnumber** contains the version number and **isserial** contains the serial number. When you open a file, or attempt to open a file, these variables are set.

You can compile and run the following lines to determine the version and serial numbers. You must link in **libisam.a**, as with any C-ISAM program.

```
#include <isam.h>
#include <stdio.h>
main()
{
    isopen ("fake", ISINPUT); /* attempts to open file*/
    printf("\nVersion number = %s,\nSerial number = %s\n",
          isversnumber, isserial);
}
```

Header Files

This appendix lists the contents of the **isam.h** and **decimal.h** header files.

The isam.h Header File

You must include the file **isam.h** in every C-ISAM program. Figure F-1 shows the contents of the **isam.h**.

Figure F-1
Contents of isam.h File

```
#ifndef ISAM_INCL /* avoid multiple include problems */
#define ISAM_INCL

#ifdef __STDC__
#include "../incl/decimal.h"
#endif

#define CHARTYPE 0
#define DECIMALTYPE0
#define CHARSIZE 1

#define INTTYPE 1
#define INTSIZE 2

#define LONGTYPE 2
#define LONGSIZE 4

#define DOUBLETTYPE 3
#ifdef NOFLOAT
#define DOUBLESIZE (sizeof(double))
#endif /* NOFLOAT */

#ifdef NOFLOAT
#define FLOATTYPE 4
#define FLOATSIZE (sizeof(float))
#endif /* NOFLOAT */
```

The isam.h Header File

```
#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 NCHARTYPE 7/* CHARACTER TYPE with localized collation */

#define MAXTYPE5
#define ISDESC 0x80 /* add to make descending type*/
#define TYPEMASK0x7F /* 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 lddb1
double lddb1();
#endif
double ldfltnull();
double lddb1null();
#endif

#define ISFIRST0 /* 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 ISEQUAL5 /* position to equal value*/
#define ISGREAT6 /* 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_len;            /* length of whole key*/
    long k_rootnode;        /* pointer to rootnode*/
};
#define k_start k_part[0].kp_start
#define k_leng k_part[0].kp_leng
#define k_type k_part[0].kp_type

#define ISNODUPS 000        /* no duplicates 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_recsize;       /* (maximum) data record size*/
    short di_idxsize;       /* index record size*/
    long di_nrecords;       /* number of records in file*/
};

```

The isam.h Header 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 ENOLCK    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 ENOLCKS   134    /* no more locks*/
#define ENOPARTN  135    /* reserved for future use */
#define ENOEXTN   136    /* reserved for future use */
#define EOVCUNK    137    /* reserved for future use */
#define EOVDBS    138    /* reserved for future use */
#define EOVLG     139    /* reserved for future use */
#define EGBLSECT  140    /* global section disallowing access - VMS */
#define EOVPARTN  141    /* reserved for future use */
#define EOVPAGE   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 EBADVCLEN 151    /* reserved for future use */
#define EBADRMSG  152    /* reserved for future use */
#define ENOMANU   153    /* must be in ISMANULOCK mode */
```

```

#define EDEADTIME 154      /* lock timeout expired */
#define EPMCHKBAD 155     /* primary and mirror chunk bad */
#define EBADSHMEM 156     /* can't attach to shared memory*/
#define EINTERUPT 157     /* interrupted isam call */
#define ENOSMI 158        /* operation disallowed on SMI pseudo table */
#define ECOL_SPEC 159     /* Invalid collation specifier */
#define ENLS_LANG ECOL_SPEC /* for compatibility with earlier versions */
#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_CHKLOG 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_PROCDIFS21584 /* can't open config file */
#define ES_IILLVAL 21586 /* illegal config file value */
#define ES_ICONFIG 21595 /* bad config parameter */
#define ES_ILLUSRS 21596 /* illegal number of users */
#define ES_ILLCKS 21597 /* illegal number of locks */
#define ES_ILLFILE 21598 /* illegal number of files */
#define ES_ILLBUFF 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

```

The isam.h Header File

```
* 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*/
extern char *islanginfo(); /* locale used for collation */
/* 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*/

#ifdef __STDC__
/*
** prototypes for file manipulation functions
*/
int isaddindex(int isfd, struct keydesc *keydesc);
int isaudit(int isfd, char *filename, int mode);
int isbegin();
int isbuild(char *filename, int reclen, struct keydesc *keydesc, int mode);
int iscleanup();
int isclose(int isfd);
int iscluster(int isfd, struct keydesc *keydesc);
int iscommit();
```

```

int isdelcurr(int isfd);
int isdelete(int isfd, char *record);
int isdelindex(int isfd, struct keydesc *keydesc);
int isdelrec(int isfd, long recnum);
int iserase(char *filename);
int isflush(int isfd);
int isglsversion(char *filename)
int isindexinfo(int isfd, struct keydesc *buffer, int number);
void islangchk(); /*used by Informix-SE only */
char *islanginfo(char *filename);
int islock(int isfd);
int islogclose();
int islogopen(char *logname);
int isnlsversion(char *filename);
void isnolangchk(); /* used by Informix-SE only */
int isopen(char *filename, int mode);
int isread(int isfd, char *record, int mode);
int isrecover();
int isrelease(int isfd);
int isrename(char *oldname, char *newname);
int isrewcurr(int isfd, char *record);
int isrewrec(int isfd, long recnum, char *record);
int isrewrite(int isfd, char *record);
int isrollback();
int issetunique(int isfd, long uniqueid);
int isstart(int isfd, struct keydesc *keydesc,
            int length, char *record, int mode);
int isuniqueid(int isfd, long *uniqueid);
int isunlock(int isfd);
int iswrcurr(int isfd, char *record);
int iswrite(int isfd, char *record);
/*
** prototypes for format-conversion and manipulation fuctions
*/
void ldchar(char *source, int length, char *destination);
double lddb1(char *location);
double lddb1null(char *location, short *nullflag);
int lddecimal(char *location, int length, dec_t *destination);
double ldfloat(char *location);
double ldfltnull(char *location, short *nullflag);
/* short ldint(char *location); */
long ldlong(char *location);
void stchar(char *source, char *destination, int length);
void stdbl(double source, char *destination);
void stdblnull(double source, char *destination, short nullflag);
void stdecimal(dec_t *source, char *destination, int length);
void stfloat(float source, char *destination);
void stfltnull(float source, char *destination, short nullflag);
/* void stint(short source, char *destination); */
void stlong(long source, char *destination);
/*
** DECIMALTYPE Functions
*/
int deccvasc(char *source, int length, dec_t *destination);
int dectoasc(dec_t *source, char *destination, int length, int right);
int deccvint(int source, dec_t *destination);
int dectoint(dec_t *source, int *destination);
int deccvlong(long source, dec_t *destination);
int dectolong(dec_t *source, long *destination);
int deccvflt(float source, dec_t *destination);
int dectoflt(dec_t *source, float *destination);

```

```
int deccvdbl(double source, dec_t *destination);
int dectodbl(dec_t *source, double *destination);
int decadd(dec_t *n1, dec_t *n2, dec_t *result);
int decsub(dec_t *n1, dec_t *n2, dec_t *result);
int decmul(dec_t *n1, dec_t *n2, dec_t *result);
int decdiv(dec_t *n1, dec_t *n2, dec_t *result);
int deccmp(dec_t *n1, dec_t *n2);
void deccopy(dec_t *source, dec_t *destination);
char *dececvl(dec_t *source, int ndigit, int *decpt, int *sign);
char *decfcvt(dec_t *source, int ndigit, int *decpt, int *sign);
#endif /* __STDC__ */

#endif /* ISAM_INCL */
```

The decimal.h Header File

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 F-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 F-2
*Structure of a
decimal or dec_t
Data Type*

The **dec_t** structure has the following four parts:

- dec_exp** holds the exponent of the normalized DECIMALTYPE number. This exponent represents a power of 100.
- dec_pos** holds the sign of the DECIMALTYPE 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 DECIMALTYPE number.
- dec_dgts** is a character array that holds the significant digits of the normalized DECIMALTYPE 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 C-ISAM functions 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

The top 1 bit is the sign of the number.

on	=	the number is positive
off	=	the number is negative

The low 7 bits are the exponent in excess of 64.

Remaining Bytes

The remaining bytes are the 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. The length can vary from 2 to 17 bytes. This format permits sorts of DECIMALTYPE numbers using a simple unsigned byte-by-byte comparison. Zero is represented as 80,00,00,... (in hexadecimal).

Figure F-3 shows the contents of the header file **decimal.h** that you must include in every program that uses the DECIMALTYPE data type.

Figure F-3
Contents of decimal.h File

```
#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 minimum 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 DECLENGTH(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 */

```


Notices

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J74/G4
555 Bailey Ave
P.O. Box 49023
San Jose, CA 95161-9023
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. (enter the year or years). All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

AIX; DB2; DB2 Universal Database; Distributed Relational Database Architecture; NUMA-Q; OS/2, OS/390, and OS/400; IBM Informix®; C-ISAM®; Foundation.2000™; IBM Informix® 4GL; IBM Informix® DataBlade® Module; Client SDK™; Cloudscape™; Cloudsync™; IBM Informix® Connect; IBM Informix® Driver for JDBC; Dynamic Connect™; IBM Informix® Dynamic Scalable Architecture™ (DSA); IBM Informix® Dynamic Server™; IBM Informix® Enterprise Gateway Manager (Enterprise Gateway Manager); IBM Informix® Extended Parallel Server™; i.Financial Services™; J/Foundation™; MaxConnect™; Object Translator™; Red Brick Decision Server™; IBM Informix® SE; IBM Informix® SQL; InformiXML™; RedBack®; SystemBuilder™; U2™; UniData®; UniVerse®; wintegrate® are trademarks or registered trademarks of International Business Machines Corporation.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Windows, Windows NT, and Excel are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names used in this publication may be trademarks or service marks of others.

Index

A

- Access method
 - implementation 2-14
 - indexed sequential 1-9
- Access modes
 - example of 1-17
 - list of 1-32
 - used in `isbuild()` function 8-16
- Adding a localized index B-9
- Adding a record
 - example 7-8
 - explanation 1-22
 - See also* Writing a record.
- Adding an index
 - example 2-9, 7-6
 - explanation 2-9
 - using `isaddindex()` function 2-9, 8-10
- Additional facilities
 - audit trail 6-6
 - file maintenance 6-3
 - force output 6-4
 - summary 6-14
- Audit trail
 - command modes 6-8
 - creating a new trail 8-12
 - example using `isaudit()` function 6-6
 - explanation 6-6
 - file format 6-8, D-8
 - using 6-6
 - using `isaudit()` function 8-12
 - with variable-length records 6-8

B

- `bcheck` utility
 - checking localized indexes B-12
 - description and use of A-1
 - using 6-11
 - using with variable-length record files 1-6
- Beginning a transaction, with
 - `isbegin()` function
 - implementing 5-4
 - syntax and description 8-14
- Block
 - definition of 2-23
 - See also* Index.
- Buffers
 - ISAMBUFS parameter E-2
 - size E-2
- Building a file
 - example 1-18, 7-5
 - explanation 1-14
 - locking mode 4-7
 - record size 1-15
 - using `isbuild()` function 8-16
- B+ tree
 - adding to 2-18
 - definition of 2-14
 - deleting from 2-22
 - growth of 2-18, 2-19
 - levels 2-16
 - maximum keys per node 2-16
 - nodes 2-14
 - organization 2-14
 - pointers 2-14

- root 2-14
- searching 2-17
- sequential addition to 2-19
- split 2-18

C

- C library functions
 - comparison to C-ISAM functions 1-8
 - using lseek() 1-8
 - using read() 1-8
 - using write() 1-8
- Call formats and descriptions 8-3
- Cancelling a transaction, using isrollback() function 5-4, 8-58
- Changing a filename
 - audit trail 8-12
 - using isrename() function 8-52
- Character set B-4
- CHARTYPE data type
 - collation order with B-8
 - description of 3-9
- Choosing an index. *See* Selecting an index.
- C-ISAM
 - compiling programs 1-35
 - running programs 1-36
- CLIENT_LOCALE environment variable
 - description of B-7
 - example B-7
 - syntax B-7
- Closing a file
 - audit trail 8-12
 - data file 1-32, 8-20
 - transaction log file 5-8, 8-40
 - using iscleanup() function 8-19
 - using isclose() function 1-32, 8-20
 - using islogclose() function 5-8, 8-40
- Closing files
 - using iscleanup() function 6-4
- Cluster index 6-10, 8-21
- .cm file extension B-5, E-4
- .cmo file extension B-5
- Code set
 - description of B-4
 - in locale name B-3
 - name mapping B-5
- Code-set file B-7
 - description of B-4
 - listing B-16
 - location of B-5
 - object B-5
 - removing E-4
 - source B-5
- Code-set order B-7
- Collation sequence
 - default 1-13
 - defined 1-13
 - multiple 1-14
 - multiple indexes B-9
 - with bcheck utility B-12
- Collation, definition of B-7
- Committing a transaction, using iscommit() function 5-4, 8-23
- Compilation
 - header files 1-35
 - using lint utility 1-35
 - See also* System administration.
- Compliance, industry standards Intro-15
- Compression of keys. *See* Key.
- Concurrency control
 - degree of concurrency 4-12
 - in transactions 5-9, 5-11
 - locking 4-3
- Conventions
 - command-line Intro-9
 - icon Intro-8
 - sample code Intro-12
 - typographical Intro-7
- Conversion functions. *See* Format-conversion functions.
- Creating a file. *See* Building a file.
- Current index. *See* Index.
- Current record
 - definition of 1-22
 - set by isread() function 1-29, 8-47
 - set by iswrcurr() function 1-22, 8-66
 - set by iswrite() function 8-68
- cvY.txt file B-16

D

- Data file
 - characteristics 1-37, 8-35, 8-38, 8-42
 - cluster 6-10, 8-21
 - organization 1-37
 - space utilization 1-37
- Data integrity
 - restoring with bcheck utility A-1
 - See also* Transaction.
- Data record
 - adding 1-22
 - address of 1-5
 - customer record 3-7
 - declaration of 1-5
 - deleting 1-24
 - employee record 1-3, 1-15, 2-3, 3-3, 7-3
 - identifying a 1-20
 - in a C-ISAM file 1-5, 3-7
 - performance record 7-3
 - record layout 1-5, 3-7
 - reservation of space for 1-5, 3-7
 - summary of identification methods 1-22
 - transferring to and from program 1-7
 - updating 1-25
- Data representation
 - character data 3-9
 - comparison of C-ISAM to C language 3-6
 - DECIMALTYPE data 3-13
 - double-precision data 3-12
 - floating-point data 3-12
 - format-conversion 3-9
 - integer data 3-11
 - long integer data 3-11
 - machine independence 3-5
 - overview 1-6
- Data type
 - collation order B-8
 - conversion functions 3-9
 - DECIMALTYPE 3-13
 - defining for keys 3-3
 - in variable-length record 3-8
 - introduction to data types 1-6
 - parameters 3-4

- summary 3-19
- See also* DECIMALTYPE data type.
- Deadlock, definition 4-12
- decadd() function, syntax and use of 8-90
- deccmp() function, syntax and use of 8-92
- deccopy() function, syntax and use of 8-93
- deccvint() function, syntax and use of 8-98
- deccvlong() function, syntax and use of 8-99
- decdiv() function, syntax and use of 8-90
- dececv() function, syntax and use of 8-100
- decfcvt() function, syntax and use of 8-100
- DECIMALTYPE data type
 - accuracy 3-15
 - dec_t structure 3-13
 - defining Decimal data 3-13
 - sizing DECIMALTYPE numbers 3-14
 - See also* DECIMALTYPE functions.
- DECIMALTYPE functions
 - decadd() 8-90
 - deccmp() 8-92
 - deccopy() 8-93
 - deccvint() 8-98
 - deccvlong() 8-99
 - decdiv() 8-90
 - dececv() 8-100
 - decfcvt() 8-100
 - decmul() 8-90
 - decsb() 8-90
 - dectoasc() 8-102
 - dectodbl() 8-104
 - dectoflt() 8-105
 - dectoint() 8-106
 - dectolong() 8-107
 - decvasc() 8-94
 - decvdbl() 8-96
 - decvflt() 8-97
 - lddecimal() 3-15, 8-74
 - overview 3-17
 - stdecimal() 3-15, 8-83
 - decmul() function, syntax and use of 8-90
 - decsb() function, syntax and use of 8-90
 - dectoasc() function, syntax and use of 8-102
 - dectodbl() function, syntax and use of 8-104
 - dectoflt() function, syntax and use of 8-105
 - dectoint() function, syntax and use of 8-106
 - dectolong() function, syntax and use of 8-107
 - decvasc() function, syntax and use of 8-94
 - decvdbl() function, syntax and use of 8-96
 - decvflt() function, syntax and use of 8-97
 - dec_t structure. *See* DECIMALTYPE data type.
 - Default locale B-3
 - Deleting a file. *See* Erasing a file.
 - Deleting a record
 - current record 1-24, 8-25
 - example 1-24, 7-11
 - using isdelcurr() function 1-24, 8-25
 - using isdelete() function 1-24, 8-26
 - using isdelrec() function 1-25, 8-30
 - using primary key 1-24, 8-26
 - using record number 1-25, 8-30
 - Deleting an index
 - explanation 2-10
 - using isdelindex() function 2-10, 8-28
 - dictinfo structure
 - definition of 2-12
 - using to get file information 2-12
 - See also* Index.
 - Dictionary block. *See* Index.
 - Dictionary format D-1
 - Dirty-read, using isread() function 8-47
 - Disk space, saving E-3
 - di_nkeys variable 8-36
 - di_recsz variable 8-36

- Documentation
 - on-line files Intro-14
 - printed Intro-13
 - related Intro-15
- Documentation notes Intro-14
- Duplicate key
 - compression 2-29
 - purpose 1-13
 - See also* Key.

E

- ELOCKED, with isread() function calls 8-47
- Ending a transaction
 - using iscommit() function 5-4, 8-23
 - using isrollback() function 5-4, 8-58
- Environment variable
 - CLIENT_LOCALE B-7
 - INFORMIXDIR 1-36
- Erasing a file
 - audit trail 8-31
 - data file 6-3
 - using iserase() function 6-3, 8-31
 - .lok lock file 8-31
- Error handling
 - C-ISAM error codes C-1
 - end of file 1-28
 - example 4-13
 - in example programs 7-4
 - locked records 4-13
 - overview 1-19
 - record not found 1-27
 - return codes 1-19
 - using iserrno global variable 1-19
 - values for iserrno global variable C-1
- Example programs
 - adding indexes 7-6
 - adding records 7-8
 - building a file 7-5
 - chaining 7-19
 - random update 7-11
 - record definitions in 7-3
 - sequential processing 7-16
 - using transactions 7-25

F

fcntl() locking
 using ISWAIT and ISLCKW 8-48
 X/Open compatibility 4-11

Field

 conversion between program and
 data record 1-6
 declaration using pointer 1-5, 3-8
 definition of 1-3
 key 1-12
 offset 1-4, 3-8

File

 charmap B-5
 code-set B-4, B-16
 cvY.txt B-16
 definition of 1-3
 lcX.txt B-14
 locale B-4
 maintenance 6-11
 maximum number of open files
 1-34
 object locale B-4
 registry E-3
 removing code-set files E-4
 removing unused locale E-3
 source locale B-4
 See also Operating system files.

File descriptor

 returned by `isbuild()` function 1-15
 returned by `isopen()` function 1-32
 using 1-15

File extension

 .cm B-5, E-4
 .cmo B-5
 .lc B-4
 .lco B-4

File-level locking

 explanation 4-7
 See also Locking.
 See also Locking modes.

Finding a record

 explanation 1-27
 key value 1-11
 See also Reading a record.
 See also Selecting an index.

Flush buffer, using `isflush()`

 function
 forcing with 6-4

 syntax and description 8-32

Format

 audit trail D-8
 dictionary D-1

Format-conversion functions

 character data 3-9
 double-precision data 3-11
 explanation 3-9
 floating-point data 3-11
 integer data 3-10
 introduction to 1-6
 ldchar() 3-9, 8-71
 lddblnull() 3-11, 8-73
 lddbl() 3-11, 8-72
 lddecimal() 3-15, 8-74
 ldfloat() 3-11, 8-76
 ldfltnull() 3-11, 8-77
 ldint() 3-10, 8-78
 ldlong() 3-10, 8-79
 long integer data 3-10
 stchar() 3-9, 8-80
 stdblnull() 3-11, 8-82
 stdbl() 3-11, 8-81
 stdecimal() 3-15, 8-83
 stfloat() 3-11, 8-85
 stfltnull() 3-11, 8-86
 stint() 3-10, 8-87
 stlong() 3-10, 8-88
 See also DECIMALTYPE
 functions.

Free-list block. *See* Index.

Function list

 format-conversion functions 8-6
 functions to determine NLS
 information 8-5
 functions to implement locking
 8-4
 functions to implement
 transactions 8-5
 functions to manipulate
 DECIMALTYPE data 3-18, 8-7
 functions to manipulate files 8-3
 functions to manipulate indexes
 8-3
 functions to manipulate records
 8-4
 functions, additional 8-5

Function return codes. *See* Error
 handling.

G

glfiles utility

 -cm option B-13
 description of B-12
 -lc option B-13
 sample output B-14, B-15, B-16

Global Language Support (GLS)

 applications 1-21
 C-ISAM locale file access 1-36
 description of Intro-6
 determining collation sequence of
 index B-11

example

 adding a localized index B-10
 checking index B-10
 identifying locale B-11

locale B-2

locale name B-2

migrating applications E-4

performance issues 2-31

 performance of indexed access
 2-31

removing

 code-set files E-4
 locale files E-3

specifying a locale B-6

supported locales B-6

 with `bcheck` utility B-12

GLS. *See* Global Language Support.

H

How to use C-ISAM 1-3

I

Identifying records

 by key value 1-20
 by record number 1-21
 current record 1-21
 summary of methods 1-22

Index

 adding a localized index B-9
 adding an index 2-9
 characteristics 2-12, 8-35, 8-38,
 8-42
 cluster 6-10, 8-21

- creating a localized index B-8
- current 1-29
- DECIMALTYPE data in 3-13
- defining a key for 1-16, 2-3
- definition in C-ISAM 2-5
- determining collation sequence B-11
- dictionary node 2-23
- explanation 2-10
- file organization 2-23
- file organization and variable-length records 1-6
- free-list node 2-24
- identifying an index 2-5, 2-13
- implementation 2-14
- key-description nodes 2-23
- localized, definition of B-8
- maximum number of parts 8-10
- organization 2-14
- performance 2-24
- performance issues 2-31
- physical order 2-11
- primary 1-13, 2-11
- record number order 2-11
- using isaddindex() function 2-9, 8-10
- using isdelindex() function 2-10, 8-28
- Indexed access, overview 1-9
- Indexed sequential access method
 - flexibility 1-10
 - overview 1-9
- INFORMIXDIR environment
 - variable 1-36
 - location of code-set files B-5, B-16
 - location of locale files B-4
 - location of registry file B-5
 - syntax 1-36
 - with glfiles B-13
- isaddindex() function
 - example 7-6
 - explanation 2-9
 - syntax and use of 8-10
- ISAMBUFS E-2
- isaudit() function
 - command modes 6-8
 - explanation 6-6
 - syntax and use of 8-12
- ISAUTOLOCK lock mode 8-16, 8-44
- isbegin() function
 - explanation 5-4
 - syntax and use of 8-14
 - See also* Transaction.
- isbuild() function
 - example 1-18, 7-5
 - syntax and use of 8-16
 - used after islogopen() function 8-41
- iscleanup() function
 - closing files implicitly 1-34
 - syntax and use of 8-19
 - using at end of program 6-4
- isclose() function
 - caution 8-20
 - syntax and use of 8-20
- iscluster() function
 - explanation 6-10
 - regenerating indexes 6-11
 - syntax and use of 8-21
- iscommit() function
 - explanation 5-4
 - syntax and use of 8-23
 - See also* Transaction.
- ISCURR locator mode 1-29, 1-30
- isdelcurr() function
 - example 7-11
 - explanation 1-24
 - syntax and use of 8-25
- isdelete() function
 - example 7-11
 - explanation 1-24
 - syntax and use of 8-26
- isdelindex() function
 - explanation 2-10
 - syntax and use of 8-28
- isdelrec() function
 - explanation 1-25
 - syntax and use of 8-30
- ISEQUAL locator mode 1-29, 1-30
- iserase() function
 - explanation 6-3
 - syntax and use of 8-31
- iserrno global variable
 - description of 1-19
 - end of file 1-28
 - locked records 4-13
- record not found 1-27
- using 1-19
- values of C-1
- ISEXCLLOCK lock mode 1-18, 8-16, 8-44
- ISFIRST locator mode 1-29, 1-30
- ISFIXLEN mode 8-17, 8-44
- isflush() function
 - explanation 6-4
 - syntax and use of 8-32
- isglsversion() function
 - checking for a localized index B-11
 - syntax and use of 8-33
- ISGREAT locator mode 1-29, 1-30
- ISGTEQ locator mode 1-29, 1-30
- isindexinfo() function
 - example 1-37, 2-13
 - general use 2-12
 - syntax and use of 8-35
- ISINOUT access mode 8-16, 8-32, 8-44
- ISINPUT access mode 8-16, 8-44
- ISKEEPLOCK 4-10
- islanginfo() function
 - determining collation sequence B-11
 - syntax and use of 8-38
- ISLAST locator mode 1-29, 1-30
- ISLCKW 8-46
- islock() function
 - explanation 4-9
 - syntax and use of 8-39
 - See also* isunlock() function.
- islogclose() function
 - explanation 5-8
 - syntax and use of 8-40
 - See also* Transaction.
- islogopen() function
 - caution 8-41
 - explanation 5-7
 - required for transaction 8-17, 8-45
 - syntax and use of 8-41
 - See also* Transaction.
- ISMANULOCK lock mode 8-16, 8-39, 8-44
- ISNEXT locator mode 1-29
- isnlsversion() function
 - syntax and use of 8-42

ISNOLOG mode 8-17

isopen() function
 explanation 1-32
 ISTRANS option 5-6
 syntax and use of 8-44
 used after islogopen() function 8-41
See also Access modes.
See also Locking.
See also Transaction.

ISOUTPUT access mode 8-16, 8-32, 8-44

ISPREV locator mode 1-29

isread() function
 example 1-27, 7-11, 7-16
 explanation 1-27
 syntax and use of 8-46
See also Locking.

isreclen global variable
 opening a file 1-33
 set by isindexinfo() function 2-12, 8-36
 with variable-length files 1-19

isrecnum global variable
 explanation 1-21
 finding records by record number 1-31
 set by isdelcurr() function 8-25
 set by isdelete() function 8-26
 set by isdelrec() function 8-30
See also Record number.

isrecover() function
 explanation 5-8
 syntax and use of 8-50
See also System administration.
See also Transaction.

isrelease() function
 explanation 4-11
 syntax and use of 8-51

isrename() function
 explanation 6-3
 syntax and use of 8-52

isrewcurr() function
 explanation 1-26
 syntax and use of 8-53

isrewrec() function
 explanation 1-26
 syntax and use of 8-55

isrewrite() function
 example 7-11
 explanation 1-25
 syntax and use 8-56

isrollback() function
 explanation 5-4
 syntax and use of 8-58
 with ISTRANS mode 8-17
See also Transaction.

isserial global variable E-6

issetunique() function
 explanation 6-5
 syntax and use of 8-60
See also isuniqueid() function.

ISSKIPLOCK 1-27

isstart() function
 example 7-11, 7-16, 7-19
 explanation 1-29
 syntax and use of 8-61

ISTRANS mode 8-17, 8-44, 8-45

isuniqueid() function
 explanation 6-5
 syntax and use of 8-64
See also issetunique() function.

isunlock() function
 explanation 4-9
 syntax and use of 8-65
See also islock() function.

ISVARLEN mode 1-33, 8-16, 8-44

isversnumber global variable E-6

ISWAIT 8-46

iswrcurr() function
 example 1-22
 syntax and use of 8-66

iswrite() function
 example 1-22, 7-8
 syntax and use 8-68

K

Kernel locking 4-11, 4-12

Key
 choice of 1-11
 compression 2-7, 2-26
 data type definition 3-3
 defining a key 1-16
 definition in C-ISAM 1-12, 2-3
 definition of 1-9

descending order 2-8

duplicate 1-13, 2-7

flags 2-6, 2-7

key-description structure 1-12, 1-16, 2-5

maximum size 2-8, 8-10

number of parts 2-6, 2-8

overview of usage 1-11

packing density 2-25

primary 1-13, 1-26, 2-11

unique 1-12, 2-6, 2-7

value 1-20

keydesc structure
 defining a key 2-5
 defining a primary key 1-16
 definition of 2-7
See also Key.

Key-description block. *See* Index.

Key-description structure
 defining a key 2-5
 definition of 2-6
 overview 1-16
See also Key.

keypart structure
 defining a key 2-5
 definition of 2-8
See also Key.

Keys in C-ISAM Files 1-10

Keyword
 definition of 1-9
See also Key.

k_flags variable 2-7

k_nparts variable 1-31, 8-17

L

Language, in locale name B-3, B-4

.lc file extension B-4

.lco file extension B-4

lcX.txt file B-14

ldchar() function
 explanation 3-9
 syntax and use of 8-71

lddbnull() function
 explanation 3-11
 syntax and use of 8-73

- lddbl() function
 - explanation 3-11
 - syntax and use of 8-72
- lddecimal() function
 - explanation 3-15
 - syntax and use of 8-74
- ldfloat() function
 - explanation 3-11
 - syntax and use of 8-76
- ldftnull() function
 - explanation 3-11
 - syntax and use of 8-77
- ldint() function
 - explanation 3-10
 - syntax and use of 8-78
- ldlong() function
 - explanation 3-10
 - syntax and use of 8-79
- Leading-character compression
 - explanation 2-27
 - See also* Key.
- Level. *See* B+ tree.
- libisam.a library E-2
- Locale
 - default B-3
 - definition of B-2
 - identifying B-2
 - specifying B-6
- Locale file
 - description of B-4
 - location of B-4
 - object B-4, E-3
 - removing unused E-3
 - required E-3
 - source B-4, E-3
- Locale name
 - code set in B-3
 - description of B-2
 - language in B-3
 - language name B-4
 - modifier in B-3
 - territory in B-3
 - territory name B-4
- Localized index
 - adding, example of B-9
 - creating B-8
 - definition of B-8
- Localized order, definition of B-7

- Locking
 - automatic locking with isread
 - 4-10
 - by transactions 5-10
 - concurrency control 4-3
 - degree of concurrency 4-12
 - duration of locking 8-47
 - during add or delete of an index
 - 4-8
 - file-level 4-7, 8-39
 - ISLOCK option in isread 4-11,
 - 8-46
 - islock() function 4-9
 - ISSKIPLOCK option in isread 8-46
 - manual 4-9, 4-10
 - overview 4-3
 - record-level 4-10, 8-46, 8-47
 - single user systems 4-7
 - specifying no locking 4-7
 - summary 4-14
 - types of 4-7
 - unlocking file 8-65
 - unlocking records 8-51
 - using 4-7
 - using islock() function 8-39
 - using isrelease() function 4-11,
 - 8-51
 - using isunlock() function 4-9, 8-65
 - See also* Locking modes.
- Locking modes
 - automatic record locking 4-10
 - exclusive file locking 1-33, 2-10,
 - 4-8
 - ISAUTOLOCK 4-10
 - ISEXCLLOCK 1-33, 2-10, 4-8
 - ISMANULOCK 1-32, 4-7, 4-9, 4-11
 - ISWAIT 4-11
 - manual file locking 4-9
 - manual record locking 4-11
 - no locking 1-32, 4-7
 - summary 4-14
 - waiting for locks 4-11
 - See also* Locking.

M

- Machine notes Intro-14
- Manipulating records in C-ISAM
 - files 1-20
- MAXKEYSIZE, explanation of 8-10
- Message, bcheck error B-12
- Migrating applications E-4
- Modifier, in locale name B-3

N

- Native Language Support (NLS)
 - checking for localized indexes
 - B-12
 - checking index with bcheck utility
 - B-12
 - collation order B-8
 - indexes, compatible locales B-15
- NCHARTYPE data type
 - collation order B-8
 - in an example program B-9
- NLS. *See* Native Language Support.
- Node
 - definition of 2-14
 - See also* B+ tree.
- Non-default locale, specifying B-6
- NPARTS 2-6, 8-10

O

- Offset, defining a field 1-4, 3-8
- Opening a file
 - access modes 1-32
 - audit trail 6-8, 8-12
 - data file 1-32, 8-44
 - file descriptor 1-32
 - ISTRANS option 5-6
 - locking mode 1-32, 4-7
 - of variable-length records 1-33
 - specifying isbuild() function
 - arguments 1-17
 - transaction log file 5-7, 8-41
 - using islogopen() function 5-7,
 - 8-41
 - using isopen() function 1-32, 8-44

Operating system files
 .dat 1-14
 .idx 1-14
Organization of C-ISAM files
 data file 1-37
 index file 2-23
 overview 1-14

P

Performance
 key compression 2-26
 key size 2-24, 2-25
 multiple indexes 2-30
 tree height 2-24
Physical order index 2-11
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.
Program files E-2
Programs
 compiling 1-35
 running 1-36
 See also Example programs.

R

Reading a record
 automatic locking 4-10
 by record number 1-31, 8-47
 example 1-27, 7-19
 explanation 1-27
 locking option 4-11, 8-46, 8-47
 retrieval modes. *See* Search modes.
 search modes 1-27, 8-46
 summary of search modes 1-29
 using `isread()` function 1-27, 8-46
 See also `isread()` function.
Record
 definition of 1-3
 length 1-5
 variable-length 1-5
 See also Data record.

Record number
 definition of 1-22
 example of retrieval by 1-31
 retrieval by 1-31, 2-11
 setting with `isrecnum` global variable 1-21
 See also Data Record.
Record-level locking
 explanation 4-10
 See also Locking modes.
 See also Locking.
Recover transactions, using
 `isrecover()` function 8-50
Recovery
 caution 5-8
 explanation 5-8
 rollforward 5-8
 transaction 8-50
 See also System administration.
 See also Transaction.
Release notes Intro-14
Release record locks. *See* `isrelease()` function.
Removing a file. *See* Erasing a file.
Removing an index. *See* `isdelindex()` function.
Renaming a file, using `isrename()` function 6-3, 8-52
Reorganization
 data file 1-37
 index 2-22
Representation of data. *See* Data representation.
Retrieval by record number
 using `isread()` function 1-31, 8-47
 using `isstart()` function 1-31, 8-62
Return codes. *See* Error handling.
Rewriting a record
 by record number 1-26, 8-55
 current record 1-26, 8-53
 example 1-25, 7-11
 identified by primary key 1-25, 8-56
 using `isrewcurr()` function 1-26, 8-53
 using `isrewrec()` function 1-26, 8-55

 using `isrewrite()` function 1-25, 8-56
 See also Updating a record.
Rollforward recovery. *See* Recovery.
Rolling back a transaction, using
 `isrollback()` function 5-4, 8-58

S

Selecting an index
 example 7-16
 explanation 1-29
 retrieval by record number 1-31
 starting position within 1-30, 8-61, 8-62
 using `isstart()` function 1-29, 8-61
Sequential access
 example 1-28, 7-16
 overview 1-9
 See also `isread()` function.
 See also `isstart()` function.
Serial number, determining E-6
Skipping locked records 1-27
Sorting
 by code-set order B-7
 by localized order B-7
`stchar()` function
 explanation 3-9
 syntax and use of 8-80
`stdbnull()` function
 explanation 3-11
 syntax and use of 8-82
`stdbl()` function
 explanation 3-11
 syntax and use of 8-81
`stdecimal()` function
 explanation 3-15
 syntax and use of 8-83
`stfloat()` function
 explanation 3-11
 syntax and use of 8-85
`stfltnull()` function
 explanation 3-11
 syntax and use of 8-86
`stint()` function
 explanation 3-10
 syntax and use of 8-87

- stlong() function
 - explanation 3-10
 - syntax and use of 8-88
- System administration
 - files E-1
 - installation E-1
 - ISAMBUFS parameter E-2
 - transaction logging and recovery E-5

T

- Territory B-4
- Territory, in locale name B-3
- Trailing spaces 1-8
- Trailing-space compression
 - explanation 2-28
 - See also* Key.
- Transaction
 - beginning 5-4, 8-14
 - cancelling 5-4, 5-6, 8-58
 - caution during recovery 5-8
 - closing log file 5-8, 8-40
 - committing 5-4, 8-23
 - concurrency control 5-11
 - concurrent execution of 5-9
 - creating log file 5-7
 - data integrity 5-9
 - definition of 5-3
 - ending 8-23
 - example 5-5, 7-25
 - implementing 5-4
 - ISTRANS option in isopen 5-6
 - locking 5-10
 - logging 5-7, E-5
 - management services 5-4
 - opening log file 5-7, 8-41
 - purpose of 5-3
 - recoverable, rollback disallowed 5-6
 - recovery 5-7, 8-50, E-5
 - rollforward recovery 5-8
 - summary 5-12
 - unit of work 5-4
 - using isaddindex() function in 8-10
 - using isaudit() function in 8-13
 - using isbegin() function 5-4, 8-14

- using isbuild() function 8-18
- using iscommit() function 5-4, 8-23
- using islogclose() function 5-8, 8-40
- using islogopen() function 5-7, 8-41
- using isrecover() function 5-8, 8-50, E-5
- using isrollback() function 5-4, 8-58
- with variable-length records 5-7
- See also* Recovery.

- Transferring data, between
 - program and C-ISAM record 1-7

U

- Unique identifier
 - changing with issetunique() function 6-5
 - explanation 6-5
 - returning with isuniqueid() function 6-5, 8-64
 - setting with issetunique() function 8-60
- Unique key
 - purpose 1-12
 - See also* Key.
- Unlocking file. *See* Locking.
- Unlocking records. *See* Locking.
- Updating a record
 - example 7-11
 - explanation 1-25
 - See also* Rewriting a record.
- Utility
 - bcheck A-1
 - glfiles B-12

V

- Variable-length records
 - building a file 1-19
 - data corruption 6-11
 - in transactions 5-7
 - programming with 1-5
 - with audit trails 6-8
- Version number, determining E-6

W

- Waiting for locks 8-46, 8-48
- What is a C-ISAM file? 1-3
- Why use transaction management? 5-3
- Writing a record
 - current record 1-22
 - example 7-8
 - using iswrcurr() function 1-22, 8-66
 - using iswrite() function 1-22, 8-68

X

- X/Open compatibility 4-12

