NetWare®

Btrieve Record Manager

NOVELL®

NOVELL

Btrieve®

Record Manager

NetWare®

NOVELL

Btrieve Record Manager

NetWare®

DISCLAIMER

FCC WARNING

Computing devices and peripherals manufactured by Novell generate, use, and can radiate radio frequency energy, and if not installed and used in accordance with the instructions in this manual, may cause interference to radio communications. Such equipment has been tested and found to comply with the limits for a Class A computing device pursuant to Subpart J of Part 15 of the FCC Rules, which are designed to provide reasonable protection against radio interference when operated in a commercial environment. Operation of this equipment in a residential area is likely to cause interference, in which case the user—at his own expense—will be required to take whatever measures may be required to correct the interference.

Some components may not have been manufactured by Novell, Inc. If not, Novell has been advised by the manufacturer of the component that the component has been tested and complies with the Class A computing device limits as described above.

Novell Development Products Division
6034 West Courtyard
Suite 220
Austin Texas 78730 USA

October 1988 Edition
Manual Revision 1.00
For Btrieve Software Version 5.0 and Above
Novell Part # 100-000516-001

BSH 89

IMPORTANT NOTICE

This software product contains

- The *Btrieve Record Manager* manual;
- The files BSERVER.VAP, BROUTER.VAP, BREQUEST.EXE, and BREQUEST.DLL;
- A library of language interface routines;
- The B and BUTIL standalone utilities.

As a part of your Btrieve application, you may distribute the language interface that is necessary for your product to access Btrieve.

You may *NOT* distribute any of the following:

- Any portion of the manual
- Unlinked language interfaces
- The B and BUTIL standalone utilities
- BSERVER.VAP, BROUTER.VAP, BREQUEST.EXE, or BREQUEST.DLL

HOW TO USE THIS MANUAL

This manual is intended to be used as a reference and operations guide for programmers, systems developers, and systems integrators using the NetWare Btrieve Record Manager. It explains the concepts and operations that programmers should understand in order to build application programs using Btrieve as a record management and retrieval system.

Immediately following this preface is a short section describing the new features added to Btrieve for this version.

Chapter 1 provides an overview of the features Btrieve provides and includes a discussion of how Btrieve works.

Chapter 2 discusses Btrieve file structures, file management, and how Btrieve ensures file integrity.

Chapter 3 explains how to use the BSETUP configuration program to configure and install Btrieve on a network.

Chapter 4 explains the utility programs that are included with Btrieve.

Chapter 5 discusses how to interface Btrieve with various programming languages.

Chapter 6 explains the purpose and use of the 36 Btrieve operations.

Appendix A lists the Btrieve operation codes.

Appendix B lists and discusses the status codes and error messages Btrieve returns to your application.

Appendixes C, D, E, and F use program examples written in Pascal, COBOL, C, and BASIC to illustrate how to use Btrieve for various record operations.

Appendix G contains discussions and diagrams of the Btrieve extended key types.

At the end of this manual is a glossary of Btrieve related terms that appear in this manual.

You should read chapters 1 through 3 before you attempt to install and use Btrieve to develop database systems. If you are only installing Btrieve on a particular server, you should read and understand the materials contained in Chapter 3. Chapters 4 through 6, and the appendixes, are useful to programmers during program development.

NEW FEATURES FOR THIS VERSION

Btrieve v5.0 contains several new features. The following paragraphs list the new features and provide a short description of each.

BROUTER VAP PROCESS

The BROUTER process included with this version provides communication capabilities between BSERVER and other VAPs. This capability allows VAPs, as well as workstation applications, to make Btrieve calls to BSERVER.

DATA COMPRESSION

Btrieve now incorporates an optional data compression algorithm in addition to the blank truncation feature found in earlier releases. The new data compression algorithm compresses certain repeating characters in a record when the application writes the record to the file, and expands the record to its original size when the application reads the record. When you create a file, you can specify that you want it to allow data compression.

STEP OPERATIONS

Three Step operations have been added to Btrieve. These operations, Step First, Step Previous, and Step Last, allow you to traverse a file in either direction using the physical order of the records. The name of the Step Direct operation has been changed to Step Next.

BUTIL CHANGES

A new command, –CLONE, has been added to the BUTIL utility. This command allows you replicate the structure of an existing file without destroying the data contained in the original file.

A new option, "Replace Existing File," has been added to the –CREATE command. This option allows you to specify whether you want to create a new file if a file of the same name already exists.

NEW BTRIEVE FILE TYPES

Btrieve now recognizes two new data file types, data-only and key-only, as well as the standard Btrieve files supported by earlier releases. You can define a file as being data-only or key-only when you create it.

Data-only Files. Data-only files do not contain any index pages. Instead, the records are stored on the data pages in chronological order of insertion, and can be accessed using the Step operations. You may add supplemental indexes to a data-only file at any time after the file is created.

Key-only Files. Key-only files do not contain any data pages. Instead, the entire record is stored with the index, improving data insertion and retrieval times. Key-only files are very useful for quick look up of records, and can be used as external indexes for other files.

FREE SPACE THRESHOLD FOR VARIABLE LENGTH RECORDS

When you define a file that contains variable length records, Btrieve allows you to reserve a certain percentage of the variable length pages for future expansion of the records. This can reduce fragmentation of the records across multiple pages, resulting in reduced access times for frequently updated variable length files.

AUTOINCREMENT KEY TYPE

The autoincrement key type allows a user to define a key for which Btrieve automatically maintains the value. You can use autoincrement keys as record numbers which Btrieve automatically increments as records are added to the file.

MANUAL KEYS

Manual keys allow you to define a key which Btrieve does not automatically insert into the index tree. Instead, your application controls which records are to be indexed by the manual key, and can insert records into the index or remove them from the index as necessary.

TABLE OF CONTENTS

C Pascal Examples

G Extended Key Types

LIST OF FIGURES

1 INTRODUCTION TO BTRIEVE

Btrieve is a ready-made record management system that provides you with the necessary functions for storing, retrieving, and updating the data in your database files. Because of Btrieve's advanced techniques and structures, you can ignore physical file structures, index maintenance, and concurrency problems, and concentrate on the logical aspects of your files and database.

To access Btrieve, you include specific function calls in your program code, passing to Btrieve the information it needs to perform the required operation. Because the calling conventions are different for the various high-level languages and compilers, Btrieve includes interface routines for many of the more popular languages and compilers, including the following:

- Microsoft QuickBASIC, IBM Interpreted and Compiled BASIC, Turbo Basic, and several other BASIC compilers
- IBM (or Microsoft) Pascal, Turbo Pascal, and several other Pascal compilers
- Microsoft C, Lattice C, Turbo C, and several other C compilers
- Microsoft COBOL, Realia COBOL, MicroFocus COBOL, and several other COBOL compilers

This manual contains documentation and program examples for BASIC, C, Pascal, and COBOL. Documentation and additional interface routines for other languages and compilers are included on the Btrieve diskette. Chapter 4 includes information about the requirements for writing an assembly language routine to call Btrieve from languages for which no interface is provided.

BTRIEVE FEATURES

The following sections introduce some of the features that make Btrieve a uniquely powerful record management system.

INDEX MAINTENANCE

Btrieve automatically creates and maintains the indexes in your files as you insert, update, and delete records. In addition to automatic index maintenance, Btrieve provides index support in the following ways:

- Support for as many as 24 indexes per file
- Support for adding or dropping supplemental indexes after a file has been created
- Support for 14 different data types for key values
- Support for duplicate, modifiable, segmented, null, manual, and descending key values

Chapter 2 contains more detailed information about how you can use Btrieve's indexing features in your application program.

FILE SPECIFICATIONS

Btrieve allows you to create data files by using function calls from your application program or by using an external utility program (BUTIL). At the file level, Btrieve offers you the following features:

- File sizes up to 4 billion bytes
- An unrestricted number of records
- Ability to extend a file across two storage devices

- Consistent file definition and management routines
- Consistent file structures

MEMORY MANAGEMENT

Btrieve allows you to specify the amount of memory used by the I/O cache buffer based on your application's memory requirements and the total amount of memory installed on your server. The amount of memory you reserve for the I/O buffer cache can have an effect on Btrieve's performance.

CONCURRENCY AND SECURITY CONTROLS

Btrieve provides capabilities for controlling concurrency and data security in a network environment. Btrieve maintains file integrity and security by allowing you to

- Specify single and multiple record level locks;
- Lock data files;
- Define logical transactions;
- Assign owner names to files;
- Specify dynamic encryption and decryption of data.

Chapter 2 contains more information about how Btrieve and your application handle concurrency and security.

DATA INTEGRITY

Btrieve uses several techniques to ensure the integrity of your data files. These techniques include

- Using pre-image files to store images of file pages before records are inserted, updated, or deleted;
- Using transaction processing to maintain consistency between data files during multiple file updates.

BTRIEVE UTILITIES

Btrieve includes two utility programs, as well as several console commands, that enable you to perform testing and data management without writing an application program. These include

- **BUTIL.EXE**, a command line utility that allows you to create and manage Btrieve data files.
- **B.EXE**, an interactive utility program that you can use for instructional purposes and for testing and debugging your application program logic.
- **Console Commands** that allow you to monitor and manage NetWare Btrieve activity on your network.

Refer to Chapter 4 for more information about the Btrieve utilities and console commands.

RECORD MANAGEMENT OPERATIONS

Btrieve provides 36 separate operations that you can perform from within your application program. To perform a Btrieve operation, your application must complete the following tasks:

- Satisfy any prerequisites the operation requires.

  For example, before your application can perform any file I/O, it must first make the file available for access by performing a Btrieve Open operation for that file.

- Initialize the parameters that the particular Btrieve operation requires.

  The parameters are program variables or data structures that correspond in type and size to the particular values that Btrieve expects for a given operation.

- Execute the Btrieve function call (BTRV).

  The exact format of the Btrieve function call varies from language to language.

- Evaluate the results of the function call.

  Btrieve always returns a status code indicating the success (status = 0) or failure (status <> 0) of an operation. Your application must always check for nonzero status codes and take appropriate action.

  In addition, Btrieve returns data or other information to the individual parameters based on the purpose of the operation.

Table 1.1 lists the Btrieve operations and their operation codes and provides a brief description of the function the operation performs.

OPERATION	CODE	DESCRIPTION
Open	0	Makes a file available for access
Close	1	Releases a file from availability
Insert	2	Inserts a new record into a file
Update	3	Updates the current record
Delete	4	Removes the current record from the file
Get Equal	5	Gets the record whose key value matches the requested key value
Get Next	6	Gets the record following the current record in the index path
Get Previous	7	Gets the record preceding the current record in the index path
Get Greater	8	Gets the record whose key value is greater than the requested key value
Get Greater or Equal	9	Gets the record whose key value is equal to or greater than the requested key value
Get Less Than	10	Gets the record whose key value is less than the requested key value
Get Less Than or Equal	11	Gets the record whose key value is equal to or less than the requested key value
Get First	12	Gets the first record in the requested access path
Get Last	13	Gets the last record in the requested access path
Create	14	Creates a Btrieve file with the specified characteristics
Stat	15	Returns file and index characteristics, and number of records

Table 1.1
Btrieve Operations

OPERATION	CODE	DESCRIPTION
Extend	16	Extends a file across two disk volumes
Set Directory	17	Changes the current directory
Get Directory	18	Returns the current directory
Begin Transaction	19	Marks the beginning of a set of logically related operations
End Transaction	20	Marks the end of a set of logically related operations
Abort Transaction	21	Removes operations performed during an incomplete transaction
Get Position	22	Gets the position of the current record
Get Direct	23	Gets the record at a specified position
Step Next	24	Gets the record from the physical location following the current record
Stop	25	Terminates the memory resident Record Manager programs at a workstation
Version	26	Returns the currently loaded version of the Record Manager
Unlock	27	Unlocks a record or records
Reset	28	Releases all resources held by a workstation
Set Owner	29	Assigns an owner name to a file
Clear Owner	30	Removes an owner name from a file
Create Supplemental Index	31	Creates a supplemental index
Drop Supplemental Index	32	Removes a supplemental index
Step First	33	Returns the record in the first physical location in the file
Step Last	34	Returns the record in the last physical location in the file
Step Previous	35	Returns the record from the physical location preceding the current record

Table 1.1 (*Continued*)
Btrieve Operations

Refer to Chapter 6 for complete descriptions of all of the Btrieve record management operations. Chapter 5 contains instructions for calling Btrieve from BASIC, Pascal, COBOL, and C. Appendixes C, D, E, and F contain program examples that illustrate how to initialize parameters, execute Btrieve function calls, and check the returned status code.

HOW BTRIEVE WORKS

NetWare Btrieve is a server-based implementation of the Btrieve Record Manager operating under Advanced NetWare v2.1 or above. All Btrieve requests from network stations are processed at a network server. In comparison with a client-based program, the server-based configuration improves network database operations for the following reasons:

- Processing at the server is centralized, allowing for efficient multiuser controls.
- The number of network requests is reduced, resulting in faster network performance because of improved server utilization.
- Network use is reduced because a smaller quantity of data is transferred across the network.

Calls to NetWare Btrieve have the same format as calls to Btrieve in other environments. Single-user Btrieve applications ported to NetWare Btrieve may require additional status checking because of the concurrency checking required in a multiuser environment.

In addition to providing a server-based record management system for workstation applications, NetWare Btrieve also includes a facility that allows VAPs to issue Btrieve calls to BSERVER. The BROUTER program provides the necessary communication between BSERVER and other VAP programs.

THE BSERVER PROGRAM

The BSERVER program should be loaded at every file server that stores Btrieve files. BSERVER consists of the Btrieve core program that handles the specific Btrieve requests, a server shell, and a network communications module. BSERVER performs the following functions:

- It performs all disk I/O for the Btrieve files stored at the server where it is resident.
- It issues and releases all record-level and file-level locks at the server where it is resident.
- It packages all Btrieve requests it processes for transmittal either to a copy of BREQUEST at a workstation, or to a copy of BROUTER at a server.

Application programs and VAPs that issue Btrieve calls always communicate with BSERVER via either BREQUEST or BROUTER.

THE BREQUEST PROGRAM

The BREQUEST program must be loaded at each station that makes Btrieve requests to the server. Application programs at workstations communicate with BSERVER via BREQUEST. BREQUEST performs the following functions:

- It receives Btrieve requests from your application program and relays them to BSERVER.
- It returns the results of the Btrieve requests to your application.

At OS/2 workstations, the BREQUEST.DLL and BTRCALLS.DLL dynamic link routines must be available to the application program. The BTRCALLS.DLL routine must be available in order to maintain compatibility between NetWare Btrieve and Btrieve for OS/2. The BREQUEST.DLL routine provides communication between your application program and BSERVER.

THE BROUTER PROGRAM

The BROUTER program loads at a network file server. It is an interprocess communications program that allows other VAP programs loaded on network file servers to communicate with BSERVER. This capability allows you to write a Btrieve application program as a VAP, making your application server-based rather than client-based.

The BROUTER program performs the following functions:

- It tracks the Btrieve activity of the VAP that executed the Btrieve interrupt.
- It tracks the Btrieve activity for each workstation that initiates Btrieve requests to a VAP.
- It monitors access to Btrieve files stored at a single server or at multiple servers on the network.
- It serializes Btrieve requests to the BSERVER programs loaded on one or more servers.

Refer to "Interfacing with BROUTER," beginning on page 5-33, for instructions about writing an interface to BROUTER.

FLOW OF CONTROL

The NetWare Btrieve programs function as if they were a subroutine of your application program. NetWare Btrieve supports the following two methods for accessing BSERVER:

- A workstation application can access BSERVER via the BREQUEST program.
- A workstation application can call a VAP, which then communicates with BSERVER via BROUTER.

The following sections describe the two methods of access.

ACCESS TO BSERVER VIA BREQUEST

The following steps illustrate the flow of control when a workstation application accesses BSERVER via the BREQUEST program loaded at the workstation.

- Your application program issues a Btrieve request in the form of a function call. The actual call is implemented slightly differently in different languages. For simplicity, this manual will refer to a Btrieve call as a function call, or Btrieve call.)
- A short interface routine included in your application program packages the call parameters in a block of memory, saves the source stack, and makes the call to BREQUEST.
- BREQUEST packages the request into a network message, determines which server should receive the request, and routes the message to the BSERVER program resident at that server.
- BSERVER receives the network message, validates the parameters, and then executes the instruction. Depending on the nature of the instruction, this could involve a memory-only operation or an I/O operation to a system storage device. BSERVER then returns the results of the operation to the BREQUEST program at the workstation.
- BREQUEST returns the appropriate data and status code to the parameter variables or structures in your application's memory, restores the source stack, and returns control to your program.

If an application at a workstation makes Btrieve requests to both a local (nonshared) drive and a network (shared) drive, a copy of either Btrieve Single User or Btrieve for DOS 3.1 Networks must be loaded at that workstation, as well as BREQUEST. BREQUEST determines whether the request should be transferred to the local Btrieve servicing the nonshared files or to the BSERVER program that services the shared files at the server.

Figure 1.2 illustrates a sample configuration for a simple Novell network using NetWare Btrieve.

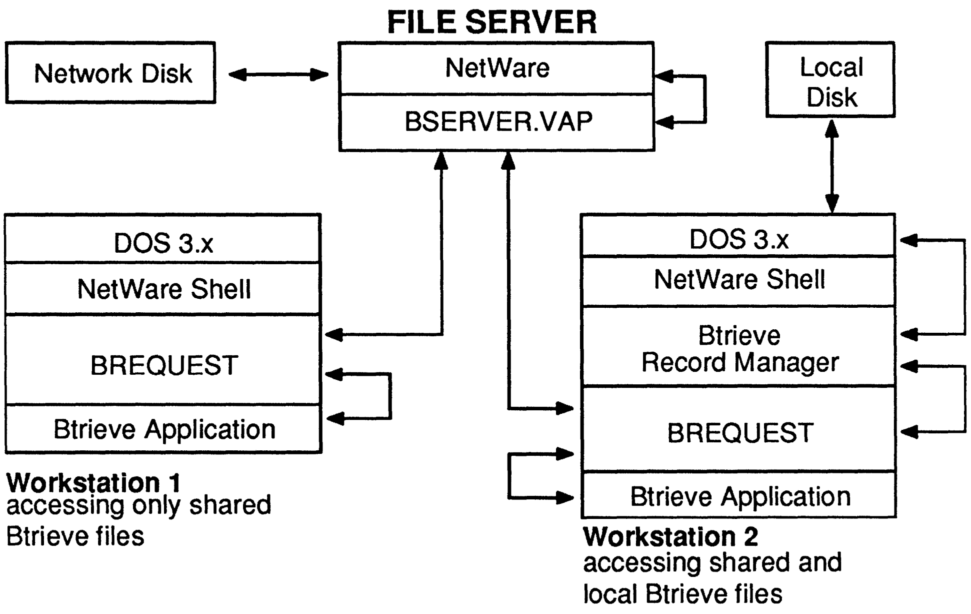


Figure 1.2
Network Configuration Using BSERVER.VAP
(Note that Workstation 2 accesses both shared and local files.)

Figure 1.3 illustrates a network with a multiple file server configuration. In this diagram, file servers A and B service the shared files on the network. Notice that all stations on the network can make Btrieve requests to either file server. The BREQUEST programs loaded at each workstation route the requests to the appropriate file server. Proper identification of the file servers and volumes is essential to the correct functioning of the system.

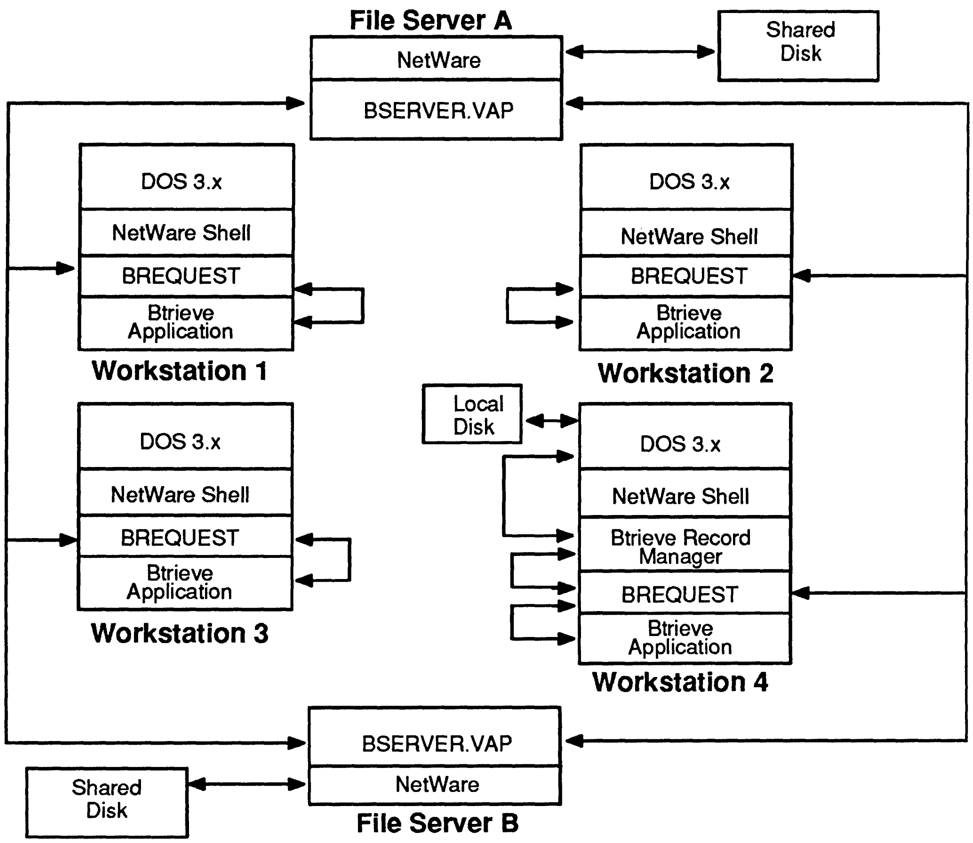


Figure 1.3
Network with Multiple File Servers
(Notice that BREQUEST loaded at each workstation can access each Btrieve file server. Workstation 4 is configured to access both shared and local Btrieve files.)

If you are using multiple file servers or an internetwork, all of the file servers do not have to be on line when you start BREQUEST at the workstations. BREQUEST recognizes new file servers or drives when you attach to a new file server or change your network drive mapping.

ACCESS TO BSERVER VIA BROUTER

NOTE:
If you are not developing a Value-Added Process (VAP) that accesses NetWare Btrieve, you may skip this section.

The following steps illustrate the flow of control when a workstation application calls a VAP which then accesses BSERVER via the BROUTER program.

- A workstation application issues a request to a VAP. The request may be formatted as a Btrieve call or in the form required by the VAP.
- An interface routine included in the application program packages the request into a network message, determines which server should receive the request, and routes the message to the VAP resident at that server.
- The VAP receives the network message, validates the parameters, and packages the call parameters as a Btrieve request in a block of memory. It then stores the client ID in the AX register and executes the 7B interrupt.
- BROUTER receives the Btrieve request, stores information about the origin of the call, and calls the copy of BSERVER active at the server where the file is stored.
- BSERVER processes the request and returns the results of the operation to BROUTER.
- BROUTER returns the appropriate data and status code to the parameter variables or structures in the VAP's memory and returns control to the VAP.
- The VAP returns the appropriate information to the application at the workstation.

If a workstation application makes Btrieve requests to a local (nonshared) drive and also to a VAP that calls Btrieve, a copy of either Btrieve Single User or Btrieve for DOS 3.1 Networks must be loaded at that workstation.

Figure 1.4 illustrates the flow of control when a VAP accesses Btrieve files using NetWare Btrieve.

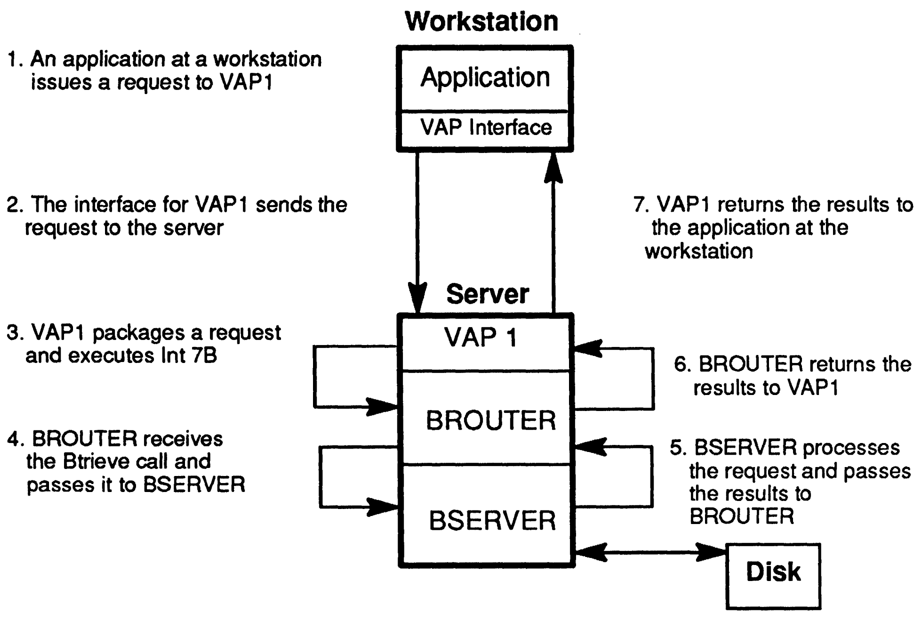

Workstation
1. An application at a workstation issues a request to VAP1
Application
VAP Interface
2. The interface for VAP1 sends the request to the server
7. VAP1 returns the results to the application at the workstation
Server
3. VAP1 packages a request and executes Int 7B
VAP 1
6. BROUTER returns the results to VAP1
BROUTER
4. BROUTER receives the Btrieve call and passes it to BSERVER
5. BSERVER processes the request and passes the results to BROUTER
BSERVER
Disk

Figure 1.4
Using Another VAP with BSERVER

CACHE BUFFERS

The cache is an area of memory that BSERVER reserves for buffering the pages that it reads from the disk. You define the size of the cache when you configure BSERVER. The cache is divided into a number of buffers, each of which is the size of the largest page your application will access. Generally, a larger cache improves performance, because it allows more pages to be in memory at a given time. (Refer to Chapter 3 for more information about configuring BSERVER.)

When your application makes a request for a record, BSERVER first checks the cache to see if the page containing the record is already in memory. If the record is already in cache, BSERVER transfers the record from the cache to your application program's data buffer. If the page is not in cache, BSERVER copies the page from the disk into a cache buffer before transferring the requested record to your application.

If every cache buffer is full when BSERVER needs to transfer a new page into memory, a least-recently-used algorithm (LRU) determines which page in the cache BSERVER needs to overlay. The LRU reduces processing time by keeping the most recently referenced pages in memory.

When your application inserts or updates a record, BSERVER first modifies the page in cache, and then writes the page to the disk. The modified page remains in cache until the LRU determines that it can be overlaid with a new page.

THE BTRIEVE CALL PARAMETERS

Btrieve requires certain information from your application in order to perform record and file management operations. Your application uses the Btrieve call parameters to specify the information Btrieve needs and to provide places for Btrieve to return information. On every Btrieve call, your application must pass to Btrieve every parameter required for the language you are using, even when Btrieve does not expect a value to be stored in that parameter. This section gives a general description of the parameters and how Btrieve uses them.

NOTE:
For specific information about how to use a parameter with a particular language, refer to Chapter 5, "Language Interfaces." For information about how Btrieve uses a parameter for a particular operation, refer to Chapter 6, "Btrieve Record Operations."

OPERATION CODE

The operation code parameter tells Btrieve which operation you want to perform. Your application must specify a valid operation code for every Btrieve call. The variable you specify to hold the operation code must be a 2-byte integer. Btrieve never changes the operation code value.

STATUS CODE

All Btrieve operations return a status code value, informing your application of any errors. A status code of 0 indicates that the operation was successful. For some languages, particularly C and Pascal, the Btrieve call is an integer function, and your application does not have to specify a separate parameter to hold the status code. If the language you use requires a separate status code parameter, specify a 2-byte integer to contain the return value.

POSITION BLOCK

Btrieve uses the position block parameter to contain positioning pointers and other information necessary for accessing a particular file. Your application uses the position block to identify to Btrieve the file you want to access on a particular operation. Btrieve expects the position block to be a 128-byte block of memory. Depending on the application language, the position block can be a string, an array, or part of the language's file buffer (as in BASIC).

Your application must assign a unique position block to each Btrieve file it needs to open and initialize the block to blanks or binary zeros before issuing the Btrieve Open operation. Your application should never write to a position block once it has been initialized and assigned to a file, unless you first issue a Btrieve Close operation for that file. You can then reinitialize the position block so that it can be used with another Btrieve file. Writing over the position block for an open Btrieve file can result in errors or file damage.

DATA BUFFER

The data buffer is a block of memory that holds specific kinds of information required by a Btrieve operation.

On operations that involve reading from or writing to a Btrieve file, the data buffer contains the records that your application transfers to and from the file. For example, when your application retrieves a record from a file, Btrieve reads the record from the file and then writes the record into the area of memory designated as the data buffer for that operation.

For other operations, the data buffer contains file specifications, definitions, and other information that Btrieve needs in order to process the operation. When your application issues a Create operation, for example, it constructs a data buffer that contains the specifications for the file you want to create, in the order that Btrieve expects them. Btrieve then reads the data buffer and creates the file according to the specifications.

Btrieve recognizes the data buffer as a series of bytes in memory. It does not distinguish any fields or variables as entities within the data buffer. You can define the data buffer to be any type of variable that your language supports: a structure, an array, or a simple string variable. For some versions of BASIC, the data buffer is the area of the file control block (FCB) defined by the FIELD statement.

DATA BUFFER LENGTH

For every operation that requires a data buffer, your application must specify the length of the data buffer, in bytes, to Btrieve. This is necessary because

- Btrieve allows you to define files that allow variable length records. You must specify how many bytes you want Btrieve to read from or write to those records.
- Btrieve does not recognize any of your program's data structures. Hence, it does not know implicitly how many bytes long the data buffer is. This creates the possibility of writing meaningless data to your files, or of returning more data than your data buffer can hold and overwriting the area of memory immediately after the data buffer.

Your application should define the data buffer as a 2-byte integer. On all operations, Btrieve writes a value back to the data buffer length parameter, even if that value is 0 (indicating that no data was returned). Therefore, you should always initialize the data buffer to the proper length for an operation before you issue the Btrieve call.

Use the following guidelines for initializing the value in the data buffer length parameter:

- When you read from or write to an existing file that contains fixed length records, specify a value equal to the record length defined for the file.
- When you read from or write to an existing file that contains variable length records, specify a value equal to the length defined for the fixed length portion of the record, plus the number of bytes that you want to read or write beyond the fixed length portion.
- When you issue any other operation, specify the exact length of the data buffer required for that operation. These requirements are included in the discussion of the individual Btrieve operations in Chapter 5 of this manual.

KEY BUFFER

Your application must pass a variable for the key buffer on every Btrieve call. Depending on the particular operation, your application may set the key buffer to a particular value, or Btrieve may return a value to the key buffer.

For some languages, Btrieve cannot implicitly determine the length of the key buffer. Therefore, you should always ensure that the variable you specify for the key buffer is long enough to hold the entire value required by the operation. Otherwise, Btrieve requests may destroy other data stored in memory following the key buffer.

KEY NUMBER

The key number parameter is always a signed 2-byte integer variable. For most Btrieve operations, this parameter tells the Record Manager which access path to follow for a particular operation. For other operations, your application uses the key number parameter to specify the file open mode, encryption, logical drive, or other information. Btrieve never returns an altered value to the key number parameter.

When you use the key number parameter to specify an access path for a file, the number must be in the range from 0 to 23, since Btrieve allows up to 24 keys or key segments in a file.

THE LANGUAGE INTERFACE

The language interface provides communication between your application program and Btrieve. Language interfaces are specific to certain languages, compilers, and environments.

When your application makes a Btrieve call, the interface makes preliminary checks on the parameters and checks to see that the Btrieve Record Manager is resident in memory. If the interface does not detect any errors, it executes the appropriate call for the operating environment, activating the Record Manager program.

NOTE:
Information about the BASIC, Pascal, COBOL, and C interfaces is included in Chapter 5 of this manual. For specific information about languages and compilers not included in Chapter 5, refer to the INTRFACE.DOC file on the Btrieve program diskette.

2 BTRIEVE FILE MANAGEMENT

This chapter describes Btrieve file structures and file management techniques. It also includes a discussion of integrity processing and concurrency controls.

BTRIEVE FILE CONCEPTS

A *file* is the highest level database entity that you can access using Btrieve. You create Btrieve files and define their characteristics either by using the CREATE utility described in Chapter 4, or by issuing a CREATE operation from your application program. Btrieve allows a maximum file size of approximately four billion characters.

Your application can specify filenames in either of the following formats:

\\\\ <*servername*> \\ <*volumename*> \\ <*pathname*> \\ <*filename*>

or

<*drive*>: \\ <*pathname*> \\ <*filename*>

Always terminate the filename with a blank or binary zero.

PHYSICAL FILE CHARACTERISTICS

Btrieve files consist of a series of *pages*. A page is the unit of storage which Btrieve transfers between memory and disk during disk I/O. A Btrieve file can be composed of as many as three different types of pages: index pages, data pages, and a header page (or File Control Record).

You specify a fixed size for each page when you create the file. The page size is always some multiple of 512 bytes, up to 4096 bytes. If you need more than eight key segments, you must specify a size of 1024 bytes or greater. The optimum page size for your application depends on the number of key segments in your file and the length of your data records. See "Determining Record Length and Page Size" beginning on page 2-12 for more information.

HEADER PAGE (FILE CONTROL RECORD)

Every Btrieve file has a single *header page*, or File Control Record (FCR), that is always the first page in the file. The FCR contains information about the file, such as the file size, the page size, the alternate collating sequence (if any), and other characteristics of the file.

DATA PAGES

Btrieve stores the records your application inserts into a file on *data pages*. Btrieve uses two kinds of data pages: fixed length record pages and variable length record pages.

If a file does not allow variable length records or data compression, its data pages will all be fixed length record pages. Each data page may contain one or more data records. The number of data records per page depends on how you define the record length when you first create a file. Btrieve will not split the fixed length portion of a record across data pages.

If a file allows variable length records or data compression, or both, it will contain both fixed length record pages and variable length record pages. The fixed length record pages contain only the fixed length portions of the records. The variable length record pages contain only the variable length portions of the records. If the variable portion of a record is longer than the defined page size for the file, Btrieve will split the the variable portion over multiple pages.

INDEX PAGES

The *index pages* contain the key values for accessing data records. Generally, index pages contain many different key values. Each key value on the page has a record address (or two addresses when you specify duplicate keys). Btrieve uses these addresses to retrieve records containing the key value.

B-TREES

Btrieve keeps all indexes to the data records in the form of *B-trees*. A B-tree is a data structure featuring quick access and efficient use of disk space. Once a B-tree is created, no periodic maintenance is required. A separate B-tree is created for each key you define within a file.

DYNAMIC EXPANSION

Btrieve allocates disk space as needed. If there is not enough room in the current allocation when your application inserts new records, Btrieve dynamically allocates additional data and index pages to the file. Btrieve also updates directory structures to reflect the new file size.

Once space has been allocated to a file, that space remains allocated as long as the file exists. To reduce the space required for a file from which numerous records have been deleted, you can create a new file with the same characteristics as the original file, and then either write a small application which reads the records from the original file and inserts them into the new file, or use the BUTIL –COPY utility described in Chapter 4. You can then delete the original file from the disk.

FREE SPACE UTILIZATION

When you delete a record, the space it formerly occupied is put on a list of free space. When your application inserts new records Btrieve uses the free space instead of allocating additional pages to the file. Btrieve's method of reusing free disk space eliminates the need to reorganize files to reclaim disk space.

BTRIEVE FILE TYPES

Btrieve allows you to define three different types of files: standard, data-only, and key-only. The file types are differentiated by the types of pages they contain. You can define and create any file as either of the two types.

STANDARD BTRIEVE FILES

A *standard* Btrieve file contains a header page followed by index pages and data pages. You can define a standard Btrieve file for use with either fixed or variable length records.

Because standard Btrieve files contain all of the index structures and data records, Btrieve can dynamically maintain all of the index information for the records in the file. You can use any of the Btrieve record retrieval operations to access the information stored in a standard Btrieve file.

DATA ONLY FILES

Btrieve allows you to create files that hold only data. When you create a *data only* file, you do not specify any key characteristics, and Btrieve does not allocate any index pages for the file. This results in smaller initial file sizes than for standard Btrieve files.

When your application inserts records into the file, Btrieve stores them in the chronological order of insertion. (The chronological order can be disturbed once you delete records and insert new ones.) Btrieve does not maintain or create any index pages as the records are inserted. At this point, you can access the records using only the Step operations, which use physical location to find records.

At any time after you have created a data only file, you can add an index using the Create Supplemental Index operation. Once you have added a supplemental index successfully, you can retrieve records with Get operations, using the supplemental index.

KEY-ONLY FILES

Key-only files contain only a header page followed by one or more index pages. In a key-only file, the entire record is stored with the key, so no data pages are required. A common use for key-only files is as an external index for a standard Btrieve file.

The following restrictions apply to key-only files:

- The file may contain only a single key.
- The maximum record length you can define is 255 bytes.
- The values stored in the file cannot be updated or deleted once they have been inserted. The only valid operations that you can perform on the file are Open, Close, Stat, Insert, and the Get and Step operations.

RECORDS

A *record* represents a set of logically associated data items in a Btrieve file. It is the unit transferred between your application program and the Record Manager in a single operation.

There is no restriction on the number of records allowed in a Btrieve file. A record can have a fixed length (a "fixed-length record"), or it can consist of a fixed length portion followed by a variable length portion (a "variable length record"). Records in files that use data compression are always variable length. All the keys you define for the file must be located within the fixed length portion of the record.

The maximum length of the fixed portion of a record depends on the physical page size you define for the file and the number of duplicate keys you defined for the file. Btrieve allows the fixed length portion of a record to contain up to 4090 bytes. You can define variable length records up to 64K in length.

See the section called "Determining Page Size and Record Length" beginning on page 2-12 for more information on maximum record length.

VARIABLE LENGTH RECORDS

When you create a Btrieve file, you can specify that you want the file to use *variable length* records. That is, the length of each record in the file can vary. If you specify that you want the file to use data compression, the file will use variable length records by default.

When you create a Btrieve file that uses variable length records, you specify a length in bytes for the fixed length portion of the record. The length you specify for the fixed length portion of the record is the minimum length the record can be. You do not define the maximum record length to Btrieve. Anything beyond the minimum length (up to 64K total bytes) is optional.

When you insert or update a record, your application uses the data buffer length parameter to specify the length of the record to Btrieve. If the data buffer length you specify is less than the defined fixed length portion of the record, Btrieve returns an error status and does not insert or update the record.

When you read a variable length record and specify a data buffer shorter than the fixed length portion, Btrieve returns an error status and does not return any data to your application. If you specify a data buffer length equal to or longer than the fixed length portion, but not as long as the record you want to read, Btrieve returns the number of bytes of data you requested and a status code informing you that Btrieve did not return the entire record.

If you specify a data buffer length longer than the record you want to retrieve, Btrieve returns only the number of bytes in the actual record. In all cases, Btrieve informs your application of the number of bytes it returned by setting the data buffer length parameter to that value.

Btrieve stores variable length records on their own data pages, separate from the fixed length portion of the record. Btrieve leaves a certain amount of free space, called the free space threshold, on each page where variable length records are stored. This allows the records to expand when they are updated, and minimizes fragmentation of a record across multiple pages. When you create a file, you can specify the amount of free space you want Btrieve to leave on each variable length page.

KEYS AND KEY ATTRIBUTES

Btrieve uses *keys* to identify specific records in a file. By using a key, Btrieve can efficiently select the record you want from the entire collection of records in a file. Because Btrieve has no way of knowing the exact structure of the records in each file, you define each key by identifying its offset in bytes from the beginning of the record, and specifying the number of bytes you want to use for that key.

For example, suppose a particular key begins at the eighth byte of the record and extends for four bytes. When you insert the record into the file, Btrieve will read four bytes, beginning with the eighth byte, and use the value it finds there to position the record in the index. The keys you define may overlap each other in the record.

When you create a file, you can specify six different attributes for each key in the file. The six key attributes are duplicate, modifiable, segmented, descending, null, and manual.

DUPLICATE KEYS

You can define a *duplicate* key to identify a subset of records, all of which can contain the same value for a particular key. If you specify that a key not allow duplicates, Btrieve does not allow an application to add multiple records to the file with the same value in the key field. Btrieve stores duplicate key values in the chronological order of insertion into the file. If one segment of a segmented key allows duplicates, all of the segments must allow duplicates.

For example, in a file containing customer records you can define the zip code field as a duplicate key so that many different records can contain the same value for zip code. However, if you also make the field for the customer number a key field, you probably would not want to allow duplicates, since each customer should have a unique number.

MODIFIABLE KEYS

You can also define a key as *modifiable*. Btrieve then allows your application to update an existing record and change the value of the key field. For example, if account balance is a key field, you may allow a program to modify the value of the field as the customer makes purchases and payments. However, you probably would make a field such as the account number a non-modifiable key since the customer's account number should never change. If one segment of a key is modifiable, all of the segments must be modifiable.

SEGMENTED KEYS

Keys can consist of one or more *segments* in each record. A segment generally corresponds to a field in the record and the segments do not have to be contiguous. The total length of a key is equal to the sum of the length of the key segments. The maximum total length is 255 characters. Different key segments may overlap each other in the record.

A Btrieve file is restricted to a maximum number of key segments rather than a maximum number of keys. The maximum number of key segments allowed depends on the page size. You can define as many as eight key segments if the page size is 512 bytes. For a file with a page size of 1024 bytes or greater, you can define up to 24 key segments. (See "Determining Record Length and Page Size" on page 2-12 for more information about page size.)

A file with a page size of 512 bytes or greater may contain one key with eight segments, eight keys with one segment each, or any combination in between. If a file has a page size of 1,024 bytes or greater, it may contain one key with 24 segments, 24 keys with one segment, or any combination in between.

The key type can be different for each segment in the key. The sort order, either ascending or descending, can be different for each segment as well.

DESCENDING KEYS

Btrieve normally orders key values in ascending order (lowest to highest). However, you can specify that Btrieve order the key values in descending order (highest to lowest) when you define the key segment.

Btrieve operations produce results based on the order of the key values in the index, not on the result of a comparison of key values. For example, when you perform a Get Greater operation on a descending key, Btrieve returns the record corresponding to the first key value that is lower than the key value you specify in the key buffer. Likewise, when you perform a Get Less Than operation using a descending key, Btrieve returns the record with the next higher key value than the one you specify in the key buffer.

NULL KEYS

You can direct Btrieve to exclude certain records from an index by defining *null* keys. When you define a null key, you specify a value that you want Btrieve to recognize as the null value for that key. If every byte of the key contains the null value, Btrieve will not include the record in the index for that key. If you define one segment of a key with the null attribute, you must define all segments of that key with the null attribute.

Btrieve will treat a key as null only if every byte in the key contains the null value. If the key is segmented, every byte in each segment must contain the null value in order for Btrieve to exclude the key from the index. You can define different null values for different segments in a segmented key. The most commonly used null values are ASCII blank (hex 20) and binary 0 (hex 0).

You can use a null value in a key when the data for the key is unavailable, or when you do not want Btrieve to include that record in the index for that key. Null keys allow you to avoid searching through meaningless records in an index path, and eliminate the overhead time required to update the index each time a blank key is inserted. When the data becomes available, or when you want to include the record in the index, you can update the record, replacing the null value with another value.

MANUAL KEYS

The *manual* key is a modified form of the null key, and can be used to exclude particular records from the index. Like a null key, you must define a value that indicates whether you want Btrieve to include the record in the index. If you define one segment of a key with the manual attribute, you must define all segments of that key with the manual attribute.

Manual keys have all the attributes of a null key, with one exception. In a manual key, if every byte of any one segment contains the null value, Btrieve will exclude the key from the index.

For example, you can use a manual key segment as a flag to indicate whether the key is to be indexed. If the segment contains only the null value that you defined for that segment, Btrieve will not include the key in the index, even though the rest of the segments in the key contain non-null values. By updating the flag segment with any value other than the specified null value, you can instruct Btrieve to include the record in the index.

You can define a key as being both null and manual. In this case, the null value will always be the same for both the null and manual attributes. However, the manual attribute will always override the null attribute.

KEY TYPES

Btrieve key types fall into two general categories—standard and extended. Standard key types are binary and string. Extended key types encompass a wider set of the most commonly used data types.

STANDARD KEY TYPES

Each standard type key segment is either a binary or string type. Btrieve sorts standard binary key values as unsigned integers. Internally, Btrieve compares string keys on a byte-by-byte basis from left to right. It compares binary keys a word at a time from right to left because the high and low bytes in an integer are reversed by the Intel 8086 family of processors.

Btrieve sorts string keys according to their ASCII value. However, you can specify an alternate collating sequence for string keys. An alternate collating sequence could be used to sort keys according to a non-English alphabet such as German, Swedish, or Finnish character sets.

EXTENDED KEY TYPES

Extended key types allow Btrieve to recognize and collate key values based on the internal storage format of 13 of the most commonly used data types. This capability allows you greater flexibility in designing the indexes of your Btrieve files.

Some of the types Btrieve supports are IEEE floating point, zero terminated string, packed decimal, autoincrement, and integer (signed binary). Appendix G lists and describes the extended key types. Two of the extended key types, string and unsigned binary, are the same as the two standard key types.

INDEXES

An *index* is a structure in a Btrieve file that contains the key values and maintains them in a sorted order. Btrieve dynamically maintains the indexes in a balanced B-tree structure. When you insert, update, or delete a record Btrieve adjusts all the indexes for the file to reflect the latest changes in the key values contained in the records.

When you create a Btrieve data file, you can define one or more keys for Btrieve to use to build indexes. Any key you define when you create a file is called a *permanent* index, because it exists for the life of the data file.

Btrieve also allows you to define internal indexes for a file after the file has been created. These indexes are referred to as *supplemental* indexes. Once you have created a supplemental index, Btrieve maintains it as the data in the file changes, just as it does for a permanent index. Positioning rules for a supplemental index are the same as those for a permanent index. The total number of permanent and supplemental indexes or segments for a file cannot exceed 24. Supplemental indexes differ from permanent indexes in two ways.

Unlike a permanent index, you can delete, or "drop," a supplemental index when it is no longer needed by your application. The space in the file that was used by the index is freed for data or for other index pages, and Btrieve no longer maintains that index.

Btrieve collates duplicate key values for a supplemental index in a different order from those in a permanent index. In a permanent index, Btrieve collates duplicate values in the chronological order in which the records are inserted into the file. In a supplemental index, Btrieve enters duplicate values into the index in the order in which their corresponding records are physically stored in the file.

You can also specify that the supplemental key use an alternate collating sequence. If an alternate collating sequence already exists in the file, Btrieve will use that alternate collating sequence for the new supplemental key key, if the key is so defined. If no alternate collating sequence exists, you can include one with the definition of the new supplemental index.

DISK UTILIZATION

You can reduce the amount of disk space a file uses by reducing the amount of unused space on data and index pages. The following sections describe how to achieve the most efficient use of disk space and how to estimate your file's total size.

NOTE:
The following discussion and the formulas for determining file size do not apply to files that use data compression, since the record length for those files depends on the number of repeating characters in each record.

DETERMINING RECORD LENGTH AND PAGE SIZE

When you create a file, you must specify the physical page size and the logical record length. The following sections will help you determine the optimum page size. For files that allow variable record lengths, logical record length refers to the fixed length portion of the record.

RECORD LENGTH

First, determine the logical record length (how many bytes of data you need to store in the fixed length portion of each record). You specify this value when you create the file. You must then determine the physical record length (how many bytes of data, including Btrieve's overhead, are required to store the fixed length portion of each record).

For files with fixed length records and no duplicate keys, the physical record length equals the logical record length.

The number of bytes required to store each record on the disk depends on how many duplicate keys you assign in the file definition. Btrieve stores eight extra bytes of information with the record for each key that allows duplicates.

Thus, to determine your file's physical record length, add eight bytes to the logical record length for each key that allows duplicates. Add an additional four bytes if the file allows variable length records, or six bytes if the file allows blank truncation.

DATA PAGE SIZE

Btrieve stores as many records as possible in each data page in the file. Each data page requires six bytes for overhead information. The fixed length portion of records do not span pages, so there will be unused space in the file if the page size minus six bytes is not an exact multiple of the physical record length. Of course, if you use the data compression feature, there is no way to precisely know what the stored length of each record will be.

To maximize disk utilization, select a page size that can buffer your records with the least amount of unused space. Page size must always be some multiple of 512 bytes, up to 4096 bytes. Larger page sizes usually result in more efficient use of disk space.

For example, suppose you need to store 320 bytes of data in each record and one of the keys in the file allows duplicates. The logical record length you specify when creating the file is 320. The file's physical record length is 328 bytes because of the eight-byte overhead for one duplicate key.

If you select a page size of 512, only one record can be stored per page and 178 (512 – 6 – 328) bytes of each page would be unused. However, if you select a page size of 1024, three records can be stored per page and only 34 (1024 – 6 – (328 * 3)) bytes of each page would be unused.

ESTIMATING FILE SIZE

You can estimate the number of pages required to store a Btrieve file with the formulas below. These formulas are based on the maximum storage (worst case) required.

The B-tree index structure guarantees at least 50% utilization of the key pages. Therefore, the key page calculations shown below multiply by two the minimum number of key pages required to account for the worst possible case.

Most files will require less space, but you should consider these maximum sizes when estimating file size. These formulas do not take into consideration the increased index page utilization available if you load Btrieve with the Index Compaction option.

These formulas apply to files with fixed length records. Any variable length portions of records are packed in additional pages.

Calculate the number of data pages using the following formula:

Data Pages = Number of Records /
((Page Size – 6) /
(Record Length + (8 * Number of Duplicate Key
Fields)))

Calculate the number of index pages for each defined key field using the following formula:

For key fields that allow duplicates:

Index Pages = (Number of Unique Key Values /
((Page Size – 12) / (Key Length + 12))) * 2

For key fields that don't allow duplicates:

Index Pages = (Number of Key Values /
((Page Size – 12) / (Key Length + 8))) * 2

After you calculate the individual index pages, calculate the pages required for the file by adding the individual index pages, the data pages, and an additional page for the File Control Record as follows:

Total file pages = 1 + Data Pages + Index Pages(1)
+ Index Pages(2) + ...

The maximum number of bytes required to store the file can then be calculated as

File size in bytes = Total file pages * Page size

FILE PREALLOCATION

The speed of file operations can be enhanced somewhat if a data file occupies a contiguous area on the disk. These speed gains are most noticeable on very large files. Btrieve allows you to preallocate up to 65,535 pages to a file when you create a data file.

In order to preallocate contiguous disk space for a file, the device on which you are creating the file must have the required number of bytes of contiguous free space available. Btrieve preallocates the number of pages you specify, whether or not the space on the disk is contiguous. If there is not enough space on the disk to preallocate the number of pages you specify, Btrieve returns a disk full status and does not create the file.

Use the formulas described in the previous section to determine how many data and index pages the file requires. You should round any remainder from this part of the calculation to the next highest whole number.

When you preallocate pages for a file, that file actually occupies that area of the disk. No other data file can use those portions of the disk until you delete or replace the file that has preallocated space.

As you insert records, Btrieve uses the preallocated space for data and indexes. When all of the preallocated space for the file is in use, Btrieve expands the file as new records are inserted.

When you issue a Btrieve Stat operation, or run the BUTIL –STAT utility, Btrieve returns the difference between the number of pages you allocated at the time you created the file and the number of pages that Btrieve has currently in use. This number will always be less than the number of pages you specify for preallocation because Btrieve considers a certain number of pages to be in use when a file is created, even if you have not inserted any records.

Once a file page is in use, it remains in use for the life of the file, even if you delete all of the records stored on that page. Thus, the number of free pages the Stat operation returns will never increase. When you delete a record, Btrieve maintains a list of free space in the file and re-uses the available space when you insert new records.

If the number of free pages returned by the Stat operation is zero, it does not always mean that there is no free space in the file. The number of free pages can be zero if one of the following is true:

- You did not preallocate any pages to the file.
- All of the pages that you preallocated were in use at one time or another.

CONSERVING DISK SPACE

Btrieve offers two optional methods for reducing the space occupied by your data files: blank truncation and data compression. The blank truncation method applies only to files that use variable length records. You can use data compression for any Btrieve file.

BLANK TRUNCATION

When you define a file that allows variable length records, you can specify that Btrieve use a blank truncation method for storing the records in order to conserve disk space. If you choose to truncate blanks, Btrieve won't store any trailing blanks in the variable length portion of the record when it writes the record to the file. Blank truncation has no effect on the fixed length portion of a record. Btrieve does not remove blanks that are embedded in the data.

When you read a record that contains truncated trailing blanks, Btrieve expands the record to its original length. The value Btrieve returns in the data buffer length parameter includes the expanded blanks, if any. Blank truncation adds two bytes of overhead to the record.

DATA COMPRESSION

When you create a Btrieve file, you can specify whether you want Btrieve to compress the data records when it stores them in the file. Data compression can result in a significant reduction of the space needed to store records that contain many repeating characters.

You should consider using Btrieve data compression when

- The records to be compressed are structured so that the benefits of using data compression are maximized;
- Your application program does not require the extra memory used by Btrieve for the compression buffers;
- The need for better disk utilization outweighs the possible increased processing and disk access times required for compressed files.

When you perform record I/O on a compressed file, Btrieve uses a compression buffer to provide a block of memory for the record compression or expansion process. To have enough memory to compress or expand a record, Btrieve requires enough buffer space to store twice the length of the longest record your application inserts into the compressed file. This requirement can have an impact on the amount of free memory left in the computer after Btrieve loads. For example, if the longest record your application writes or retrieves is 2,000 bytes long, Btrieve will require 4,000 extra bytes of memory in order to compress and expand that record.

Because the final length of a compressed record cannot be determined until the record is written to the file, Btrieve always creates a compressed file as a variable length record file. Since the compressed images of the records are stored as variable length records, individual records may become fragmented across several file pages if your application performs frequent insertions, updates, and deletions. The fragmentation can result in slower access times, since Btrieve may need to read multiple file pages to retrieve a single record.

The data compression option is most effective when each record has the potential for containing a large number of repeating characters. For example, a record may contain several fields, all of which may be initialized to blanks by your application when it inserts the record into the file. Compression will be more efficient if these fields are grouped together in the record, rather than being separated by fields containing other values.

To use data compression, the file must have been created with the compression flag set. For more information about the compression flag, refer to the discussion of the Create operation on page 6-10, or to the discussion of BUTIL description files beginning on page 4-4.

POSITIONING

Btrieve allows you to retrieve records from a file based on either the record's physical address within the file or a key value contained in the record.

When you insert a record into a Btrieve file, Btrieve writes the record into the first free space available in the file, regardless of any key values contained in the record. This location is referred to as the "physical location," or "address," of the record. The record will remain in this location until your application deletes it from the file.

When Btrieve writes the record into the file, it also updates the defined access paths (indexes) with the appropriate key values in the record. The place in each index where the record's key values exist is referred to as the "position" of the record. The position of a record in an index can change as new records are inserted and existing records are either deleted or their key values are updated. Thus, there is not necessarily a correspondence between the physical address of a record and its position in an access path.

RECORD RETRIEVAL BY PHYSICAL LOCATION

Your application can use the four Step operations provided by Btrieve to retrieve records based on their physical location in the file. For example, the Step First operation retrieves the record that is stored in the first (lowest) physical location in the file. The Step Next operation retrieves the record stored in the next higher physical location, and the Step Previous operation retrieves the record stored in the next lower physical location in the file. The Step Last operation retrieves the record that is stored in the last (highest) physical location in the file.

Retrieving a record by its physical location is usually faster than retrieving a record based on a key value because Btrieve does not have to update any positioning information for an index and because the next or previous physical record is very likely already in Btrieve's memory cache.

The Step operations are useful for traversing a data file quickly if your application does not need to retrieve the records in a specific order. Because the Step operations use only physical location to retrieve records, they are not useful for finding records that contain a specific value or for retrieving records in any specific order.

Your application can establish logical position in an index after issuing Step operations by using the following method:

- Retrieve the desired record by issuing one of the Step operations.
- Issue a Get Position operation to retrieve the 4-byte physical address of the record.
- Issue a Get Direct operation, passing Btrieve the 4-byte position and the key number on which you want to establish position.

After performing the above operations, your application can proceed to retrieve records based on their key values.

RECORD RETRIEVAL BY KEY VALUE

Your application can use the Get operations to retrieve records based on their key values for a specified access path. The appropriate Get operation can retrieve a specific record from a file, or retrieve records in a certain order.

For example, the Get First operation retrieves the first record in an access path. The Get Last operation retrieves the last record in an access path. Some Get operations, such as Get Equal or Get Less Than, return a record based on a key value your application specified in the key buffer parameter.

Get operations establish Btrieve's position in an index. Your application can change from one index to another by using the following method:

- Retrieve a record by issuing one of the Get operations.
- Issue a Get Position operation to retrieve the 4-byte physical address of the record.
- Issue a Get Direct operation, passing Btrieve the 4-byte physical address and the key number to which you want to change.

THE POSITION BLOCK

Btrieve maintains positioning information associated with each open file. It stores this positioning information in a 128-byte block of memory which passes between the Record Manager and your application program. This area of memory is known as the "position block."

Specifically, the position block contains the following:

- The access path identifier (the key number)
- The access path index pointers
- Three record pointers

The three record pointers are the following:

- The *previous* record pointer points to the record preceding the current record in the index.
- The *current* record pointer points to the most recently retrieved record.
- The *next* record pointer points to the record following the current record in the index.

Btrieve updates the position block on every operation to reflect the new index position within the file. Btrieve uses positioning information to read sequentially through the file on a given access path.

Your application should maintain a 128-byte position block for every Btrieve file that is open. Your application should never write to the position block, since this could result in a lost position error or in damage to the Btrieve file.

INTEGRITY PROCESSING

Btrieve's normal operating mode provides several levels of automatic data recovery. Btrieve uses the Transaction Tracking Service to protect individual Insert, Update, and Delete operations against system failure if the TTS is installed and the file is flagged as transactional. If the file is not flagged transactional or if TTS is not active, Btrieve guards file integrity through a system called pre-imaging. Additionally, Btrieve protects sets of related operations on multiple files with a process called transaction control. The processing required for both of these features is described in the following paragraphs.

You can disable any level of automatic recovery with the performance options described under "Accelerated Access" on page 2-25. If a file is damaged because of a system failure while Btrieve is running with the integrity processing disabled, you can use the RECOVER utility to recover data.

PRE-IMAGING

The Record Manager may need to perform several physical page updates to process a single logical request from an application. If Btrieve processing is interrupted by a system failure, files may become inconsistent, possibly introducing invalid access paths which could make data retrieval impossible.

NOTE:
The discussion of pre-imaging in this section applies only to files that are <u>not</u> flagged transactional or that are stored on a file server where TTS is <u>not</u> active.

Btrieve guards against inconsistent file states with a process called pre-imaging. If processing is interrupted, Btrieve automatically restores the file to its state just prior to the incomplete request.

Btrieve creates a pre-image file the first time you update the data file or access it during a transaction. The temporary file has the same name as the file it is protecting, except that the filename extension is always .PRE. For example, if you opened the ACCTID.DTA file, the temporary filename would be ACCTID.PRE. Btrieve uses this temporary file to track changes to the actual file during the scope of a request. Don't create file names with the extension .PRE since they will invalidate the recovery process.

When you update a file, Btrieve stores the updates in the local cache buffers. Before it changes a page, it writes a copy of that page to the pre-image buffers.

At the end of the operation, Btrieve first writes the pre-image buffers to the pre-image file on the disk. Then it writes the updates to the data file on the disk. If either the cache buffers or the pre-image buffers fill up before the end of the operation, Btrieve writes the information stored in the buffers to the disk in order to free the buffer space. No matter when Btrieve writes the information to the disk, it always writes the data in the pre-image buffers to the disk before it updates the actual data file.

Each Btrieve file you access must have its own pre-image file. Since the name of the pre-image file is the same as the Btrieve file (except for the extension), do not create multiple Btrieve files with the same names but different extensions. For example, do not use a naming scheme like INVOICE.HDR and INVOICE.DET for your Btrieve files. Otherwise, Btrieve tries to use INVOICE.PRE for both files and automatic recovery becomes impossible.

Once Btrieve creates a pre-image file, do not delete it unless the Btrieve file has been successfully closed. Without the pre-image file, Btrieve cannot recover data after an abnormal termination.

NOTE:
Pre-imaging does not eliminate the need to back up files, since pre-imaging cannot recover from a damaged disk. Therefore, it is imperative that backups be made to protect against the catastrophic loss of a database because of a hardware failure.

TRANSACTION TRACKING SYSTEM (TTS)

Btrieve uses TTS to protect and guarantee individual write operations to Btrieve files. In order to use this facility, two conditions must be met:

- TTS must be active at the file server.
- The Btrieve files must be flagged as transactional.

When you install Btrieve, you must specify whether files should automatically be flagged as transactional as they are created. If you created the file on another system, and copied or restored it to your network disk, you must flag it as transactional using the NetWare FLAG utility. Refer to the *NetWare Command Line Utilities* manual for more details about the FLAG utility.

If processing at the file server is interrupted during a write operation, files that are flagged transactional may become inconsistent and could contain invalid access paths. When the file server is rebooted, TTS restores any inconsistent files to the state they were in just prior to the incomplete write operation.

TRANSACTIONS

You can obtain a higher level of data integrity than the individual file consistency provided by either TTS or pre-imaging alone if you define logical transactions within your program. Your program may perform Btrieve operations on up to 12 different files within a single transaction.

When you bracket a set of Btrieve operations with Begin and End Transaction calls, Btrieve will not complete any of the operations unless it is able to successfully complete all of them. If the system fails during an incomplete transaction, Btrieve automatically removes ("rolls back") all of the operations performed during the transaction, whether or no TTS is active.

Btrieve uses a transaction control file to keep track of the files involved in a transaction. It creates the transaction file, \SYS\SYSTEM\BTRIEVE.TRN, if you specify transaction processing in the BSETUP utility. See "Configuring and Installing Btrieve" in Chapter 3 of this manual.

When you restart the file server after a system failure, Btrieve looks at the transaction file to determine if recovery procedures are necessary. In most cases, Btrieve performs transaction recovery the first time you open the involved files after the failure. However, if the system fails while Btrieve is processing an End Transaction call, it performs some recovery when you restart the file server. In this case, all files involved in the incomplete transaction at that file server must be online when you restart.

When you are running Btrieve applications that use transactions, follow the specifications below:

- Always ensure that the Btrieve files accessed within transactions are online when you start the Record Manager.
- Never delete or modify the transaction file or any files with the extension ".PRE".

For files that are not flagged transactional, Btrieve uses the pre-image files to implement transaction-level integrity. The pre-image files continue to grow for the duration of the transaction (until an End Transaction or Abort Transaction operation) , not just during a single operation. Therefore, the pre-image files require a larger amount of disk space when you are using transactions in your applications. The more Update, Delete, or Insert operations your application performs during a single transaction and the larger the page size, the larger the pre-image file will become.

TRANSACTIONS AND TTS

If TTS is active and the files you access during a transaction are flagged transactional, Btrieve uses the TTS facility to implement transaction level integrity. If an error occurs during a write operation to a file within a transaction, Btrieve signals TTS to roll back all operations since the beginning of the transaction. At this point, the transaction is effectively aborted, and Btrieve returns an error status to your application that reflects the cause of the error. Issue an Abort Transaction operation (see Chapter 6) to exit the transaction.

If your application issues an Abort Transaction operation because of an operator request or other internal reason, Btrieve removes the operations since the beginning of the transaction from the files using the facilities provided by TTS.

You can mix files that are flagged transactional with nontransactional files within a logical Btrieve transaction. If an error occurs or if your application aborts the transaction, Btrieve uses TTS to roll back the transactional files and pre-imaging to roll back the nontransactional files.

When you mix transactional and nontransactional files, the pre-image files for the nontransactional files will grow for the duration of the transaction, requiring a larger amount of disk space than if you only use files that are flagged transactional.

ACCELERATED ACCESS

Both methods of automatic data recovery, TTS and pre-imaging, require processing time overhead. There may be circumstances in which you feel such overhead is not warranted. Btrieve allows you to disable pre-imaging for files that are not flagged transactional by opening them in accelerated mode. This decreases the time required for Update, Delete, and Insert operations, and is especially useful if you are loading a large number of records. Accelerated mode can be advantageous in environments that guard against system failures with an uninterruptible power supply, or in systems in which the risk of system failure from other causes is low.

When you open a file in accelerated mode, Btrieve does not perform writes to the operating system until its cache is full and the least-recently-used algorithm controlling the I/O buffer cache selects a buffer for reuse. Btrieve never writes to the pre-image file unless the pre-image buffers are entirely filled during a single operation. In addition, Btrieve does not perform the reset disk operations that cause the operating system to flush its cache buffers to the disk. It does not close and reopen the file each time it physically expands in order to flush the directory structure.

If you open a file in accelerated mode, you cannot assume that any of your updates have been written to the disk until you perform a Close operation. If the system fails while you are accessing a file in accelerated mode, Btrieve cannot automatically recover the file the next time you open it. In this case, you must use the BUTIL –RECOVER utility to recover your data records to a sequential file. The data you recover may not reflect all of the operations you performed before the system failed.

Once a workstation successfully opens a file in accelerated mode, others can open the file only if they also use accelerated mode. Any attempt by another workstation to open the file in a different mode results in an incompatible mode error status.

NOTE:
Accelerated mode has no effect on files that are flagged transactional when the TTS system is active. Also, opening a file in accelerated mode does not affect Btrieve's speed during nonwrite operations.

CONCURRENCY CONTROLS

Btrieve provides three different methods to resolve conflicts that can occur when two workstations attempt to update or delete the same records at the same time: transaction control, passive concurrency, and record locks. You can use these methods individually or combine them within a single program.

TRANSACTION CONTROL

Whenever your program accesses a file within a transaction, Btrieve locks the file for that workstation until the transaction ends or aborts. Other workstations may read the file, provided the reads are not performed within a transaction. However, no one else can access the file within a transaction.

Btrieve normally performs wait locks. If two workstations try to access the same file within a transaction, Btrieve will not allow the second workstation to access the file until the first workstation completes its transaction.

However, if you specify nowait locks when you begin the transaction, Btrieve will perform nowait locks during the transaction. If two workstations try to access the same file within a transaction, the second workstation receives a file busy status from Btrieve.

Since a transaction temporarily locks a file against updates, your Btrieve application should follow these important guidelines:

- A program should never wait on keyboard input during a transaction. No other workstations will be able to update the file(s) accessed within that transaction until the operator responds and the transaction is terminated.
- To avoid deadlock, all transactions that access the same set of files should be started with the nowait option, or else they should access the files in the same order. Otherwise, it is possible to get into a situation in which each workstation is waiting for access to files that the other workstation has already locked.

Figure 2.1 illustrates how two workstations would interact while running the same application that reads and updates two files within a wait transaction. The steps are numbered to indicate the order in which the operations occur.

Station 1 Application	Station 2 Btrieve	Station 2 Application
(1) Open File 1		
		(2) Open File 1
(3) Open File 2		
		(4) Open File 2
(5) Begin Tran		
		(6) Begin Tran
(7) Read File 1		
		(8) Read File 1
	(9) Delay since Station 1 has locked File 1	
(10) Read File 2 (11) Update File 1 (12) Update File 2 (13) End Tran		
	(14) Read completes	
		(15) Read File 2 (16) Update File 1 (17) Update File 2 (18) End Tran

Figure 2.1
Transaction Locking

Figure 2.2 demonstrates how two workstations on a network can become deadlocked. The workstations are executing two different programs that access the same files in different orders within wait transactions.

Station 1 Application	Btrieve	Station 2 Application
(1) Open File 1		
(2) Open File 2		
		(3) Open File 1
		(4) Open File 2
(5) Begin Tran		
		(6) Begin Tran
(7) Read File 1		
		(8) Read File 2
(9) Read File 2		
	(10) Btrieve on Station 1 delays since Station 2 has locked File 2	
		(11) Read File 1
	(12) Btrieve on Station 2 delays since Station 1 has locked File 1	

Figure 2.2
Deadlock Example

PASSIVE CONCURRENCY

If your application performs single record read and update sequences that are not logically connected, you may choose to rely on the passive method of concurrency. Btrieve allows any workstation to read and update (or delete) records without performing any locks or transactions.

If a record changes between the time your program reads it and the time your program attempts to update (or delete) it, Btrieve returns a conflict status. This status indicates that the data has been modified by another workstation since you originally read it. In that case, your program must reread the record to perform the update (or delete) operation.

The passive method allows you to move applications directly from the single-user to the multi-user environment with only minor modifications. It also allows for a high degree of concurrency in the system.

Figure 2.3 shows how two applications interact when using the passive method of concurrency.

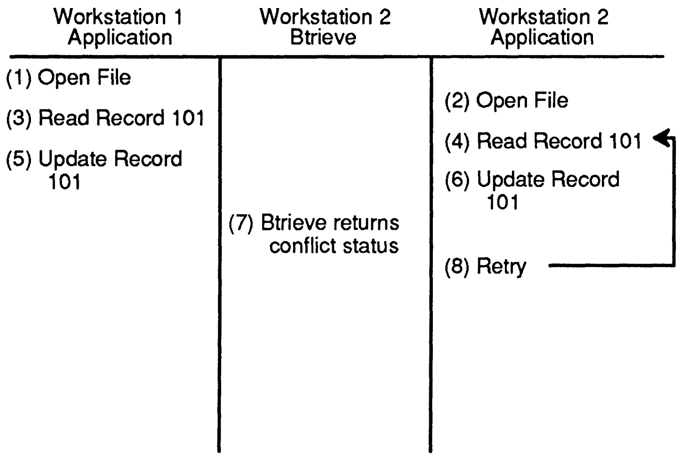


Figure 2.3
Passive Method

You can combine the passive method effectively with transaction level control. When one workstation accesses a file within a transaction, other workstations can no longer update the file or access it within their own transactions.

If such an attempt occurs, Btrieve returns a busy status or waits until the first transaction terminates, depending on how the second workstation's transaction began. Any number of workstations can gain access to read a file if the reads occur outside of a transaction.

Figure 2.4 illustrates how an application using wait transaction control can interact with an application using the passive method of concurrency. When the application on Workstation 1 reads file 1 within a transaction, Workstation 2 can still read the file outside of a transaction, but its update waits until the transaction terminates.

Since the two workstations are accessing different records in the same file, Workstation 2's update succeeds. If both had been updating the same record, Btrieve would have returned a conflict status to Workstation 2 on the update.

Workstation 1 Application	Workstation 2 Btrieve	Workstation 2 Application
(1) Open File 1		
(2) Open File 2		
		(3) Open File 1
(4) Begin Tran		
(5) Read File 1, Record 10		
		(6) Read File1, Record 95
		(7) Update File 1, Record 95
	(8) Btrieve delays since Workstation 1 has locked File 1	
(9) Read File 2, Record 10		
(10) Update File 1, Record 10		
(11) Update File 2, Record 10		
(12) End Tran		
	(13) Successful update completes	

Figure 2.4
Transaction and Passive Combination

RECORD LOCKS

Whenever a program accesses a record within a file, it can specify that it wants to lock the record. Other workstations may read the record, provided the reads are not performed with the lock option. However, no other workstation can lock, update, or delete the record until the workstation that holds the lock releases it.

When you lock a record, you use the appropriate Btrieve operation code to specify either a single record lock or a multiple record lock. A single record lock allows a workstation to have only one record locked in a file at a time; a multiple record lock, several.

Btrieve permits both wait and nowait locks. *Wait* locks do not return a status code to the application until the lock is successful. *Nowait* locks immediately return to the application with a busy status when another workstation holds the lock for the record. When you use nowait locks, the application must contain logic either to wait and retry the operation or to report the error to the user when the lock is busy.

Once a workstation has successfully locked a record with a single record lock, the lock remains in effect until one or more of the following events occur:

- The workstation updates or deletes the locked record. (The workstation can read and update other records in the file without releasing the lock).
- The workstation locks another record in the file.
- The workstation issues an unlock operation for the file.
- The workstation closes the file.
- The workstation issues a Reset operation, closing its open files.
- The workstation accesses the file within a transaction.

Once a workstation has successfully locked one or more records with multiple record locks, the locks remain in effect until one or more of the following events occur:

- The workstation deletes the locked record(s).
- The workstation explicitly unlocks the record(s).
- The workstation closes the file.
- The workstation issues a Reset operation, closing its open files.
- The workstation accesses the file within a transaction.

RESTRICTING ACCESS TO A FILE

Btrieve allows you to restrict access to a file by either assigning an owner name to the file or by opening the file in exclusive mode.

OWNER NAMES

Btrieve also allows you to restrict access to a file by specifying an owner name. Once you assign an owner name to a file, Btrieve requires that name to be specified for all future Open operations. This prevents any unauthorized access or changing of a file's contents by users or application programs that do not have access to the owner name.

You can restrict access so that users may have read-only access to the file without specifying the owner name. By assigning this selective ownership right to a file, users and application programs still cannot change the file's contents unless they specify the owner name. Btrieve allows you to remove ownership rights from a file if you know the owner name assigned to it.

When you assign an owner name, you may also request that Btrieve encrypt the data in the disk file using the owner name as the encryption key. Encrypting the data on the disk ensures that unauthorized users cannot examine your data by using a debugger or a file dump utility. Since encryption requires additional processing time, select this option only if data security is important in your environment.

EXCLUSIVE MODE

If you want to limit access to a file to a single workstation, you can specify that Btrieve open the file in exclusive mode. When a workstation opens a file in exclusive mode, no other workstation can open the file until the workstation that opened the file in exclusive mode closes it.

3 RUNNING NETWARE BTRIEVE

This chapter contains information you will need in order to install and configure NetWare Btrieve for your system.

SYSTEM REQUIREMENTS

NetWare Btrieve must be run under NetWare v2.1*x* or above. The NetWare Transaction Tracking System is required if you want to guarantee individual disk write operations, and to ensure file integrity and recovery in the event of a file server failure.

IMPORTANT:
NetWare Btrieve does not run under ELS NetWare Level I or Advanced NetWare 68 because these versions of NetWare do not support Value-Added Processes.

Btrieve requires a network file server with enough memory to load NetWare, BSERVER, and BROUTER. In most cases, the file server should have a minimum of 2MB of memory in order to run both NetWare and Btrieve efficiently.

The Btrieve Requester program (BREQUEST) requires approximately 25KB of memory at each workstation (assuming it is loaded with the default start-up options). The exact amount of memory required depends on the start-up options you specify when you load the program.

If you are using an internetwork system, workstations on sibling networks that utilize NetWare Btrieve must have access to a file server loaded with BSERVER and BROUTER.

THE BTRIEVE DISKETTES

NetWare Btrieve is shipped on two diskettes: a *PROGRAMS* diskette and a *UTILITIES* diskette. You should make copies of the Btrieve diskettes, saving the original diskettes in case the copies are damaged or lost. The original Btrieve diskettes are write-protected so that you cannot accidentally erase or replace their contents.

The *PROGRAMS* diskette contains the following files:

File	**Description**
BSERVER.VAP	The file that NetWare loads as a Value-Added Process on the file server.
BROUTER.VAP	The Btrieve Message Routing program. This program is used to provide interprocess communications between BSERVER and other VAP products, such as NetWare SQL.
BREQUEST.EXE	The Btrieve Requester program for DOS workstations This is the memory-resident program that you load a each workstation that makes Btrieve requests. BREQUEST transfers Btrieve requests from your application to the file server.
BREQUEST.DLL	The Btrieve Requester program for OS/2 workstations. This is the dynamic link library that OS/2 links to each Btrieve application. The BREQUEST routines transfer Btrieve requests from your application to the file server.
BTRCALLS.DLL	The Btrieve dynamic link routine for OS/2 workstations. This program is included to remove the need for relinking OS/2 programs specifically for NetWare Btrieve.

The *UTILITIES* diskette contains the following files:

BUTIL.EXE	The Btrieve standalone utility.
BSETUP.EXE	The Btrieve configuration and installation utility.
BSETUP.HLP	The message file for the BSETUP program.

File	**Description**
BASXBTRV.EXE	The memory resident BASIC interface used with interpretative BASIC.
BASXBTRV.OBJ	The object module containing the IBM Compiled BASIC interface to Btrieve.
QBIXBTRV.OBJ	The object module containing the Microsoft QuickBASIC interface to Btrieve.
PASXBTRV.OBJ	The object module containing the Pascal interface for the IBM (or Microsoft) Pascal compiler.
TURXBTRV.PAS	The source module containing the Pascal interface for the Turbo Pascal compiler.
BEXTERN.PAS	A Pascal source file containing the external declaration for the BTRV function.
COBXBTRV.OBJ	The object module containing the COBOL interface for the Microsoft COBOL compiler, v2.
MSCXBTRV.C	The source module for the Microsoft C interface to Btrieve.
C2XBTRV.C	The C interface source module used in OS/2 protected mode applications.
C2FXBTRV.C	The C interface source module used with OS/2 FAPI applications that run in either protected or compatibility mode.
UPPER.ALT	A file containing the definition of an alternate collating sequence that compares keys in upper and lower case as if they were all in upper case.
B.EXE	The Btrieve function executor, a testing and training program.
README.DOC	A document that describes any changes in or enhancements to Btrieve since this manual was published.
INTRFACE.DOC	A document that describes the language interfaces provided on the Btrieve diskette that are not discussed in this manual.

CONFIGURING AND INSTALLING BTRIEVE

Before your application can access NetWare Btrieve, you must first configure the Btrieve programs and then install them on a file server.

The BSERVER.VAP and BROUTER.VAP files included on the *PROGRAMS* diskette are supplied with the default settings for the initialization options. While these settings are adequate for many systems, they may not be correct for your needs. This section discusses the Btrieve configuration options and explains how to use the BSETUP.EXE program to configure and install customized versions of BSERVER.VAP and BROUTER.VAP on your file server.

THE BTRIEVE CONFIGURATION OPTIONS

Btrieve must reserve memory and resources when it loads in order to operate correctly. You can customize Btrieve for your system by specifying a set of configuration options. These include the following:

- The maximum number of open files
- The maximum number of file handles
- The maximum number of record locks
- The number of concurrent transactions
- The maximum size of the compression buffer
- The maximum record length
- The maximum page size
- The maximum number of concurrent sessions
- The console refresh delay count
- Automatic transaction flagging

The sections on the following pages discuss each option, the default values, and the memory required, if any.

MAXIMUM NUMBER OF OPEN FILES

Range: 1 - 255

Default: 20

Memory required per file: 115 bytes

The “Maximum Number of Open Files” option allows you to specify the maximum number of unique files that will be open at that file server at any given time. Btrieve uses the value you specify to determine the size of the internal tables it uses to track active files. Each unique Btrieve file on the file server requires one entry.

MAXIMUM NUMBER OF FILE HANDLES

Range: 1 - < *network limit* >

Default: 60

Memory required per handle: 114 bytes

The “Maximum Number of File Handles” option allows you to specify the maximum number of file handles that Btrieve will allow your system to use simultaneously. If two stations open the same file on the file server, they use two file handles. To calculate the maximum number of handles your system requires, multiply the number of stations by the maximum number of files each workstation can open.

MAXIMUM NUMBER OF RECORD LOCKS

Range: 0 - < *network limit* >

Default: 20

Memory required per lock: 8 bytes

The "Maximum Number of Record Locks" option sets the maximum number of records that can be locked simultaneously at that file server. Specifically, it determines the number of entries for the lock table that Btrieve builds when it loads. The value for this option includes both single and multiple record locks. To calculate the value for this option, determine the maximum number of records that each workstation can have open, and multiply that value by the number of workstations that access Btrieve files.

NUMBER OF CONCURRENT TRANSACTIONS

Range: 0 - < *maximum number of sessions* >

Default: 0

Memory required per transaction: 1,046 bytes

The "Number of Concurrent Transactions" option sets the maximum number of stations that may have concurrent active transactions at the file server. If you specify a value of zero for this option, no workstations may issue a Begin Transaction operation to the file server. If you specify a value greater than zero for this option, Btrieve creates a transaction file, BTRIEVE.TRN, in the \SYS\SYSTEM directory at the file server, and allows as many active transactions at the file server as you specify.

MAXIMUM COMPRESSED RECORD SIZE

Range: 0 - < *longest record in a compressed file* >

Default: 0

Memory required per transaction: 2 * number of kilobytes specified

The "Maximum Compressed Record Size" option specifies the size (in kilobytes) of the longest record that you will be accessing from a compressed file. Btrieve will allocate twice the number of kilobytes you specify to its compression buffer. Specifying a higher value than you need does not result in improved performance, and may result in a reduction of the memory available for other processes at the file server.

If you use compressed files, set the value for this option to the size of the longest record in any of your compressed files. Specify the value in kilobytes. Round any uneven values up to the next higher kilobyte value. For example, if the longest record you will be accessing is 1,800 bytes long, specify a value of 2 for this option.

If you do not use compressed files, set the value to 0.

MAXIMUM RECORD LENGTH

Range: 4 bytes - 32KB

Default: 8,192 bytes

Memory required: (specified value + 269 bytes)

The "Maximum Record Length Allowed" option specifies the length of the longest record that any Btrieve application can access at that file server. Always specify the length of the record in bytes. Specifying a higher value than you need does not result in improved performance.

MAXIMUM PAGE SIZE

Range: 512 - 4,096 bytes

Default: 4,096 bytes

Memory required: n/a

The "Maximum Page Size Allowed" option allows Btrieve to calculate the size of the cache buffers it needs. The value you specify here should be the maximum page size of any Btrieve file you want to access. It must be a multiple of 512 bytes, but no greater than 4,096 bytes.

MAXIMUM NUMBER OF CONCURRENT SESSIONS

Range: 1 - < *number of workstation tasks* >

Default: 15

Memory required per session: 1,296 bytes

The "Maximum Number of Concurrent Sessions" option specifies the maximum number of workstation tasks that can access BSERVER at any given time. A session is defined as one copy of BREQUEST communicating with the BSERVER program. Each session is allocated two packet buffers for Btrieve requests.

Specifying a higher value than you need does not result in improved performance.

CONSOLE REFRESH DELAY COUNT

Range: 1 - 60 seconds

Default: 3 seconds

Memory required: n/a

The "Console Refresh Delay Count" option controls the number of seconds that B STATUS and B ACTIVE delay before refreshing the screen with new information. The delay allows you to maintain the readability of the information while observing the effect of database activity in a real time mode.

AUTOMATIC TRANSACTION FLAGGING

Range: Yes/No

Default: No

Memory required: n/a

The "Automatic Transaction Flagging" option controls whether or not Btrieve automatically flags files as transactional at the time you create them on the system. If you answer "Yes" to this option, Btrieve flags newly created files as transactional. If you answer "No," Btrieve does not flag the files as transactional.

INSTALLATION OPTIONS

The BSETUP program is designed to give you the following installation options:

- You can install BSERVER and BROUTER directly on a file server. If this is the case, you must be logged in as SUPERVISOR or have supervisor rights. If you are not logged in as SUPERVISOR, the "Install" and "Remove" options will not be displayed on the menu.
- You can copy the BSERVER.VAP and BROUTER.VAP files to a subdirectory or a diskette, run BSETUP to configure the copy, and then either install the copy on the current file server or transport the file to another file server.

STARTING BSETUP

To start BSETUP, complete the following steps.

1) Start the personal computer that you will use to run BSETUP.

2) Copy BSERVER.VAP and BROUTER.VAP to a subdirectory or diskette, depending on which installation option you are using.

3) Make sure that the current directory is the subdirectory to which you copied the NetWare Btrieve programs.

4) At the DOS prompt, type the BSETUP command as follows:

```
BSETUP <Enter>
```

USING BSETUP

When BSETUP loads, a menu similar to the following will appear:

```
        Available Options
    Change File Server
    Install Btrieve
    Remove Btrieve
    Save Configuration
    Set Configuration
```

The following sections discuss each option and give information about using them to configure and install Btrieve. The options are presented in the order in which you would most likely perform them. If you need more detailed help about the task you are currently performing, press the F1 (Help) key.

Use the Up- and Down-arrow keys to highlight menu options. If at any time you want to exit a menu option, press <Esc>.

To exit BSETUP, press <Esc> at the "Available Options" menu, and highlight "Yes" on the "Exit BSETUP" menu.

CHANGE FILE SERVER

The "Change File Server" option allows you to choose the file server on which you want to configure, install, remove, or save NetWare Btrieve.

To change file servers, complete the following steps.

1) Use the Up- or Down-arrow key to highlight the "Change File Server" option and press <Enter>.

   A menu will appear, listing names of the file servers that are currently attached to the workstation.

2) Use the Up- or Down-arrow key to highlight the name of the file server on which you want to perform the BSETUP operation and press <Enter>.

   There will be a short wait, and then the "Available Options" menu will reappear.

   If you have supervisor rights on the file server that you select, all of the menu options will appear. You can now configure, install, remove, or save NetWare Btrieve at that file server.

   If you do not have supervisor rights on the file server that you select, the "Install NetWare Btrieve" and "Remove NetWare Btrieve" options will not appear. You can only configure and save the NetWare Btrieve configuration at a file server on which you do not have supervisor rights.

SET CONFIGURATION

The "Set Configuration" option allows you to define the Btrieve options for the copy of Btrieve in the current directory.

IMPORTANT:
If BSERVER.VAP and BROUTER.VAP are not present in the current directory when you attempt to set the initialization options, BSETUP returns an error message and terminates execution.

To set one or more of the initialization options, complete the following steps.

1) Highlight "Set Configuration" in the "Available Options" menu and press <Enter>.

   A screen similar to the following will appear:

```
           Number of open files:20
             Number of handles: 60
                Number of locks: 20
          Number of transactions: 0
 Largest compressed record size: 0
         Largest record size: 8192
          Largest page size: 4096
          Number of sessions: 15
          Console refresh count: 3
  Create files as transactional: Yes
```

   The values displayed in the fields to the right of the colon are the values that were last defined for the options. If you are configuring a new copy of NetWare Btrieve, the default values appear in the fields.

2) Highlight the field you want to define.

3) Type in the new value and press <Enter>.

   If you press <Esc>, the previously defined option will remain in effect.

   If you enter an invalid value for an option, BSETUP will warn you via an error message. At this point, you can delete the incorrect value and enter a valid value.

4) Continue to highlight fields and enter new values for all the options you want to define.

When you have entered the needed values, press <Esc> to return to the "Available Options" menu. You must execute the "Save Configuration" option before the new values you have specified will be saved to the NetWare Btrieve programs in the current directory.

SAVE CONFIGURATION

The "Save Configuration" option allows you to save the Btrieve configuration to the copies of BSERVER.VAP and BROUTER.VAP in the current directory. You do not have to be logged in as SUPERVISOR to save a new Btrieve configuration.

To save the new Btrieve configuration, complete the following steps.

1) Highlight "Save Configuration" and press <Enter>.

   The "Update Btrieve" prompt will appear.

2) Highlight "Yes" and press <Enter>.

   When you save a new configuration, BSETUP adjusts the values for the Btrieve options in the copies of BSERVER.VAP and BROUTER.VAP stored in the current directory.

   BSETUP will return to the "Available Options" menu.

If you want the new configuration to take effect on the current file server, execute the "Install Btrieve" option on the "Available Options" menu, and then reboot the network.

INSTALL BTRIEVE

The "Install Btrieve" option allows you to install Btrieve on the file server to which you are currently logged in. You must be logged in as SUPERVISOR or have supervisor rights to install Btrieve on a file server.

To install Btrieve on a file server complete the following steps.

1) Highlight the "Install Btrieve" option on the "Available Options" menu and press <Enter>.

   If NetWare Btrieve is not installed on the current file server, an entry box will appear, prompting you to enter a password for Btrieve.

   If NetWare Btrieve is already installed on the current file server, the "Replace Btrieve" prompt will appear, asking whether you want to replace the currently installed programs. If this is the case, complete the following steps.

   a) To abandon the installation and return to the "Available Options" menu, highlight "No" and press <Enter>, or press <Escape>.

   b) To replace the existing configuration, highlight "Yes" and press <Enter>.

      If you answer "Yes" to the "Replace Btrieve" prompt, an entry box will appear, prompting you to enter the Btrieve password.

3) NetWare Btrieve requires a password to identify itself to NetWare. This prevents an unauthorized program from gaining access to NetWare using Btrieve's object name.

   To abandon the installation at this point and return to the "Available Options" menu, press <Escape>.

   To continue the installation, type a password up to eight characters long for Btrieve and press <Enter>. If you do not want to specify a password, but you want to continue the installation, press <Enter> at the password prompt. The "Install BROUTER" prompt will appear.

IMPORTANT:
The password prompt is the last step of the installation process at which you can completely abandon the installation without installing any NetWare Btrieve programs. If you proceed any farther in the process, you must install at least the BSERVER program.

4) The "Install BROUTER" prompt gives you the option of installing the BROUTER program at the same time as BSERVER.

   To install both BSERVER and BROUTER, highlight "Yes" and press <Enter>. BSETUP will

   - Copy BSERVER.VAP and BROUTER.VAP to the SYS:SYSTEM directory on the preferred file server;
   - Define the NetWare Btrieve programs as objects to NetWare and assign BSERVER the password you specified. (BROUTER does not require a password.)

   To install only BSERVER, highlight "No" and press <Enter>. BSETUP will

   - Copy BSERVER.VAP to the SYS:SYSTEM directory on the preferred file server;
   - Define BSERVER as an object to NetWare and assign it the password you specified.

   Do not install BROUTER if you are not using another VAP that makes Btrieve calls (such as NetWare SQL).

5) After you have installed the new NetWare Btrieve programs on the file server, you must restart the file server in order for the new configuration options to take effect.

REMOVE BTRIEVE

The "Remove Btrieve" option removes the Btrieve programs from the file server to which you are currently logged in. Once you have executed this option, the Btrieve VAPs will no longer load at that file server. You must be logged in as SUPERVISOR or possess supervisor rights to remove Btrieve from a file server.

To remove Btrieve from a file server, complete the following steps.

1) Highlight the "Remove Btrieve" option and press <Enter>.

   The "Remove Btrieve" prompt will appear.

2) Highlight "Yes" and press <Enter>.

   When you select this option, BSETUP will

   - Delete BSERVER.VAP (and BROUTER.VAP, if it is installed) from the SYS:SYSTEM directory on the preferred file server;
   - Remove the Btrieve object name and password from the network.

IMPORTANT:
If NetWare SQL is defined as an object in the NetWare bindery, BSETUP will not remove the NetWare Btrieve program files. You must remove the NetWare SQL program files from the file server before you can remove the NetWare Btrieve files. Refer to Chapter 3 of the *NetWare SQL User's Manual* for more information.

STOPPING THE BTRIEVE VAPS

Once BSERVER and BROUTER are activated, they remain resident in the file server's memory the entire time the NetWare operating system is running. You cannot remove them from the file server's memory without first removing them from the file server using the BSETUP utility (see "Remove Btrieve" on page 3-16) and then restarting the operating system.

When you issue a DOWN command at a file server, BSERVER performs a reset for all active network connections that are accessing files on the file server.

THE BREQUEST PROGRAM

NetWare Btrieve provides BREQUEST programs for both DOS and OS/2 workstations. This section describes the BREQUEST start-up options and provides instructions for running BREQUEST in both environments.

BREQUEST START-UP OPTIONS

This section describes the BREQUEST start-up options and the values you can assign to them.

[*/R: mapped drives*]

The /R option denotes the maximum number of mapped drives the workstation can access. When you omit this option, BREQUEST uses a default value of three. Each drive you specify increases the memory resident size of BREQUEST by 20 bytes. For example, if the workstation has five mapped drives, specify the /R option as follows:

```
    /R:5
```

[*/D: data message length*]

The /D option specifies the length of the longest record you will be accessing through Btrieve. BREQUEST uses the value you specify here to calculate the length of the data message buffer that it reserves for passing records between BSERVER and your application. The value you enter here should be the same as the maximum record length you configure for Btrieve through the BSETUP program. See "Maximum Record Length" on page 3-7.

The default value for the /D option is 2,048 bytes. The maximum record length you can specify is 32K bytes. Specifying a higher value than you need for the /D option does *not* result in improved performance.

BREQUEST maintains two copies of the data message buffer. The /D option increases the memory resident size of BREQUEST by twice the number of bytes you specify plus 538 bytes.

Always specify the record length in bytes. For example, if the longest record your application uses is 3,000 bytes long, specify the /D option as follows:

```
/D:3000
```

[*/S: number of file servers*]

The /S option specifies the number of file servers to which the workstation will be mapped. The default value for the /S option is one. The maximum value you can specify is eight. For example, if the workstation has drives mapped to three file servers, specify the /S option as follows:

```
/S:3
```

BREQUEST FOR DOS WORKSTATIONS

You must start the BREQUEST program at a workstation before the workstation can access network Btrieve files through the BSERVER VAP. If you want to access local files at the workstation, you must load a copy of the Btrieve Record Manager (either Single User or DOS 3.1 Network) before you load BREQUEST.

Start BREQUEST at the workstation by issuing the following command:

```
<Drive> BREQUEST [/R: number of mapped drives]
                 [/D: data message length]
                 [/S: number of file servers]
```

Replace *<Drive>* with the name of the drive on which BREQUEST is stored.You can omit the drive name if BREQUEST is stored on the default drive, or if it is located in a directory in your search path.

The BREQUEST start-up options are described beginning on page 3-17.

For example, to specify 4 mapped drives, a 2,048 byte data message length, and 2 file servers, use the following command:

```
BREQUEST /R:4 /D:2048 /S:2
```

To ensure that the start-up options are always loaded, place the BREQUEST command in the workstation's AUTOEXEC.BAT file.

BREQUEST FOR OS/2 WORKSTATIONS

BREQUEST.DLL and BTRCALLS.DLL, the Btrieve dynamic link routines, must be installed at a workstation before the workstation can access network Btrieve files through the BSERVER VAP. When the first Btrieve application is started, OS/2 loads automatically.

INSTALLING BREQUEST

To install BREQUEST for OS/2, copy the BREQUEST.DLL and the BTRCALLS.DLL files from the diskette to either of the following:

- One of the directories specified in the LIBPATH command in the CONFIG.SYS file
- The root directory of the OS/2 boot drive

Refer to your OS/2 manual for more information on LIBPATH and on identifying the locations for the dynamic link libraries.

IMPORTANT:
The BREQUEST.DLL and BTRCALLS.DLL dynamic link routines supplied with NetWare Btrieve allow access only to remote files. An application at an OS/2 workstation cannot access local files using these routines.

INITIALIZING BREQUEST

You can specify initialization options that are specific to each Btrieve application running at the workstation. BREQUEST uses an OS/2 environment variable, REQPARMS, to define the initialization options an application needs to use. The BREQUEST initialization options are described under "BREQUEST Start-up Options" beginning on page 3-17.

Set the BREQUEST initialization options using the following environment command:

```
SET REQPARMS=[/R:  number of mapped drives]
             [/D:  data message length]
             [/S:  number of file servers]
```

Do not include a space between the REQPARMS variable name and the equal sign. You can, however, insert a space between each initialization option you specify.

For example, to specify 4 mapped drives, a 2,048 byte data message length, and 2 file servers, use the following command:

```
SET REQPARMS=/R:4/D:2048/S:2
```

To ensure that the initialization options are always loaded, place the SET REQPARMS command in one of the special initialization batch files used by OS/2.

STOPPING BREQUEST

At a DOS workstation, there are two methods you can use to remove BREQUEST from memory:

- Your application can issue a Stop operation (Btrieve operation 25).
- You can issue the BUTIL –STOP command from the workstation's command line.

At an OS/2 workstation, the operating system removes the dynamic link routines from memory when the last Btrieve application is terminated. You cannot remove the dynamic link routines from memory while a Btrieve application is active because the operating system dynamically links them with the application when the application is loaded.

4 UTILITIES

NetWare Btrieve provides you with a complete set of utilities to help you with file creation, file maintenance, testing, and debugging. In addition, NetWare Btrieve includes several command line utilities that allow you to monitor and manage NetWare Btrieve activity on your network.

The three NetWare Btrieve utilities are

- **BUTIL.EXE**, a program containing commands that allow you to create and manage Btrieve data files;
- **B.EXE (The Btrieve Function Executor)**, an interactive utility that you can use for instructional purposes and for testing and debugging your application program logic;
- **Command Line Utilities**, utilities that allow you to monitor and manage NetWare Btrieve activity on your network.

THE BUTIL PROGRAM

The BUTIL program contains a complete set of commands for use in file creation, maintenance, and recovery. The following sections describe how to run BUTIL, check for BUTIL error messages, create BUTIL description files and alternate collating sequence files. The BUTIL commands are presented in alphabetical order under the heading “BUTIL Commands”, beginning on page 4-16.

RUNNING BUTIL

To run BUTIL, complete the following steps.

1) Start the BREQUEST program before running BUTIL. If you need help with this procedure, see "The BREQUEST Program" beginning on page 3-17.

2) Enter the BUTIL command in the following format:

```
<Drive>:BUTIL –COMMAND [Parameters] [–O<Owner>]
```

Replace *<Drive>* with the name of the drive or other device that contains the Btrieve program files. You can omit the drive if you intend to use the default disk drive.

Replace *–COMMAND* with the name of the Btrieve command (COPY, LOAD, etc.) that you want to use. You must preface the command with a dash (–).

Replace [*Parameters*] with the information BUTIL needs to perform the command you choose. The parameters are described under the heading for the corresponding command.

Replace *<Owner>* with either the owner name for the Btrieve file the command will access or with an asterisk (*). If you use an asterisk in place of the owner name, BUTIL will prompt you for the owner name of the file. BUTIL requires the –O*<Owner>* parameter if any of the Btrieve files specified in the command requires an owner name.

BUTIL discards leading blanks unless an asterisk (*) is the first nonspace character of the –O parameter.

If you specify two Btrieve files in the command, you must specify the –O parameter twice, once for each file. If only the second file has an owner name, BUTIL ignores the first owner name. You can use the asterisk option twice.

If you enter the BUTIL command with no parameters, BUTIL will list the correct command format for all of the commands. You can redirect this output to a file or printer using the DOS redirection feature.

BUTIL ERROR MESSAGES

BUTIL returns error messages in two ways, depending on whether you run it from the command line or in a batch file.

If any errors occur while you are running BUTIL from the command line, a message will appear on the screen describing the problem. Refer to Appendix B for information about BUTIL and Btrieve error messages.

If any errors occur while you are running BUTIL in a batch file, BUTIL will return a DOS error level. The following table lists and describes the BUTIL error levels.

Error Level	Description
0	The utility ran to completion with no errors.
1	The utility ran to completion, but a nonfatal error, such as a status code 5 (Duplicate Key Value), occurred.
2	The utility did not run to completion because a fatal error, such as a status code 2 (I/O Error), occurred.

You can check for the DOS error level by including a condition similar to the following in the batch file after the BUTIL command:

```
IF ERRORLEVEL n ECHO message
```

Replace *n* with the number of the BUTIL error level. Replace *message* with a meaningful message.

Because of the way DOS batch commands test for error levels, you should always test for an error level 2 first. The example code from a batch file on the next page illustrates one way to handle BUTIL error levels.

```
        .
        .
        .
BUTIL -LOAD LOADFILE BTRFILE -OOWNRNAME
IF ERRORLEVEL 2 ECHO Fatal Error: BUTIL operation incomplete.
IF ERRORLEVEL 1 ECHO Nonfatal Error: Check for duplicates in load file.
IF ERRORLEVEL 0 ECHO Operation completed successfully.
        .
        .
        .
```

Refer to the discussion of batch files in your operating system's reference manual for more information about how to use the ERRORLEVEL condition.

BUTIL DESCRIPTION FILES

Several of the BUTIL commands, including CREATE, SAVE, and SINDEX, use a *description file.* A description file is a sequential ASCII file that contains certain information that Btrieve needs to perform its operations.

The next section describes the elements used in BUTIL description files. A section containing a set of rules you should follow when you create a description file follows the element descriptions.

DESCRIPTION FILE ELEMENTS

Description files consist of a list of *elements.* An element consists of a keyword, followed by an equal sign (=), followed by a value. Each element in the description file corresponds to a particular characteristic of a Btrieve file or key definition. For a complete description of the Btrieve file and key characteristics, see Chapter 2, "Btrieve File Management."

The following pages contain explanations of the description file elements. Some description file elements are marked "optional", and may be omitted from the description file if they are not needed for the file or key definition.

The elements are presented in the order in which they must appear in the description file. Under each element heading you will find the following subheadings:

- **Element**–presents the correct format for the element keyword. The value is presented as a variable. Variables representing numeric values are shown by the characters *nnnn*. Other values are explained in the text that accompanies each element description.
- **Range**–defines the range of acceptable values you can specify for the element.
- **Commands**–lists the BUTIL commands that require the element in their description files.

The subheadings are followed by a brief description of the element and how you use it.

NOTE:

The values in the element descriptions are shown here enclosed in quotation marks. Do not enclose the values in your description file in quotation marks.

Record Length

Element : record=< *nnnn* >

Range: 4 - 4090

Commands: CREATE

The "Record Length" element defines the logical data record length for the file. For fixed-length records, this value should correspond to the length of the data buffer parameter that performs operations on the file. If you are using variable length records, the record length you specify here should correspond to the fixed length portion of the record.

The minimum data record length must be large enough to contain all the keys defined for the file, but not less than four bytes. The record length, plus its key overhead, plus six bytes must not exceed the file's page size.

Variable Length Records

Element : variable=< *y* | *n* >

Range: n/a

Commands: CREATE

The "Variable Length Records" element specifies whether you want the file to contain variable length records.

Specify "y" if you want the file you are creating to allow variable length records. Otherwise, specify "n".

Blank Truncation

Element : truncation=< *y* | *n* >

Range: n/a

Commands: CREATE

The "Blank Truncation" element specifies whether you want Btrieve to perform blank truncation on variable length records. The truncation key word is only required if you specify "y" for the variable length records element.

Specify "y" if you want Btrieve to use blank truncation. Otherwise, specify "n".

Data Compression

Element : compress=< *y* | *n* >

Range: n/a

Commands: CREATE

The "Data Compression" element specifies whether you want Btrieve to perform data compression on records that are inserted into the file.

Specify "y" if you want Btrieve to perform data compression. Otherwise, specify "n".

Key Count

Element : key=< *nn* >

Range: 1 - 24

Commands: CREATE

The "Key Count" element specifies the number of keys to be defined in the file. If you specify a value of 0 for this element, Btrieve will create a data-only file. Otherwise, Btrieve will create either a standard Btrieve file or a key-only file, depending on the value you specify for the "Include Data" element.

For values greater than 0, you must define at least 1 key but no more than 8 if the page size is 512 bytes. If the page size is 1024 bytes or more, you can define up to 24 keys.

Page Size

Element : page=< *nnnn* >

Range: 512 - 4096

Commands: CREATE

The "Page Size" element specifies the physical page size in bytes for the file. You can specify any multiple of 512, up to 4096.

Page Pre-allocation

Element : allocation=< *nnnnn* >

Range: 1 - 64K

Commands: CREATE

The "Page Pre-allocation" element specifies the number of pages you want to pre-allocate to the file. If you don't want to pre-allocate any pages, don't include this keyword in your description file.

Replace Existing File (Optional)

Element : replace =< *y* | *n* >

Range: n/a

Commands: CREATE

The "Replace Existing File" element indicates that you do not want Btrieve to create the new file if a file of the same name already exists, and to warn you of the file's existence. Specify "n" if you do not want to create a new file over an existing file.

If you want to replace an existing Btrieve file with a new, empty file of the same name, either specify "replace=y", or do not include this element in the description file.

Include Data (Optional)

Element : data=< *y* | *n* >

Range: n/a

Commands: CREATE

The "Include Data" element specifies whether you want Btrieve to create a key-only file. Specify "n" if you want Btrieve to create a key-only file.

To create a standard file, either specify "y" for the "Include Data" element, or omit the element from the description file.

To create a data-only file, specify "y" for the "Include Data" element and set the "Key Count" element to 0.

Free Space Threshold

Element : fthreshold=< *10* | *20* | *30* >

Range: 10%, 20%, or 30% of the page size

Commands: CREATE

The "Free Space Threshold" element specifies the amount of free space you want Btrieve to reserve on a file page for variable length record expansion. The value you specify is expressed as a percentage of the page size.

Do not include the "Free Space Threshold" element in the description file if the file does not allow variable length records. If you specify "variable=y" and you do not specify a free space threshold, Btrieve will use a default value of 5% of the page size.

NOTE:

The following elements define the key information for the file. You must enter the key information, beginning with key position, for each key segment you want to define.

Key Position

Element : position=< *nnn* >

Range: 1 - < *record length* >

Commands: CREATE, INDEX, SINDEX

The "Key Position" element indicates the position of the key segment in the record. The key position must be at least 1 and cannot be larger than the value you specified for the record length. The keys you define may overlap.

Key Length

Element : length=< *nnn* >

Range: 1 - < *limit for key type* >

Commands: CREATE, INDEX, SINDEX

The "Key Length" element defines the length of the key or key segment field. The total of the key's length and starting position cannot exceed the file's defined record length. The key length must be an even number if the key is a binary key type.

Duplicate Key Values

Element : duplicates=< *y* | *n* >

Range: n/a

Commands: CREATE, INDEX, SINDEX

The "Duplicate Key Values" element specifies whether you want to allow more than one record in the file to contain the same value for this key field.

Specify "y" if you want to allow duplicate key values for the key. Otherwise, specify "n".

Modifiable Key Values

Element : modifiable=< *y* | *n* >

Range: n/a

Commands: CREATE, INDEX, SINDEX

The "Modifiable Key Values" element specifies whether you want to allow an application to modify the key value during an Update operation.

Specify "y" if you want the values for this key to be modifiable. Otherwise, specify "n" .

Key Type

Element : type=< *valid key type* >

Range: Any of the Btrieve key types

Commands: CREATE, INDEX, SINDEX

The "Key Type" element specifies the data type for the key. You can specify the entire word (as in "float") or just the first two letters of the word ("fl" for the float type). Refer to Appendix G for more information about key types.

Descending Sort Order (Optional)

Element : descending=y

Range: n/a

Commands: CREATE, INDEX, SINDEX

The "Descending Sort Order" element specifies whether you want Btrieve to collate the index in descending order.

Include the "Descending Sort Order" element and specify "y" if you want Btrieve to collate the key values in descending order. If you don't include this element, Btrieve will collate the key values in ascending order.

Alternate Collating Sequence

Element : alternate=< *y* | *n* >

Range: n/a

Commands: CREATE, INDEX, SINDEX

The "Alternate Collating Sequence" element specifies whether you want to sort the data by a collating sequence other than the standard ASCII sequence. This is useful if you want to use a non-English alphabet or if you want to map lower-case characters to upper-case.

You may only specify an alternate collating sequence for string, lstring, or zstring key types. If you want Btrieve to sort an index by an alternate collating sequence, enter "y" in this field. Otherwise, specify "n".

Manual Key (Optional)

Element : manual=< *y* | *n* >

Range: n/a

Commands: CREATE, INDEX, SINDEX

The "Manual Key" element specifies that the key (or key segment) you are defining is manual. If you define a key segment as manual, you must define a null value for that segment. If the key is a segmented key, and you define one segment as manual, you must define each segment as manual.

Null Key

Element : null=< *y* I *n* >

Range: n/a

Commands: CREATE, INDEX, SINDEX

The "Null Key" element specifies whether the key you are defining should have a null value. If you define a null value for one segment of a segmented key, you must define a null value for every segment of that segmented key. The null values you define for each segment can be different.

You can include the "Null Key" element in a description file for the INDEX command. However, INDEX disregards any null value you specify. BUTIL allows this in order to maintain consistent formats for the CREATE, INDEX, and SINDEX description files.

Specify "y " if you want to define a null value for this key. Otherwise, specify "n".

Null Key Value

Element : value=<*nnn*>

Range: any 1-byte hexadecimal value

Commands: CREATE, INDEX, SINDEX

The "Null Key Value" element specifies in hexadecimal form the value you want Btrieve to recognize as the null character for the key. Typical null values are 20 for blank and 0 for binary zero. Include this element only if you defined the key as allowing null values. If you specify "n" for the "Null Key" element, do not include the "Null Key Value" element in the description file.

Segmented Key

Element : segment=<*y* | *n*>
Range: n/a
Commands: CREATE, INDEX, SINDEX

The "Segmented Key" element specifies whether the key you are defining has any more segments.

Specify "y" if the key has another segment. Specify "n" if you are defining a nonsegmented key or the last segment of a segmented key.

Alternate Collating Sequence Filename

Element : name=<*filename*>
Range: valid filename
Commands: CREATE, INDEX, SINDEX

The "Alternate Collating Sequence Filename" element specifies the name of the file that contains the alternate collating sequence for the file you are creating. You can include any number of directory levels in the filename.

If you specified "n" for the "Alternate Collating Sequence" element, do not include this keyword in your description file.

RULES FOR DESCRIPTION FILES

The following rules apply to BUTIL description files. If BUTIL returns an error while trying to access the description file, check for problems in the following areas.

- All elements, such as "type=fl", must be in lower-case characters.
- All elements must be spelled correctly and separated from adjoining elements by "white space" (i.e., blank, tab, CR/LF, etc.)
- The elements must be presented in the description file in the order in which they are presented in the previous section.
- Responses must be consistent. For example, if you specify "null=y" for the "Null Key" element, the "Null Key Value" element and its value must appear; otherwise it must not. If you specify "alternate=y" for the "Alternate Collating Sequence" element in one or more key segments, the "Alternate Collating Sequence Filename" element and the full or relative path name of the alternate collating sequence file must be present as the last element in the description file.
- Ensure that there are enough key descriptions to form the number of keys given in the "Key Count" element. Enter the key information, beginning with key position, for each key segment in the file.
- Ensure that the description file contains no text formatting characters. (Some word processors embed formatting characters in a text file. The description file must not contain any formatting characters.)
- BUTIL does not check for the end of the description file. If you haven't specified an alternate collating sequence, it is possible to include too many key descriptions without receiving an error message. You can include comments at the end of description file after all the key descriptions and the "Alternate Collating Sequence Filename" element.
- Note that the description files for CREATE, INDEX, and SINDEX have slightly different formats.

ALTERNATE COLLATING SEQUENCE FILES

The first 265 bytes of an alternate collating sequence file contain the definition of a collating sequence other than the standard ASCII sequence. If you want to create an alternate collating sequence file, you should write an application that generates a file in the format specified in the table below.

Offset	Length	Description
0	1	Signature byte. This byte should always contain the hexadecimal value AC.
1	8	An 8-byte name that uniquely identifies the alternate collating sequence to Btrieve.
9	256	A 256-byte table containing the sort value for every character. Store the value for each sort character at the offset corresponding to the character's representation in the ASCII collating sequence. For example, to sort the character A as something other than 0x41, store the new sort value at offset 0x41 in the table.

For example, if you wanted to insert a character with a hex value of 5D between the letters U (hex 55) and V (hex 56) in your sequence, byte 5D in the sequence would contain the value 56 and bytes 56-5C in the sequence would contain the values 57-5D.

The UPPER.ALT file, which you will find on your Btrieve program diskette, contains an example of an alternate collating sequence. That particular sequence compares upper- and lower-case characters as if they were all upper-case. If you have multiple files with different alternate collating sequences, each sequence should have a different name.

BUTIL COMMANDS

The following sections describe how to use each of the following BUTIL commands:

Command	**Function**
CLONE	Creates an empty Btrieve file with the same file specifications as an existing file
COPY	Copies the contents of one Btrieve file to another Btrieve file
CREATE	Creates a Btrieve file
DROP	Drops a supplemental index
EXTEND	Creates a partitioned file
INDEX	Creates an external index file
LOAD	Loads the contents of a standard sequential file into a Btrieve file
RECOVER	Recovers data from a damaged Btrieve file
RESET	Closes data files and releases resources
SAVE	Saves the contents of a Btrieve file into a standard sequential file
SINDEX	Creates a supplemental index
STAT	Reports statistics about file attributes and current sizes
STOP	Terminates BREQUEST and the local Record Manager (if loaded) and removes them from memory
VER	Retrieves the Btrieve version and revision numbers

CLONE

Command Format

```
BUTIL –CLONE < Existing File >< New File >[ –O < Owner >]
```

Description

The CLONE command creates a new, empty Btrieve file with the same file specifications, including any supplemental indexes, as an existing file. You can use CLONE when you want to replicate an existing file, but you don't want to destroy the information contained in the existing file, as would happen if you used CREATE. In addition, CLONE does not require a description file in order to create a new file.

How To Use Clone

To run COPY, enter the command in the format shown above.

For *<Existing File>*, substitute the name of the Btrieve file that you want to replicate. You can specify a full path name if necessary.

For *<New File>*, substitute the name you want to use for the new, empty Btrieve file. You can specify a full path name if necessary.

For *<Owner>*, substitute the owner name of the existing file, if one is required. The new file will retain the existing file's owner name.

Example

The following command clones the NEWINV.DAT file from the INVOICE.DAT file. The owner name for the INVOICE.DAT file is "Sandy."

```
BUTIL –CLONE INVOICE.DAT NEWINV.DAT –O Sandy
```

COPY

Command Format

```
BUTIL –COPY < Input File >< Output File >[ –O < Owner1 >[ –O < Owner2 >]]
```

Description

The COPY command copies the contents of one Btrieve file to another. A common use of COPY is to change the defined key characteristics (such as key position, key length, or duplicate key values) for a Btrieve file.

COPY retrieves each record in the input file and inserts it into the output file using Btrieve Get and Insert operations. COPY performs in a single step the same function as SAVE followed by LOAD.

How To Use Copy

To run COPY, enter the command in the format shown above.

Replace *<Input File>* with the name of the Btrieve file from which you are transferring the records. You can specify a full path name if necessary.

Replace *<Output File>* with the name of the Btrieve file into which you want to insert the records. The file may or may not be empty. You can specify a full path name if necessary.

Replace *<Owner1>* and *<Owner2>* with the owner names for the Btrieve files, if required. If the *<Input File>* requires an owner name, you may specify the owner for *<Owner1>* or use the asterisk option described on page 4-2. If the *<Output File>* requires an owner name, use both the *<Owner1>* and *<Owner2>* options. If the input file does not require an owner name, you can leave *<Owner1>* blank. Use *<Owner2>* to specify the owner name for the *<Output File>*.

After the records have been copied from the input file to the output file, COPY will display on the screen the total number of records copied.

COPY
(Continued)

Example

The following command copies the records in the CUSTOMER.DAT file to the ACCTRECV.DAT file. The CUSTOMER.DAT file does not require an owner name. The ACCTRECV.DAT file has the owner name “Pam.”

```
BUTIL –COPY CUSTOMER.DAT ACCTRECV.DAT –O –O Pam
```

CREATE

Command Format

```
BUTIL -CREATE < New Filename > < Description File >
```

Description

The CREATE command generates an empty Btrieve file using the characteristics you specify in a description file.

How To Use Create

Before you can run CREATE, you must first generate a description file with a text editor. Description files are described under "BUTIL Description Files," beginning on page 4-4.

To run CREATE, enter the command in the format shown above.

Replace *<New Filename>* with the name of the file you want to create. You can specify a full path name if necessary.

NOTE:
If the name you specify for *<New Filename>* is the name of an existing Btrieve file, BUTIL will create a new, empty file in place of the existing file. Any data that was stored in the existing file will be lost, and cannot be recovered.

For *<Description File>*, substitute the name of the description file containing the specifications for the new file. You can specify a full path name if necessary.

CREATE
(Continued)

Sample Description File For BUTIL -CREATE

The description file illustrated in Figure 3.1 creates a Btrieve file with a page size of 512 bytes and two keys. The fixed length portion of the record is 98 bytes long. The file allows variable length records but does not use blank truncation. The file uses data compression. The free space threshold is set to 20%. Btrieve will pre-allocate 100 pages when it creates the file.

The first key (Key 0) is a segmented key with two duplicatable, non-modifiable, string segments with an alternate collating sequence defined in the UPPER.ALT file, and a null value of space. The second key (Key 1) is a numeric, non-segmented key that does not allow duplicates but does permit modification. It is sorted in descending order.

File specifications	record=98 variable=y truncation=n
	compress=y key=2 page=512
	allocation=100 replace=n fthreshold=20

Key 0, segment 1	position=1 length=5 duplicates=y
	modifiable=n type=string alternate=y
	null=y value=20 segment=y

Key 0, segment 2	position=6 length=10 duplicates=y
	modifiable=n type=string alternate=y
	null=y value=20 segment=n

Key 1	position=16 length=2 duplicates=n
	modifiable=y type=numeric
	descending=y alternate=n null=n
	segment=n

Alternate collating sequence filename	name=UPPER.ALT

Figure 3.1
Sample Description File for CREATE

CREATE
(Continued)

Example

The following command creates a Btrieve file named ACCTS.NEW using the description provided in the BUILD.DES description file.

```
BUTIL –CREATE ACCTS.NEW BUILD.DES
```

DROP

Command Format

```
BUTIL –DROP < Btrieve File > < Key Number > [ –O < Owner >]
```

Description

You can use the BUTIL –DROP command to remove a supplemental index from a Btrieve file.

Btrieve adjusts the key number of other supplemental indexes, if necessary, upon completion of the DROP command.

How To Use DROP

To run DROP, enter the command in the format shown above.

Replace *<Btrieve File>* with the name of the Btrieve file from which you are dropping the index. You can specify a full path name if necessary.

Replace *<Key Number>* with the number of the supplemental index you want to drop.

Replace *<Owner>* with the owner name of the file, if there is one.

Example

The following example drops key number 6, a supplemental index, from the MAILER.ADR file. The owner name of the file is "Sales."

```
BUTIL –DROP MAILER.ADR 6 –O Sales
```

EXTEND

Command Format

```
BUTIL –EXTEND < Btrieve File > < Extension File > [ –O < Owner >]
```

Description

When you create a Btrieve file, you can define the file only for a single volume. EXTEND enables you extend an existing file across two logical disk drives. This feature is useful when the data to be contained in a single file exceeds the physical storage capacity of a single disk or the maximum volume size the operating system supports.

How To Use EXTEND

To run EXTEND, enter the command in the format shown above.

Replace <*Btrieve File*> with the name of the Btrieve file you want to extend. You can specify a full path name if necessary.

Replace <*Extension File*> with the name you want to use for the extension file. Be sure to include the drive specifier for the new drive. The drive should not be the same as the one you specified for the original file. You can specify a full path name if necessary. Btrieve expects you to load the extension file in the specified drive every time you access the file.

Replace <*Owner*> with the owner name, if there is one.

If a file extends across two disks, you must load both disks before you access the file. Moreover, you must load the file's extension in the same drive you specified when you first extended the file. Once a file has been extended, you cannot reverse the operation.

Example

The following example extends the MAILER.ADR file to the MAILER2.ADR file in the \SALES directory of drive E. The owner name of the file is "Sales."

```
BUTIL –EXTEND MAILER.ADR E:\SALES\MAILER2.ADR –O Sales
```

INDEX

Command Format

```
BUTIL –INDEX < Btrieve File>< Index File>< Description File>[ –O < Owner>]
```

Description

The INDEX command builds an external index file based on a field that you had not previously specified as a key. The records in the new file consist only of the 4-byte address of each record in the original Btrieve file followed by the value on which you want to sort.

After Btrieve creates an external index, you can use the external index to retrieve the original file's data records in two ways:

- You can use the SAVE command to retrieve the file's records using the external index file. Refer to the discussion of the SAVE command beginning on page 4-34 for more information.
- You can develop an application that searches the file using the external index. The application should first retrieve the 4-byte address using the key value from the index file. Your application can then retrieve the record from the original file using the 4-byte address in a Get Direct operation.

How To Use INDEX

Before you can build an external index using the INDEX command, you must create a description file to specify the new key characteristics.

To run INDEX, enter the command using the format shown above.

Replace *<Btrieve File>* with the name of the existing Btrieve file for which you want to build an external index. You can specify a full path name if necessary.

INDEX
(Continued)

Replace *<Index File>* with the name of the file in which Btrieve should store the external index. You can specify a full path name if necessary.

NOTE:
Since both the original file and the index file may have a corresponding pre-image file, you should not use the same filename with two different extensions.

For *<Description File>*, substitute the name of the file you have created containing the new key definition. The file should contain a definition for each segment of the new key. You can specify a full path name if necessary. Refer to "BUTIL Description Files" beginning on page 4-4 for complete information about description files.

For *<Owner>*, substitute the owner name, if there is one, for the Btrieve file.

Sample Description File for BUTIL -INDEX

For example, the description file in Figure 4.2 defines a new key with one segment. The key begins at the 30th byte of the record, and is 10 bytes long. It allows duplicates, is modifiable, is a string type, and uses no alternate collating sequence.

```
position=30 length=10 duplicates=yes modifiable=yes type=string
alternate=no segment=no
```

Figure 4.2
Sample Description File for INDEX

After you define the key for the external file, INDEX creates the file. After the file is created, INDEX will display on the screen the number of records that were indexed.

INDEX
(Continued)

Example

The following command creates the QUICKREF.IDX external index file for the CUSTOMER.DAT file. The CUSTOMER.DAT file does not require an owner name. The description file containing the definition for the new key is NEWIDX.DES.

```
BUTIL –INDEX CUSTOMER.DAT  QUICKREF.IDX  NEWIDX.DES
```

LOAD

Command Format

```
BUTIL –LOAD < Input File > < Btrieve File > [–O < Owner >]
```

Description

The LOAD command allows you to insert records from a sequential file into a Btrieve file without writing an application program specifically for that purpose. LOAD also provides a convenient way to transfer records from a sequential file created by another program into a Btrieve file. LOAD performs no conversion on the data in the load file.

After Btrieve transfers the records, it displays on the screen the total number of records loaded into the Btrieve file.

How To Use LOAD

Before you run the LOAD command, you must create a sequential file that contains the new records. You can create the file using either a standard text editor or an application program.

To run LOAD, enter the command using the format shown above.

Replace *<Input File>* with the name of the sequential file containing the records to be loaded into a Btrieve file. You can specify a full path name if necessary.

Replace *<Btrieve File>* with the name of the Btrieve file into which you want the records inserted. You can specify a full path name if necessary.

Replace *<Owner>* with the owner name of the Btrieve file, if there is one.

LOAD
(Continued)

LOAD expects each record in *<Input File>* to be in the following format:

- The first *n* bytes should be the length of the record in ASCII.

  For files with fixed length records, the length specified should always equal the record length you specified when you created the file.

  For files with variable length records, the length you specify must be at least as long as the minimum fixed length you specified when you created the file.

- The length must be followed by a one-character separator (either a comma or a blank).

- The separator must be followed by the data itself. The length of the data must be the exact length specified at the beginning of the record.

- The record must be terminated with a carriage return/line feed (0D0A hex).

- The last record in the file should contain the end of file character (Ctrl-Z, or 1A hex). Most text editors and the SAVE command automatically insert this character in the file.

You can create your input file using either a text editor or an application program.

If you use a text editor to create your load file, be sure to pad each record with blank spaces as necessary to fill the record to the length you specified at the beginning of the record. Fields containing binary data cannot be edited with most text editors.

If you use an application program to create your load file, be sure to append a carriage return and line feed to each record and include an end of file record. The sequential I/O calls provided by most high-level language processors insert carriage return, line feed, and end of file characters automatically.

LOAD
(Continued)

Figure 4.3 illustrates the correct format for each record in the input file. Assume that the Btrieve file does not allow variable length records, and has a defined record length of 40 bytes.

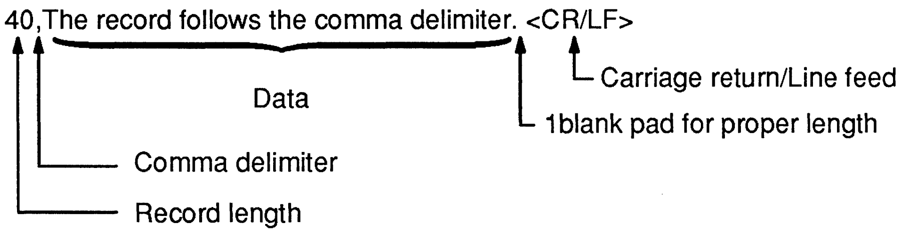

40,The record follows the comma delimiter. <CR/LF>
Data
Carriage return/Line feed
1blank pad for proper length
Comma delimiter
Record length

Figure 4.3
Record format for input file

Example

The following example loads sequential records from the MAIL.LST file into the MAILER.ADR file. The owner name of the MAILER.ADR file is "Sales."

```
BUTIL –LOAD MAIL.LST MAILER.ADR –O Sales
```

RECOVER

Command Format

```
BUTIL -RECOVER < Btrieve File > < Output File > [-O < Owner >]
```

Description

The RECOVER command reads records from a specified Btrieve file using the Step operations, and creates a sequential file that is compatible with the LOAD command. Each record ends with a carriage return and line feed (0D0A hex). The file terminates with an end of file record (1A hex).

You can use RECOVER to retrieve data from a damaged Btrieve file. For example, a file can be damaged if the system fails while the file is being accessed in accelerated mode. The RECOVER command may be able to retrieve many, if not all, of the records from the file. You can then use the LOAD command to insert the records into a new, undamaged Btrieve file.

How To Use RECOVER

To run RECOVER, enter the command using the format shown above.

Replace <*Output File*> with the name of the file where RECOVER should store the recovered records. You can specify a full path name if necessary.

Replace <*Btrieve File*> with the name of the Btrieve file that you want to recover. You can specify a full path name if necessary.

Replace <*Owner*> with the owner name for the Btrieve file, if there is one.

After RECOVER retrieves the records, it displays on the screen the total number of recovered records. If the logical drive containing your output file fills up before the entire Btrieve file has been recovered, RECOVER stops, displays the number of records already recovered, and then displays the following message:

```
Disk volume is full. Enter new file name to continue or . to quit,
then press <ENTER>.
```

RECOVER
(Continued)

To continue the operation in another output file, complete one of the following instructions:

- *If you are recovering the Btrieve file to diskettes*, remove the full diskette and replace it with another formatted diskette.
- *If you are recovering the Btrieve file to a hard disk*, specify another logical drive that has space available.

In either case, enter the name of a file that you want Btrieve to use to continue storing records and press the Enter key. Btrieve will continue copying records from the Btrieve file to the new output file.

If a logical drive fills up and you want to terminate the RECOVER operation, type a period (.) and press <Enter>.

Example

The following example retrieves records from the MAILER.ADR file and loads them into the MAIL.LST sequential file. The owner name of the MAILER.ADR file is "Sales."

```
BUTIL –RECOVER MAILER.ADR MAIL.LST –O Sales
```

RESET

Command Format

```
BUTIL -RESET [< Connection Number >]
```

Description

RESET performs a Btrieve Reset operation to release the resources used by BREQUEST and the Record Manager at a workstation. It releases all locks, aborts any pending transactions, and closes any open files for the station.

You can release the resources for a station other than your own by substituting the connection number of the station for *<Connection Number>*. If you do not know the connection number, be aware that the B ACTIVE, WHOAMI, and USERLIST console commands return connection numbers as part of their output.

How To Use RESET

To run RESET, enter the command using the format shown above.

You can issue this command from any workstation on the network at which BREQUEST is loaded. If you do not specify a station number, BUTIL -RESET releases the resources for the station issuing the command.

Example

The following example releases the resources for the workstation using connection number 12 on the network.

```
BUTIL -RESET 12
```

SAVE

Command Format

```
BUTIL -SAVE <Btrieve File><Output File><Index (Y/N)>
        [<Index File> | <Key Number>] [-O <Owner>]
```

Description

SAVE allows you to retrieve the records from a Btrieve file and store them in sorted order in a sequential file. It is the exact inverse of LOAD. This command can be used in conjunction with LOAD so that the data in a Btrieve file can easily be extracted, edited, and then stored in another Btrieve file.

SAVE generates a single record in the output file for each record in the Btrieve file it is reading. Each record is preceded by its length and ends with a carriage return and a line feed (0D0AH). The file terminates with an end of file record (1AH) and is compatible with most text editors. SAVE performs no conversion on the data in the records. Therefore, if you use a text editor to modify an output file containing binary data, the results may be unpredictable.

After SAVE completes its processing, it displays on the screen the total number of records saved.

How To Use SAVE

To run SAVE, enter the command in the format shown above.

Replace *<Btrieve File>* with the name of the Btrieve file containing the records you want to save. You can specify a full path name if necessary.

Replace *<Output File>* with the name of the sequential file in which you want Btrieve to store the records. You can specify a full path name if necessary.

SAVE
(Continued)

Use one of the following methods to specify the order in which you want SAVE to store the records:

- *If you want to save records by an external index,* specify Y for *<Index(*Y / N*)>* and replace *<Index File>* with the name of the external index file. You can specify a full path name if necessary.
- *If you want to save the records by a key other than key 0,* specify "N" for *<Index(*Y / N*)>* and replace *<Key Number>* with the appropriate key number.
- *If you want to save the records by key 0,* do not specify an index file or a key number.

Replace *<Owner>* with the owner name for the Btrieve file, if there is one.

If the logical drive containing your output file fills up before the entire file has been saved, SAVE stops, displays the number of records already saved, and then displays the following screen:

```
Disk volume is full. Enter new file name to continue or . to quit,
then press <ENTER>.
```

To continue the operation in another output file, complete one of the following instructions:

- *If you are saving the Btrieve file to diskettes,* remove the full diskette and replace it with another formatted diskette.
- *If you are saving the Btrieve file to a hard disk,* specify another logical drive that has space available.

In either case, enter the name of a file that you want Btrieve to use to continue storing records and press the Enter key. Btrieve will continue copying records from the Btrieve file to the new output file.

SAVE
(Continued)

If a logical drive fills up and you want to terminate the SAVE operation, type a period (.) and press <Enter>.

Example

The following two examples illustrate how to use SAVE to retrieve records from a file.

The first example uses the QUICKREF.IDX external index file to retrieve the records from the CUSTOMER.DAT file and store them in the CUST.SAV sequential file.

```
BUTIL –SAVE CUSTOMER.DAT CUST.SAV Y QUICKREF.IDX
```

The next example retrieves the records from the CUSTOMER.DAT file using key number 3 and stores them in the CUST.SAV sequential file.

```
BUTIL –SAVE CUSTOMER.DAT CUST.SAV N 3
```

SINDEX

Command Format

```
BUTIL –SINDEX < Btrieve File > < Description File > [ –O < Owner > ]
```

Description

SINDEX creates a supplemental index for an existing Btrieve file. The key number of the new index will be one higher than the previous highest key number for the file.

How To Use SINDEX

Before you can run SINDEX, you must provide a definition for the supplemental index in a description file. Refer to "BUTIL Description Files" beginning on page 4-4 for more information about BUTIL description files. Use the sample description file on page 4-26 as a guide for creating the SINDEX description file.

To run SINDEX, enter the command in the format shown above.

Replace *<Btrieve File>* with the name of the Btrieve file for which you are creating the index. You can specify a full path name if necessary.

Replace *<Description File>* with the name of the description file containing the description of the index you want to create. You can specify a full path name for this parameter.

Replace *<Owner>* with the owner name of the Btrieve file, if there is one.

Example

The following example creates a supplemental index for the MAILER.ADR file. The name of the description file is SUPPIDX.DES. The owner name of the Btrieve file is "Sales."

```
BUTIL –SINDEX  MAILER.ADR  SUPPIDX.DES  –O Sales
```

STAT

Command Format

```
BUTIL –STAT < Filename > [ –O < Owner > ]
```

Description

STAT reports the defined characteristics of a Btrieve file and statistics about its contents. You can use STAT to determine all the parameters specified for a file with CREATE. The STAT command also provides information on the volume of keys and records in the file and the name of the extension file, if one exists.

How To Use STAT

To run STAT, enter your command using the format shown above.

Replace *<Filename>* with the name of an existing Btrieve file whose statistics you want to retrieve. You may specify any number of directory levels in the filename.

Replace *<Owner>* with the owner name of the Btrieve file, if there is one.

Example

The following example retrieves the file statistics for the ADDRESS.BTR file. The file does not have an owner name.

```
BUTIL –STAT ADDRESS.BTR
```

Figure 4.4 illustrates the output from the above command.

This example shows that the file called ADDRESS.BTR was defined with a page size of 1,536 bytes, a record length of 147 bytes, and 2 keys. The file uses data compression, allows variable length records, and has a free space threshold of 10%.

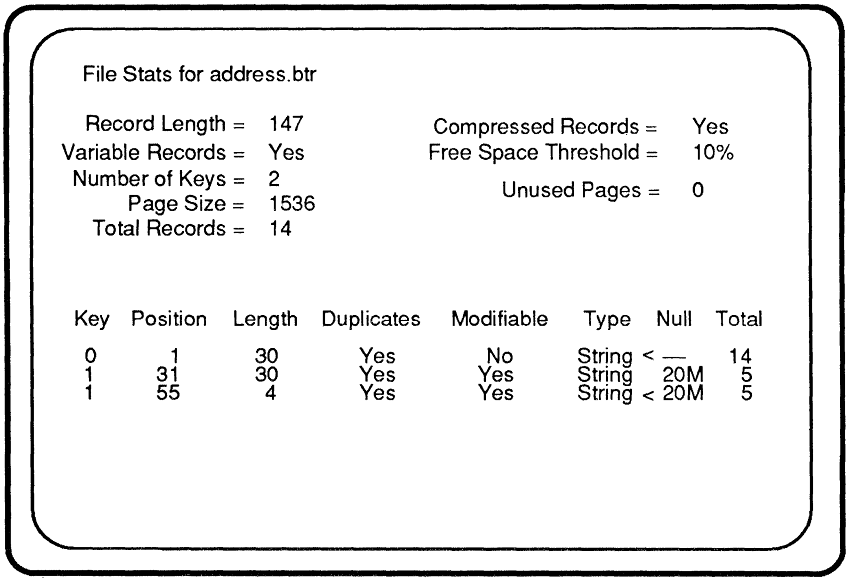
STAT
(Continued)

The first key (Key 0) consists of one segment, starts in position one, is 30 characters long, allows duplicates, is not modifiable, has a string key type, and does not have a null value defined. Key 0 is collated in descending order.

The second key (Key 1) allows duplicates, is modifiable, is manual, and has a null value of hex 20 (blank). It consists of two segments.

The first segment starts in position 31, is 30 bytes long, has a string key type, and has a null value of 20H (blank). The second segment starts in position 55, has a length of four, has a string key type, is descending, and has a null value of 20H (blank).

Fourteen records have been inserted into the file. The file contains 14 unique values for the first key and five unique values for the second. There is no extension file.



```
File Stats for address.btr

  Record Length =  147           Compressed Records =   Yes
Variable Records = Yes         Free Space Threshold =   10%
 Number of Keys =  2                    Unused Pages =   0
      Page Size =  1536
  Total Records =  14


Key  Position  Length  Duplicates  Modifiable   Type   Null  Total

0        1       30       Yes         No       String < —     14
1       31       30       Yes         Yes      String   20M    5
1       55        4       Yes         Yes      String < 20M    5
```

Figure 4.4
Sample BUTIL –STAT Output

STOP

Command Format

```
BUTIL -STOP
```

Description

STOP removes BREQUEST and the Btrieve Record Manager from memory and, when possible, returns the allocated memory to the operating system. Once you issue the STOP command, you cannot run a Btrieve application unless you reload BREQUEST or the Record Manager.

How To Use STOP

To run STOP, enter the command in the format shown above.

VER

Command Format

```
BUTIL -VER
```

Description

VER reports the version of BREQUEST that is loaded at that workstation.

How To Use VER

To run VER, enter your command in the format shown above.

BTRIEVE FUNCTION EXECUTOR

The Btrieve diskette includes a program called B that allows you to perform individual Btrieve operations interactively. B.EXE is a Btrieve application program that performs Btrieve operations based on the values you specify for the different prompts. Each prompt is explained in the chart on the next page. The B program is useful for learning how Btrieve operates, testing your program's logic, and debugging.

To execute B, type the following command at the DOS prompt:

```
B <Enter>
```

When you execute B, a menu will appear with prompts for each of the parameters required on a Btrieve call. A list of the Btrieve operation codes is displayed below the prompts.

To perform any Btrieve call, initialize all of the Btrieve parameters normally required for that operation. Refer to the discussion of the Btrieve record operations in Chapter 6 for information about the required parameters. For example, to perform an Open operation complete the following steps.

1) Specify an operation code of 0.

2) Specify an open mode in the "Key Number" prompt (if necessary).

3) Specify the file name in the "Key Buffer" prompt.

4) Press <F1> to execute the Btrieve operation.

   The B program makes the call to Btrieve and displays the resulting status.

You can continue to perform Btrieve operations as desired. To terminate the program, first close any open files, and then press the Escape key.

The following list describes the B utility menu prompts.

Prompt	**Description**
Function	Enter the operation code for the Btrieve operation you want to perform. A list of the operation codes will be displayed on the lower half of the screen. Appendix A also contains a list of the Btrieve operation codes.
Key Path	Specify the key number for the Btrieve operation in the key path field. Valid entries for this field are 0-23.
Position Block	Specify the file you want to access in the position block field. B assigns the number in the position block prompt to the file when you successfully open the file to Btrieve. You can have up to 10 open files. When you open files, specify a number for the file in the position block. Valid entries are from 0 to 9, inclusive. Begin with 0 for the first file you open, 1 for the second file, and so on. After you have opened a file, identify it for subsequent operations by entering its number in the position block field.
Status	Btrieve returns the status from each operation in this field. It is helpful to initialize the status to 99, or another unlikely code, before each operation. Doing so allows you to see the status code change when Btrieve completes the operation.
Data Buffer Length	Set the data buffer length to the correct value for the operation you want to perform.
Data Buffer	Enter the data for the record in this field for Insert or Update operations. You can only enter data in ASCII format. For Get operations, Btrieve returns the data you requested in the data buffer. Only ASCII data is displayed on the screen.

Prompt	Description
Key Buffer	Store either the filename or a key value, depending on the operation you are performing, in the key buffer. As with the data buffer, only ASCII data can be entered or displayed.

The following key sequences can be used when you are running the B program:

Keys	Description
<Esc>	Terminates the program
<F1>	Executes the Btrieve call
<Home>	Moves to the first prompt
<End>	Moves to the last prompt
<Up-Arrow>	Moves to the previous prompt
<Down-Arrow>	Moves to the next prompt
<Left-Arrow>	Moves one character left
<Right-Arrow>	Moves one character right
<Backtab>	Moves to the previous prompt
<Tab>	Moves to the next prompt
<Ctrl-Home>	Moves to the beginning of the field
<Ctrl-End>	Moves to the end of the field
<Delete>	Deletes a character
<Insert>	Toggles insert mode

You can specify only ASCII values in the data buffer and key buffer parameters.

CONSOLE COMMANDS

Btrieve provides console commands that allow you to determine the file activity and the current level of utilization of the BSERVER process. You can execute these commands at the console of any file server in which a BSERVER.VAP is loaded.

The following sections describe the NetWare Btrieve console commands. Each description includes the following information:

- **Command Format.** This section presents the console command in the format in which it should be entered at the file server where BSERVER is loaded.
- **Purpose.** This section describes the uses of the command.
- **How to Use (the Command).** This section describes how to issue the command, and the command's results.

B ACTIVE

Command Format

```
B ACTIVE < Screen >
```

Description

The B ACTIVE console command allows you to list all Btrieve files that are currently open at a server, and to view the connection number (workstation) that has each file open and the lock types that are held for each file.

Btrieve will display the results of the command in a tabular format on the screen. The following information will be displayed for each file:

- The full pathname of the file
- The network connection number that has the file open
- The type of locks held for the file

You can use the connection number returned by B ACTIVE to determine which user has the file open. You can also use this number as a parameter in a B RESET command, a BUTIL –RESET command or a Btrieve Reset operation to close the files and release the resources of a particular station.

If another VAP has the file open, B ACTIVE will display a two-character code for the connection number. For example, the two-character code for NetWare SQL is "NS". You cannot use this two character code as input for a B RESET command. Refer to the discussion of the B RESET command on page 4-50 for more information.

The codes for the three lock types are as follows:

Lock	**Code**
Transaction	T
Single record	A
Multiple Record	M

B ACTIVE
(Continued)

How to use B ACTIVE

To run B ACTIVE, enter the command in the format shown above at the server where BSERVER.VAP is loaded.

Insert a blank space between B and ACTIVE. You can enter the command in either upper- or lower-case characters.

As long as there are more active files to display, B ACTIVE will display a message so indicating at the bottom of the screen. To see additional screens, re-enter the B ACTIVE command, substituting a number for <*Screen*> that corresponds to the screen you want to view. For example, to view the second screen of a B ACTIVE operation type the following:

```
B ACTIVE 2 <Enter>
```

B DOWN

Command Format

```
B DOWN
```

Description

The B DOWN console command releases all resources used by BSERVER, and terminates the BSERVER process. When you issue B DOWN, Btrieve will

- Close all Btrieve files open at the file server;
- Release all of the locks Btrieve holds at the file server;
- Abort any pending transactions at the file server;
- Halt the BSERVER process.

If you are running another VAP that uses BSERVER to access network files, you should take the following precautions before you issue the B DOWN command:

- Issue the B ACTIVE command to make sure that no files are open by another VAP.
- If any other VAP has Btrieve files open at the file server, you should issue the appropriate command for that VAP that closes its files and halts the process.

For example, if you were running NetWare SQL, you would issue the NS RESET and NS DOWN commands to make sure that all NetWare SQL files were closed. After you issue the B DOWN command, workstations and other VAPs will not be able to access Btrieve files through BSERVER until the file server is restarted.

How to use B DOWN

To run B DOWN, enter the command in the format shown above at the server where BSERVER is loaded. Insert a blank space between B and DOWN. You can enter the command in either upper- or lower-case characters.

B OFF

Command Format

```
B OFF
```

Description

The B OFF console command causes the previous Btrieve command line utility to stop refreshing the screen.

You should issue B OFF after B ACTIVE, B STATUS, or B USAGE have displayed the information you need. If you do not issue B OFF, these commands will continue to refresh the screen, even after you issue another command.

How to use B OFF

To run B OFF, enter the command in the format shown above at the server where BSERVER.VAP is loaded.

Insert a blank space between B and OFF. You can enter the command in either upper or lower case.

B RESET

Command Format

```
B RESET  < connection number >
```

Description

The B RESET console command releases all resources used by a particular station on the network. When you issue a B RESET console command followed by a station's connection number, Btrieve

- Closes all open files at the station
- Releases all locks held by the station
- Aborts any pending transaction at the station

How to Use B RESET

To run B RESET, enter the command in the format shown above at the server where BSERVER.VAP is loaded. Insert a blank space between B and RESET. You can enter the command in either upper- or lower-case characters.

Substitute the connection number of the station that you want to reset for *<connection number>*. For example, to release all resources for station 12, you would issue the following command at the server console:

```
B RESET 12  <Enter>
```

Use an asterisk (*) to denote all the Btrieve stations on the network, instead of a single station. To reset all the stations on the network that have Btrieve files open, issue the following command at the server console:

```
B RESET *  <Enter>
```

The B RESET operation will not accept the two-character ASCII connection ID, which signifies a VAP, as input. To reset any files open by a VAP, you must use the appropriate reset command for that VAP.

B STATUS

Command Format

```
B STATUS
```

Description

You can use B STATUS to help determine whether the level of resources allocated for BSERVER is the most efficient level for your environment.

B STATUS returns information about network requests, packet buffers, and the sessions in use for the file server where you issue the command. The command also returns the number of times the console screen has been refreshed since the command was issued.

How to Use B STATUS

To run B STATUS, enter the command in the format shown above at the server where BSERVER.VAP is loaded. You may enter the command in either upper- or lower-case characters. Insert a blank space between each word.

When you execute B STATUS, Btrieve will display the following screen:

```
            Status for NetWare Btrieve Server VAP v5.xx

    Current, Total requests processed:       nn    nn
    Available, Max request buffers:          nn    nn

    Available, Max SPX packet buffers:       nn    nn
    Unprocessed SPX packet buffers:          nn
    Current, Total SPX packets received:     nn    nn
    Current, Total SPX packets sent:         nn    nn
    Current.Total SPX requests processed: nn nn
    Current, Max, Peak SPX sessions:         nn    nn    nn

    Number of display iterations:            nn
```

B STATUS
(Continued)

"Current" values are the values accumulated since the B STATUS command was issued, and are displayed in the first column. "Total" values are the values accumulated since BSERVER was loaded, and are displayed in the second column. "Max" values are the maximum values for the resource that are available in the network's current configuration, and are displayed in the second column. "Peak" values reflect the highest utilization of the resource since BSERVER was loaded. The following paragraphs describe the information returned by the B STATUS command.

"Current, Total requests processed" reflects the number of network requests processed by BSERVER from both workstations and other VAPS, such as NetWare SQL.

"Available, Max request buffers" reflects the number of processes active for the VAP. For BSERVER, this value should always be 1.

"Available, Max SPX packet buffers" reflects the number of NetWare packet buffers available to BSERVER, and the maximum number available at the server.

"Unprocessed SPX packet buffers" reflects the difference between the maximum number of NetWare packet buffers available at the server and the number available to BSERVER.

"Current, Total SPX packets received" reflects the number of network packets received by BSERVER from workstations.

"Current, Total SPX packets sent" reflects the number of network packets sent to workstations by BSERVER.

"Current, Max, Peak SPX sessions" reflects the number of BSERVER sessions in use, and the maximum number available to BSERVER.

"Number of display iterations" reflects the number of times the B STATUS screen has been refreshed since the B STATUS command was issued.

B USAGE

Command Format

```
B USAGE
```

Description

You can use B USAGE to help determine whether BSERVER is configured to the most efficient level for your environment.

The B USAGE console command returns information about the following Btrieve configuration options:

- Open files
- File handles
- Locks
- Transactions

How to use B USAGE

To run B USAGE, enter the command in the format shown above at the server where BSERVER.VAP is loaded. You may enter the command in either upper- or lower-case characters. Insert a blank space between B and USAGE. When you execute B USAGE, Btrieve will display the following screen:

```
            Usage for NetWare Btrieve Server VAP v5.xx

    Current, Max, Peak files:              nn    nn    nn
    Current, Max, Peak handles:            nn    nn    nn
    Current, Max, Peak  locks:             nn    nn    nn
    Current, Max, Peak transactions:       nn    nn    nn
```

“Current” values are the values that are currently in use, and are displayed in the first column. “Max” values are the maximum values for the resource that are available in the current BSERVER configuration, and are displayed in the second column. “Peak” values reflect the highest usage of the resource since BSERVER was loaded.

5 APPLICATION INTERFACES

The Btrieve application interfaces give you access to Btrieve file structures from your application program. Through these routines, your application issues calls which identify the operation to perform, the data to pass or receive, and the status and positioning information. This chapter describes how to call Btrieve from programs written in any of the following languages:

- IBM (or Microsoft) BASIC and Compiled BASIC
- IBM (or Microsoft) Pascal
- Turbo Pascal
- Microsoft COBOL
- Microsoft C
- Lattice C
- Assembly Language

For a description of interfaces that are not described in this manual, see the file called INTRFACE.DOC on your Btrieve program diskette.

NOTE:
Before you can call Btrieve from an application in any language, you must load the Btrieve Requester (BREQUEST) into memory at the workstation.

INTERFACING BTRIEVE WITH BASIC

The interface to Btrieve is essentially the same from both compiled and interpretive BASIC. For Compiled BASIC, you link the Btrieve interface with your Basic program after compiling. For interpretive BASIC, you load the Btrieve interface as a memory resident program. The format of the Btrieve calls is identical for both.

INTERPRETIVE BASIC

To call Btrieve from interpretive BASIC, your application must initiate BASIC execution with the appropriate parameters and load the Btrieve BASIC Interface correctly. If you do not accomplish these two steps correctly, the Btrieve application will not run properly, if at all.

THE INTERPRETIVE BASIC INTERFACE

The Btrieve BASIC Interface is an assembly language subroutine, called BASXBTRV.EXE, that the BASIC application must call in order to communicate with the Btrieve Record Manager.

In the MS-DOS operating system, the BASIC interface is a memory-resident program that you must load before you can run your BASIC application. Each workstation must have its own copy of BASXBTRV loaded. Once you load the Btrieve BASIC Interface, your application uses CALL statements to perform Btrieve operations.

The interface routine writes a single record to the output file containing the segment address, in decimal notation, at which it loaded. Your BASIC program reads that file and uses the segment address stored there in a DEF SEG statement. After your program performs the DEF SEG, it can use the CALL statement (described later in this chapter) to communicate with the Btrieve Record Manager.

In order to load the memory resident interface, enter the following command:

```
< Drive >: BASXBTRV < Filename.Extension >
```

Replace *<Drive>* with the name of the device containing the Btrieve files.

Replace *<Filename.Extension>* with the name of the file that will contain the interface's segment address. You must specify the file name as

```
< Drive > : < Filename.Extension >
```

You can omit *<Drive>* if you intend to use the default device. Once you have loaded the interface, it remains in memory until you restart your system.

In a network environment, it is important that the filename you specify for BASXBTRV to initialize is either a local or a unique filename. Since the segment address where BASXBTRV loads may vary between workstations, each workstation must have its own segment address file to read. For example, to load BASXBTRV and specify SEGMENT.ADR as the file for the segment address, you would issue the following command:

```
BASXBTRV SEGMENT.ADR
```

After the BASIC interface loads into memory and writes its segment address to a file, it displays the following message:

```
Btrieve Basic interface loaded at segment xxxxx
```

Your interpretive BASIC program should include the following statements that read and define the segment address to use for Btrieve calls:

```
30 OPEN "SEGMENT.ADR" FOR INPUT AS #1        'Open file containing segment address
40 INPUT #1, SEG.ADDR%                          'Get segment address of interface
50 DEF SEG = SEG.ADDR%                              'Set address for Btrieve calls
```

Figure 4.1 illustrates the various programs that are loaded in memory when you run a Btrieve application written in interpretive BASIC. MS-DOS loads first, followed by the memory resident BASIC interface, BASXBTRV. Btrieve is loaded after the interface. The remaining memory is available to your application program.

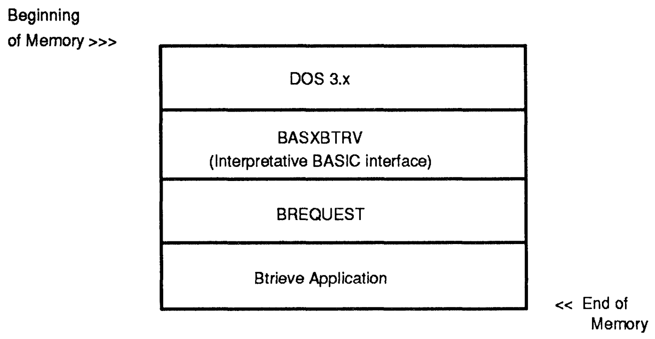


Figure 5.1
Map for Memory Resident BASIC Interface

INITIATING THE BASIC INTERPRETER

Normally, BASIC assumes a record length of 128 bytes for any file a program opens. To access a Btrieve file with a logical record length greater than 128 bytes, you must include a file size parameter specifying the file's logical record length in the command that executes the BASIC interpreter. Therefore, to execute the BASIC interpreter for a Btrieve application, enter the following command:

```
BASIC  [/S:yyy]
```

In the above example, *yyy* is the logical record length of the largest Btrieve file your program will access. See your BASIC reference manual for more information on specifying this option.

COMPILED BASIC

To execute a compiled BASIC program that calls Btrieve, you must link the appropriate Btrieve interface routines with your compiled BASIC object file. The Btrieve diskette contains the file that you must include in your BASIC link: BASXBTRV.OBJ. For a complete explanation of linking, refer to your operating system manual and your BASIC reference manual.

In order to link a BASIC program for which the object is stored in the file called BASPROG with the Btrieve BASIC interface BASXBTRV.OBJ, you would respond to the linker prompt for object modules as follows:

```
Object Modules [.OBJ]:basprog+basxbtrv
```

NOTE:
Microsoft Quick BASIC uses the file QBIXBTRV.OBJ as the Btrieve interface routine, and requires different procedures from other versions of BASIC for opening files and referencing the data buffer. Refer to the INTRFACE.DOC file on the Btrieve *PROGRAM* diskette.

CALLING BTRIEVE FROM BASIC

Whether you are using compiled or interpretive BASIC, the steps for calling Btrieve are the same. To access data in a Btrieve file, your BASIC application must first execute a standard BASIC OPEN statement to NUL to allocate a BASIC field buffer, as in the following statement:

```
OPEN "NUL" AS #1
```

When BASIC processes an OPEN statement, it allocates an area called the File Control Block (FCB). This block contains, among other things, a buffer area which stores records from the file as they are transferred to and from the disk. BASIC allows you to define this buffer area as a set of contiguous string variables with the FIELD statement.

For example, if a you define a file to contain addresses, your application might include the following FIELD statement:

```
FIELD #1, 30 AS NAM$, 30 AS STREET$, 30 AS CITY$, 2 AS STATE$,
    5 AS ZIP$
```

This statement indicates that the field buffer, which was previously allocated for file #1, contains records in which the first 30 characters contain a name, and the next 30 characters contain a street, etc.

BASIC restricts your total statement length to 255 characters. If your record contains very many fields, you may not be able to completely describe your data in a single BASIC statement. BASIC allows you to use as many FIELD statements as necessary to describe the records. The variable names in all the field statements are in effect at the same time. Each new FIELD statement redefines the buffer from the first character position. Therefore, you have to use a dummy field as the first entry in subsequent FIELD statements to account for the fields which have already been defined.

For example, if the records defined by the previous FIELD statement contained a phone number after the zip code, you could define the phone number field in the following statement:

```
FIELD #1, 97 AS DUMMY$, 7 AS PHONE$
```

Since Btrieve uses the buffer in the FCB for record transfers, the application must include a FIELD statement in order to access the data returned by Btrieve. See your BASIC reference manual for a more complete description of the OPEN and FIELD statements. You must use an LSET command to store values in the buffer defined by a FIELD statement.

After a standard BASIC OPEN statement opens the Btrieve file, your application is ready to issue calls to the Btrieve Record Manager. First your application performs a Btrieve Open operation. After that, Btrieve handles all file reads, writes, and modifications through Btrieve calls. Your application should perform a Btrieve Close operation before it terminates.

All calls to Btrieve from a BASIC program must be in the following format:

```
CALL BTRV (Operation, Status, FCB, Data Buffer Length, Key Buffer,
                          Key Number)
```

For interpretive BASIC, BTRV should be a numeric variable with a value of 0. In compiled BASIC, BTRV is an external name resolved by the linker. Though all parameters are required on every call, Btrieve does not use all the parameters to perform every operation. In some cases, Btrieve ignores their value. For a more detailed description of relevant parameters, see Chapter 5 of this manual. The following sections describe each parameter.

OPERATION CODE

The operation parameter determines which Btrieve function you want to perform. The variable you specify must be an integer type and can be any one of the legal Btrieve operation codes described in Chapter 6 of this manual. (Also see Appendix A for a complete list of these codes.) Your application must specify a valid operation code on every Btrieve call. The Btrieve Record Manager never changes the code.

STATUS CODE

The status parameter contains a coded value that indicates whether any errors occurred during the Btrieve operation. The Btrieve Record Manager returns a status of 0 after a successful operation. Btrieve indicates any errors that occur during processing by returning a non-zero value in the status parameter.

A BASIC application must always pass an integer variable as the status parameter on a Btrieve call. After a Btrieve call, the application must always check the value of the status variable to ascertain whether the call was successful. See Appendix B for a list of Btrieve error messages and their possible causes.

FILE CONTROL BLOCK (FCB)

BASIC allocates an area called the File Control Block (FCB) when it processes an OPEN statement. Btrieve uses this block to maintain its positioning information and to transfer data records. Therefore, your application must pass the address of the FCB to the Record Manager on every call. Your application should use a different FCB address for each separate Btrieve file it accesses.

To determine the address of the FCB, your BASIC application must use a VARPTR statement. In the following example, BASIC returns the address of the FCB for the file opened as #1, in the integer variable FCB.ADDR%.

```
FCB.ADDR% = VARPTR(#1)
```

See your BASIC reference manual for a more complete description of the VARPTR statement.

DATA BUFFER LENGTH

For any operation using the data buffer, your program must pass the length of the data buffer in an integer variable. For a file with fixed length records, this parameter should match the record length specified when you first created the file. When you are inserting records in or updating a file with variable length records, this parameter should equal the record length specified when you first created the file, plus the number of characters included beyond the fixed length portion. When you are retrieving variable length records, this parameter should be large enough to accommodate the longest record in the file.

KEY BUFFER

On each Btrieve call, your BASIC program must pass a string variable containing the key value. If the key value is an integer, your program must convert it to a string using the MKI$ statement before calling Btrieve. If the key consists of two or more non-contiguous segments, you must concatenate them into a single string variable and pass the variable as the key buffer. Depending on the operation, your program may set the variable, or the Record Manager may return it.

The Record Manager returns an error if the string variable passed as the key buffer is shorter than the key's defined length. If your application's first call does not require initialization of the key buffer, you should assign the string variable the value SPACE$(*x*), where *x* represents the key's defined length. Until your application assigns some value in BASIC to the string variable, it has a length of 0.

KEY NUMBER

You may define up to 24 different keys when you first create a Btrieve file. When your application accesses the file, it must tell the Record Manager which access path to follow for a particular operation. The key number parameter is an integer variable with a value from 0 through 23, with 0 being the first key segment defined for the file. Btrieve never alters this value.

PARAMETER LIST EXAMPLE

The BASIC code shown in Figure 4.2 below opens a Btrieve file and retrieves the data record corresponding to the first value for key 0, the name field.

```
' Lines 5 through 20 apply only to interpretive BASIC. Do not include them in Compiled BASIC.
  5  BTRV = 0
 10  OPEN "SEGMENT.ADR" FOR INPUT AS #1              'Open file containing seg address
 15  INPUT #1, SEG.ADDR%                             'Get segment address of interface
 20  DEF SEG = SEG.ADDR%                                 'Set address for Btrieve calls
 30  OPEN "NUL" AS #2                                          'Open file from BASIC
 40  FIELD #2, 30 AS NAM$, 30 AS STREET$,
     30 AS CITY$, 2 AS STATE$, 6 AS ZIP$
 50  OP% = 0 : STATUS% = 0                  'Set Open operation code and initialize status
 70  FCB.ADDR% = VARPTR(#2) : BUF.LEN% = 98    'Get address of FCB, set buffer length
 80  KEY.BUF$ = "ADDRESS.BTR "                                 'Initialize key buffer
 90  KEY.NUM% = 0                                            'Use key 0 access path
100  CALL BTRV (OP%, STATUS%, FCB.ADDR%, BUF.LEN%, KEY.BUF$, KEY.NUM%)
110  IF STATUS% <> 0 THEN
     PRINT "Error opening file. Status = ", STATUS%  : END
120  OP% = 12                                         'Set Get First operation code
125  KEY.BUF$ = SPACE$(30)
130  CALL BTRV (OP%, STATUS%, FCB.ADDR%, BUF.LEN%, KEY.BUF$, KEY.NUM%)
140  IF STATUS% <> 0 THEN
     PRINT "Error reading file. Status = ", STATUS%  : END
150  PRINT "First record in file is:", NAM$, STREET$, CITY$, STATE$, ZIP$
```

Figure 5.2
Btrieve Call from BASIC

INTERFACING BTRIEVE WITH PASCAL

In order to access a Btrieve file, your Pascal application must define BTRV as an integer function. When your application calls that function, it performs various types of file accesses, depending on which parameters you specify. The Pascal interface communicates with the Btrieve Record Manager. You must load the Record Manager, a memory-resident assembly language program, before you start your application.

Declare the Btrieve function as external for IBM (or Microsoft) Pascal. Btrieve provides a small assembly language routine that you can link with your Pascal application as an external function. For Turbo Pascal, Btrieve provides the source code of the interface so that you can include it with your Pascal program when you compile it.

The Btrieve program diskette contains a file that needs to be included in your Pascal source file. For IBM Pascal, the file contains the external function declaration for BTRV. For Turbo Pascal, the file contains the code for the entire Btrieve interface.

Your application accomplishes all Btrieve file access by calling the function BTRV. This is an integer function which returns the status of the operation. If you are using IBM (or Microsoft) Pascal, use the $INCLUDE metacommand to include the file BEXTERN.PAS. The example below shows how the external function, BTRV, is defined:

```
function BTRV  (      OP             : integer;
               vars  POS_BLOCK      : string;
               vars  DATA_BUFFER    : string;
               vars  DATA_LEN       : integer;
               vars  KEY_BUFFER     : string;
                     KEY_NUMBER     : integer )  : integer; extern;
```

If you are using Turbo Pascal, use the $I command to include the file TURXBTRV.PAS. The Btrieve function is defined as follows:

```
function BTRV (      OP            : integer;
                var  POS_BLOCK,
                var  DATA_BUFFER;
                var  DATA_LEN      : integer;
                var  KEY_BUFFER;
                     KEY_NUMBER    : integer): integer;
```

LINKING A PASCAL APPLICATION WITH BTRIEVE

If you are using IBM (or Microsoft) Pascal, you must include the file called PASXBTRV.OBJ in your Pascal link. To link the Pascal object file (PASPROG) with the IBM Pascal interface, you would respond to the linker prompt for object modules as follows:

```
Object Modules [.OBJ]: pasprog+pasxbtrv
```

If you are using Turbo Pascal, include the interface source file (TURXBTRV.PAS) with your program when you compile.

CALLING BTRIEVE FROM PASCAL

Your Pascal application should never perform any standard Pascal I/O against a Btrieve file. Your application should perform all I/O to a Btrieve file using the Btrieve function. The first Btrieve call your application must perform is an Open operation. Following that, it can read, write, and modify files through Btrieve calls. Before your application terminates, it should perform a Btrieve Close operation.

All calls to Btrieve must be performed through the BTRV function. The result of the function is always an integer value which corresponds to one of the status codes listed in Appendix B. After a Btrieve call, your application should always check the value of the status variable. A status of 0 indicates a successful operation. Your application should be able to recognize and resolve a non-zero status.

Although you must provide all parameters on every call, Btrieve does not use every parameter for every operation. See Chapter 6 for a more detailed description of which parameters are relevant for each operation. The following sections describe the parameters.

OPERATION CODE

The operation parameter determines which type of Btrieve function you want to perform. Your application is responsible for specifying a valid operation code on every Btrieve call. The Record Manager never changes the operation code. The variable you specify must be an integer type and can be any one of the legal Btrieve operation codes described in Chapter 6. Appendix A contains a complete list of these codes.

POSITION BLOCK

Your application must allocate a separate position block for each Btrieve file it opens. Btrieve initializes the position block when your application performs the Open operation, and references and updates the data in the position block on all file operations. Therefore, your application should pass the same position block on all subsequent Btrieve operations for the file. When your application has more than one file open at a time, Btrieve uses the position block to determine which file is referenced in a particular call. In addition, your application should never change the values contained in the position block.

An IBM Pascal application must allocate a 128-byte string for the position block. If you are using Turbo Pascal, you should allocate the position block parameter as a 128-byte character array.

DATA BUFFER

The data buffer contains the records that your application transfers to and from the Btrieve file. Btrieve expects a string type for IBM Pascal. For Turbo Pascal, you can use any data type.

You may want to define a record structure in Pascal to describe the data stored in a file. To pass a record type variable in IBM Pascal, use the case option to define a string type variant for the structure. For Turbo Pascal, you can send the record itself.

When you calculate the length of the variant string, take into account the fact that odd length elements in a record may require an extra byte of storage whether or not the record is packed. This is also an important consideration when you define the record length for the CREATE utility. See your Pascal reference manual for more information on record types.

DATA BUFFER LENGTH

For any operation that requires a data buffer, your program must pass the length of the data buffer in an integer variable. For a file with fixed length records, this parameter should match the record length specified when you first created the file.

When you are inserting records into or updating a file with variable length records, this parameter should equal the record length specified when you first created the file, plus the number of characters included beyond the fixed length portion. When you are retrieving variable length records, this parameter should be large enough to accommodate the longest record in the file.

KEY BUFFER

Your application must pass a string variable for IBM Pascal, or any type variable for Turbo Pascal, that will contain the key value on each Btrieve call. Depending on the operation, your application may set this variable, or the Record Manager may return it.

For IBM Pascal, if the key is an integer, you should define it as a record structure with two variants. One variant defines the key as an integer. The other defines it as a two character string. You must use the string variant for Btrieve calls.

For Turbo Pascal, you can pass the key buffer itself, regardless of type.

If the key consists of two or more segments, use a record structure to define the individual fields in the key. Then use a variant to pass the key buffer to Btrieve.

KEY NUMBER

You may define up to 24 different keys when you create a Btrieve file. Therefore, your application must tell the Record Manager which access path to follow for a particular operation. The key number parameter is an integer variable with a value from 0 through 23, with 0 being the first key segment defined for the file. The Record Manager never alters this parameter.

PARAMETER LIST EXAMPLE

The IBM (or Microsoft) Pascal code shown in Figure 5.3 below opens a Btrieve file and retrieves the data record corresponding to the first value for key 0, the name field.

```
const
  B_GET_FST  = 12;          {Get first}
  B_OPEN     = 0;           {Open file}

type
  ADDRESS_REC = record                                   {Structure of address file entry}
   case integer of
     1: (NAME               : string(30);
         STREET             : string(30);
         CITY               : string(30);
         STATE              : string(2);
         ZIP                : string(5));
     2: (ENTIRE             : string(98))
  end;

var
  DATA_BUF                  : ADDRESS_REC;
  DB_LEN                    : integer;
  FILE_NAME                 : string(14);
  KEY_BUF                   : string(30);
  POS_BLOCK                 : string(128);
  STATUS                    : integer;
```

Figure 5.3
Btrieve Call from IBM Pascal

```
begin
    FILE_NAME := 'B:ADDRESS.BTR ';
    STATUS := BTRV (B_OPEN, POS_BLOCK, DATA_BUF.ENTIRE, DB_LEN,
                        FILE_NAME, 0);
    if STATUS <> 0 then
        begin
            writeln (OUTPUT, 'Error opening file. Status = ',STATUS); return;
        end;
    DB_LEN := sizeof (ADDRESS_REC);
    STATUS := BTRV (B_GET_FST, POS_BLOCK, DATA_BUF.ENTIRE, DB_LEN,
                        KEY_BUF, 0);
    if STATUS <> 0 then
        writeln (OUTPUT, 'Error reading file. Status = ',STATUS)
    else
        writeln (OUTPUT, 'First record in file is:',DATA_BUF.ENTIRE);
end.
```

Figure 5.3 (*Continued*)
Btrieve Call from IBM Pascal

Figure 5.4 illustrates the same program written for Turbo Pascal. This is the only Turbo Pascal example in this manual. All other Pascal examples are shown for IBM (or Microsoft) Pascal.

In Figure 5.4, the application uses character arrays instead of strings for the fields in the data buffer and the key buffer because Turbo Pascal stores a binary length byte in the first position of a string field when it initializes the field. If you attempt to use such a value as a key in a Btrieve file without defining it as lstring, the results are unpredictable. When Btrieve compares key values for random or sequential searching, it compares them byte-by-byte on an absolute basis. The length byte is treated as part of the value instead of as an indicator of length, unless the key is defined as an lstring type.

Although the example in Figure 5.4 uses variant records for the position block, data buffer, and key buffer parameters, Btrieve does not require that you do this. This example simply illustrates one way of writing this program.

```
const
  B_GET_FST = 12;                                                    {Get first}
  B_OPEN = 0;                                                        {Open file}

type
 ADDRESS_REC = record
{Structure of address file entry}
    case integer of
         1:(NAME         : array [1..30] of char;
            STREET       : array [1..30] of char;
            CITY         : array [1..30] of char;
            STATE        : array [1..2] of char;
            ZIP          : array [1..5] of char);
         2:(START        : integer);
   end;
 FILE_NAME = record
    case integer of
         1:(VALUE        : array [1..14] of char);
         2:(START        : integer);
    end;
 KEY_BUF = record
 case integer of
         1:(VALUE        : array [1..30] of char);
         2:(START        : integer);
end;
var
  DATA_BUF               : ADDRESS_REC;
  DB_LEN                 : integer;
  FNAME                  : FILE_NAME;
  KBUF: KEY_BUF;
  POS  : record
    case integer of
           1: (START     : integer );
           2: (BLK       : array[1..128] of byte);
    end;
  STATUS                 : integer;
   I                     : integer;
 {$I TURXBTRV.PAS }
```

Figure 5.4
Btrieve Call from Turbo Pascal

```
begin
    FNAME.VALUE := 'ADDRESS.BTR ';
    STATUS := BTRV (B_OPEN, POS.START, DATA_BUF.START, DB_LEN,
                        FNAME.START, 0);
    if STATUS <> 0 then
        writeln ('Error opening file. Status = ', STATUS)
    else
        begin
            DB_LEN := sizeof (ADDRESS_REC);
            STATUS := BTRV (B_GET_FST, POS.START, DATA_BUF.START, DB_LEN,
                                KBUF.START, 0);
            if STATUS <> 0 then
                writeln ('Error reading file. Status = ', STATUS)
            else
                writeln ('First record in file is:', DATA_BUF.NAME, DATA_BUF.STREET,
                        DATA_BUF.CITY, DATA_BUF.STATE, DATA_BUF.ZIP);
        end;
end.
```

Figure 5.4 (*Continued*)
Btrieve Call from Turbo Pascal

INTERFACING BTRIEVE WITH COBOL

In order to access a Btrieve file, your IBM or Microsoft COBOL application must issue a CALL statement. The type of file accesses Btrieve makes when it executes the statement depends on the parameters you specify. Btrieve provides a small assembly language routine, the COBOL interface, which you must link with your COBOL application. This interface communicates with the Record Manager, which you must load before you start your application.

LINKING A COBOL APPLICATION WITH BTRIEVE

If you are using Microsoft's v2 COBOL compiler, the Btrieve interface must be linked with the MS-COBOL object modules to produce a new runtime executor. Refer to "Creating and Linking Assembly Language Subroutines" in the *Microsoft COBOL User's Guide* for a more complete description of the required procedures.

The following files are available for these procedures: ASM.OBJ (a table of subroutines that contains only the entry point BTRV), COBXBTRV.OBJ (object code for the Btrieve COBOL interface), and MAKERUN.BAT (batch file which does not require the assembler). To create a new runtime executor, RUNSUB.EXE, that includes the Btrieve interface, enter the following command:

```
MAKERUN RUNSUB COBXBTRV
```

CALLING BTRIEVE FROM COBOL

Your application should never perform any standard COBOL I/O against a Btrieve file. It should handle I/O through a call to Btrieve. After issuing an Open operation, your program can read, write, and modify files through Btrieve calls. Before terminating, your application should perform a Btrieve Close operation.

Format all calls to Btrieve from a COBOL program according to the example below:

```
CALL 'BTRV' USING OPERATION, B–STATUS, POSITION–BLOCK,
DATA–BUFFER,DATA–LEN, KEY–BUFFER, KEY–NUMBER.
```

Although you must provide all parameters for every call, Btrieve does not use every parameter to perform every operation. In some cases, Btrieve ignores their value. See Chapter 6 for a more detailed description of which parameters are relevant for each operation. The following sections describe each parameter.

OPERATION CODE

The operation parameter determines which type of Btrieve function you want to perform. The operation may be a read, write, delete, or update. Your application is responsible for specifying a valid operation code on every Btrieve call. The Record Manager never changes the code. The variable you specify must be COMP-0 type for IBM (or Microsoft) COBOL and can be any one of the legal Btrieve operation codes described in Chapter 6. See Appendix A for a complete list of these codes.

STATUS CODE

All calls to Btrieve return a 2-byte integer status value (a COMP-0 field for IBM or Microsoft COBOL) that corresponds to one of the status codes listed in Appendix B. After a Btrieve call, your application should always check the value of the status variable. A status of 0 indicates a successful operation. The application should recognize and resolve a non-zero status.

POSITION BLOCK

A COBOL application must allocate a 128-byte record which Btrieve uses to store the file I/O structures and the positioning information described in Chapter 2. Your application must allocate a separate position block for each Btrieve file it opens. Btrieve initializes this record when your application performs the Open operation, and references and updates the data in this record on all file operations. Therefore, your application should pass the same record on all subsequent Btrieve operations for the file. It should never change the value of the position block. When an application has more than one file open at a time, Btrieve uses the position block to determine which file a particular call is for.

DATA BUFFER

The data buffer contains the records that your application transfers to and from the Btrieve file. Ensure that you allocate a large enough data buffer to accommodate the longest record in your file. If the buffer is too short, Btrieve requests may destroy data items following the data buffer.

DATA BUFFER LENGTH

For any operation that requires a data buffer, your program must pass the length of the data buffer as a 2-byte integer (COMP-0 field for IBM or Microsoft COBOL). For a file with fixed length records, this parameter should match the record length specified when you first created the file.

When you are inserting records into or updating a file with variable length records, this parameter should equal the record length specified when you first created the file, plus the number of characters included beyond the fixed length portion. When you are retrieving variable length records, this parameter should be large enough to accommodate the longest record in the file.

KEY BUFFER

On each Btrieve call, your application must pass a record variable to contain the key value. If the key consists of two or more segments, list them in the correct order as individual fields under an 01 level record. Then you can pass the entire record to Btrieve as the key buffer. Depending on the operation, your application may set this variable or the Btrieve Record Manager may return it.

Btrieve cannot determine the key buffer length when you call it from an IBM (or Microsoft) COBOL program. Therefore, you must ensure that the buffer is at least as long as the key length you specified when you first created the file. Otherwise, Btrieve requests may destroy data items following the key buffer.

KEY NUMBER

You can define up to 24 different keys when you create a Btrieve file. The application accessing the file must tell the Record Manager which access path to follow for a particular operation. The key number parameter is a 2-byte integer (COMP-0 for IBM or Microsoft COBOL) with a value from 0 through 23. The Btrieve Record Manager never alters this parameter.

PARAMETER LIST EXAMPLE

The code for an IBM (or Microsoft) COBOL application in Figure 4.5 below opens a Btrieve file and retrieves the data record corresponding to the first value for key 0, the name field.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EX1.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  B-OPEN              PIC 99 COMP-0 VALUE 0.
77  KEY-NUMBER          PIC 99 COMP-0 VALUE 0.
77  B-STATUS            PIC 99 COMP-0.
01  B-GET-FIRST         PIC 99999 COMP-0 VALUE 12.
01  DSP-STATUS          PIC 99999.
01  DATA-BUFFER.
02  NAME                PIC X(30).
02  STREET              PIC X(30).
    02 CITY             PIC X(30).
    02 STATE            PIC XX.
    02 ZIP              PIC 9(5).
01  BUF-LEN             PIC 99 COMP-0 VALUE 97.
01  FILE-NAME           PIC X(14) VALUE "ADDRESS.BTR  ".
01  KEY-BUFFER          PIC X(30).
01  POSITION-BLOCK      PIC X(128) VALUE SPACES.

PROCEDURE DIVISION.
    CALL 'BTRV' USING B-OPEN, B-STATUS, POSITION-BLOCK,
       DATA-BUFFER,BUF-LEN, FILE-NAME, KEY-NUMBER.
    IF B-STATUS NOT = 0
      MOVE B-STATUS TO DSP-STATUS
      DISPLAY "Error opening file. Status = " DSP-STATUS
      STOP RUN.
    DISPLAY (1, 1) ERASE.
    CALL 'BTRV' USING B-GET-FIRST, B-STATUS, POSITION-BLOCK,
       DATA-BUFFER, BUF-LEN, KEY-BUFFER, KEY-NUMBER.
    IF B-STATUS NOT = 0
      MOVE B-STATUS TO DSP-STATUS
      DISPLAY (5, 1) "Error reading file. Status =" DSP-STATUS
    ELSE
      DISPLAY (5, 1) "First record in file is:" DATA-BUFFER.
    STOP RUN.
```

Figure 5.5
Btrieve Call from IBM COBOL

INTERFACING BTRIEVE WITH C

The Btrieve diskette contains interfaces for Microsoft and Lattice C as well as several other C compilers. The format for the Btrieve calls is identical for each. To interface with any other C compiler, refer to Appendix F, which describes how to interface to Btrieve from assembly language.

Your Btrieve program diskette contains the source code for each of these interfaces. All the C interfaces are written in C.

To access a Btrieve file, your application must call the integer function, BTRV. Btrieve provides a small interface routine that links the Btrieve Record Manager with your C application. You must load the Record Manager before you start your application.

LINKING A C APPLICATION WITH BTRIEVE

After you have successfully compiled your C program, link it with the C interface. The method used to link the C interface varies slightly depending on which C compiler you are using. For a complete explanation of linking, refer to your operating system manual and the reference manual for your compiler.

If you are using the Microsoft C compiler, you should compile the interface in the file MSCXBTRV.C using your compiler. If you are compiling with the large model, you should edit the interface source file and make the changes described in that document. Link your application as shown in the following example:

```
    Object Modules [.OBJ]: c+cprog+mscxbtrv
```

CALLING BTRIEVE FROM C

Your application should never perform any standard C I/O against a Btrieve file. It should handle I/O through a call to Btrieve. After issuing an Open operation, your program can read, write, and modify files through Btrieve calls. Before terminating, your application should perform a Btrieve Close operation.

Your application issues calls to Btrieve through the integer function, BTRV. The result of the function is always an integer value that corresponds to one of the status codes listed in Appendix B. After a Btrieve call, your application should always check the value of the status variable. A status of 0 indicates a successful operation. Your application should be able to recognize and resolve a non-zero status.

The function, BTRV, expects parameters of the following types:

```
int BTRV (OP, POS_BLK, DATA_BUF, BUF_LEN, KEY_BUF, KEY_NUM)
          int     OP;                               /* operation code */
          char    POS_BLK[];                        /* position block */
          char    DATA_BUF[];                          /* data buffer */
          int     *BUF_LEN;                  /* length of data buffer */
          char    KEY_BUF[];                            /* key buffer */
          int     KEY_NUM;                              /* key number*/
```

Although you must provide all parameters on every call, Btrieve does not use every parameter to perform every operation. In some cases, Btrieve ignores their value. See Chapter 6 for a more detailed description of which parameters are relevant for each operation. The following sections describe each parameter.

OPERATION

The operation parameter determines which type of Btrieve function you wish to perform. The operation may be a read, write, delete, or update. Your application must specify a valid operation code on every Btrieve call. The Btrieve Record Manager never changes the code. The variable you specify must be an integer type and can be any one of the legal Btrieve operation codes described in Chapter 6. Also see Appendix A for a complete list of these codes.

POSITION BLOCK

A C application allocates a 128-byte character array that Btrieve will use to store the file I/O structures and the positioning information described in Chapter 2. Btrieve initializes this array when your application performs the Open operation. Btrieve references and updates the data in this array for all file operations. Therefore, your application should pass the same array on all subsequent Btrieve operations for the file. It should not change the value stored in this array at any time.

When an application has more than one file open at a time, Btrieve uses the position block to determine which file a particular call is for. Your application must allocate a separate position block for each Btrieve file it opens.

DATA BUFFER

The data buffer parameter is the address of an array or structure-type variable containing the records that your application transfers to and from the Btrieve file. Ensure that you allocate a large enough data buffer to accommodate the longest record in your file. If the buffer is too short, Btrieve requests may destroy data items following the data buffer.

DATA BUFFER LENGTH

For any operation that requires a data buffer, your program must pass the address of an integer variable that contains the length of the data buffer. For a file with fixed length records, this parameter should match the record length specified when you first created the file.

When you are inserting records into or updating a file with variable length records, this parameter should equal the record length specified when you first created the file, plus the number of characters included beyond the fixed length portion. When you are retrieving variable length records, this parameter should be large enough to accommodate the longest record in the file.

KEY BUFFER

Your application must pass the address of a variable containing the key value on every Btrieve call. If you defined the key as binary when you first created the file, you should define the variable as type int, long, or unsigned. If you originally defined the key as a string, you should define the key buffer variable as a structure or a character array. If the key consists of two or more segments, use a structure variable that contains the segment fields in the correct order. Depending on the operation, your application may set this variable or the Btrieve Record Manager may return it.

Btrieve cannot determine the key buffer length when called from a C program. Therefore, you must ensure that the buffer is at least as long as the key length you specified when you first created the file. Otherwise, Btrieve requests may destroy data items stored in memory following the key buffer.

KEY NUMBER

You can define up to 24 different keys when you create a Btrieve file. Therefore, the application accessing the file must tell the Record Manager which access path to follow for a particular operation. The key number parameter is an integer variable with a value from 0 through 23. The Btrieve Record Manager never alters this parameter.

PARAMETER LIST EXAMPLE

The C code in Figure 4.6 opens a Btrieve file and retrieves the data record corresponding to the first value for key 0, the name field.

```
#define B_OPEN  0
#define B_FIRST 12

main (){
struct ADDR_REC                                   /* Structure of address file record */
     {
        char  NAME[30];
        char  STREET[30];
        char  CITY[30];
        char  STATE[2];
        char  ZIP[5];
     };

struct ADDR_REC  ADDR_BUF;
int      DB_LEN;
char     KEY_BUF[30];
char     POS_BLK[128];
int      STATUS;
STATUS = BTRV (B_OPEN, POS_BLK, &ADDR_BUF, &DB_LEN, "ADDRESS.BTR", 0);
if (STATUS != 0)
     {
        printf ("Error opening file. Status = %d", STATUS);
        exit (0);
     }
DB_LEN = sizeof (ADDR_BUF);
STATUS = BTRV (B_FIRST, POS_BLK, &ADDR_BUF, &DB_LEN, KEY_BUF, 0);
if (STATUS != 0)
     printf ("Error reading file. Status = %d", STATUS);
else
     printf ("First record in file is: %.97s", &ADDR_BUF);
};
```

Figure 5.6
Btrieve Call from C

INTERFACING WITH BTRIEVE IN ASSEMBLY LANGUAGE

If you are using Assembly Language or a programming language for which no interface is provided on the Btrieve diskette, you can write an interface using Assembly Language. Interfacing Btrieve with an Assembly Language routine requires three basic steps:

- Storing the Btrieve parameters in memory in the format expected by the Record Manager
- Verifying that the Record Manager has been loaded into memory
- Calling the Record Manager by performing an interrupt that transfers control to Btrieve

STORING THE PARAMETERS

Previous sections in this chapter have covered only seven Btrieve parameters: status, operation code, position block, data buffer, data buffer length, key buffer, and key number. (The BASIC interface combines position block and data buffer into the single parameter, FCB. The Pascal and C interfaces return status as the value of a function.)

Actually, the Record Manager expects ten different parameters. The language interface routines provided by Btrieve derive three of the parameters before transferring control to the Record Manager.To write your own Assembly Language interface to Btrieve, you must initialize all ten parameters. The following figure illustrates the ten Btrieve parameters and their format.

Parameter #	Offset	Contents	
1	0	data buffer offset	
		data buffer segment	
2	4	data buffer length	
3	6	positioning information offset	
		positioning information segment	
4	10	FCB offset	
		FCB segment	
5	14	operation code	
6	16	key buffer offset	
		key buffer segment	
7, 8	20	key length	key number
9	22	status offset	
		status segment	
10	26	interface ID	

Figure 5.7
Btrieve Parameters Structure

Before you can call Btrieve, you must initialize a data area containing all ten parameters listed in Figure 5.1. Store the offset of this data area in the DX register. Btrieve expects the address of the parameter block to be in DS:DX when it assumes control.

PARAMETER DESCRIPTIONS

The following sections describe each of the ten parameters expected by Btrieve when you use an Assembly Language interface routine.

DATA BUFFER

Pass the data buffer as a double word containing the segment address and offset of the application's data buffer. The data buffer is the area used to transfer records between the application and the Record Manager.

DATA BUFFER LENGTH

Pass the data buffer length as a word containing the length of the data buffer. The application passes to the interface the address of the integer containing the data buffer length. The interface must copy the value of the data buffer length to the parameter block it will pass to Btrieve. After the Btrieve interrupt returns, the interface must copy the data buffer length value from the parameter block back to the address specified by the application.

POSITIONING INFORMATION

Pass the positioning information as a double word containing the segment address and offset of a 90-byte data area that Btrieve uses for storing positioning information about the file. This is one of the parameters derived by the Btrieve application interfaces. In BASIC, this area comes from within the BASIC FCB. In the other languages, part of the position block parameter is used for this purpose. You must provide the address of a 90-byte area for the Record Manager to use when calling Btrieve from Assembly Language. The same data area should be passed to Btrieve on all calls for the same file.

FCB

Pass the FCB as a double word containing the segment address and offset of a 38-byte data area that Btrieve uses for storing a DOS FCB. This is one of the parameters derived by the Btrieve application interfaces. In BASIC, this area comes from within the BASIC FCB. In other languages, part of the position block parameter is used for this purpose. You must provide the address of a 38-byte area for the Record Manager to use when calling Btrieve from Assembly Language. The same FCB should be passed to Btrieve on all calls for the same file.

OPERATION CODE

Pass the operation code as a word containing the Btrieve operation code. This must be one of the Btrieve operation codes listed in Appendix A.

KEY BUFFER

Pass the key buffer as a double word containing the segment address and offset of the application's key buffer.

KEY LENGTH

Pass the key length as a byte containing the length of the key buffer. This is one of the parameters derived by the Btrieve application interfaces that you must provide when calling Btrieve from Assembly Language.

The interfaces to Btrieve from COBOL and C cannot determine the length of the key buffer since it is not passed with the data type by the compiler. In these cases, Btrieve passes the maximum key length allowed. The Record Manager will never write to the key buffer beyond the key's defined length. The Record Manager will return an error if this parameter is shorter than the key's defined length.

KEY NUMBER

Pass the key number as a byte containing the key number by which the file is to be accessed.

STATUS CODE

Pass the status code as a double word containing the segment address and offset of the status parameter. This is the address at which Btrieve will store the status code after the operation is completed.

INTERFACE ID

Pass the interface ID as a word initialized to 06176H. If you set the identifier to any other value, Btrieve will not permit access to files with variable length records. This check allows Btrieve to determine if it is called from applications linked with previous versions of the interface that do not understand variable length records.

VERIFY THAT THE RECORD MANAGER IS LOADED

Once your parameters are initialized, verify that the Record Manager is loaded before you try to call it. When the Record Manager is loaded, it stores its entry point in interrupt vector 07BH. Be sure that the word at interrupt vector 07BH is initialized to 033H so that the interface can determine that the Record Manager is loaded.

CALLING THE RECORD MANAGER

After you have stored the address of the Btrieve parameters in DX and have checked to be sure that the Record Manager is loaded, you are ready to call Btrieve. Perform an interrupt 07BH, and the Record Manager will process your request. When the operation is completed, the Record Manager returns control to your code at the instruction following the interrupt. The following example illustrates the code that checks whether the Record Manager is loaded and then performs the interrupt.

```
BTR_ERR          EQU 20
BTR_VECTOR       EQU 07BH*4
    PUSH DS
    SUB  BX,BX                                          ; Clear BX
    MOV  DS,BX                                    ; DS => absolute 0
    CMP  WORD PTR BTR_VECTOR[BX],033H  ; Has Btrieve been initialized?
    POP  DS
    JE   DO_INT                            ; Yes – go perform interrupt
    MOV  STAT,BTR_ERR                            ; No – set status 20
    JMP  OUT                                   ; and return to caller
DO_INT:
    INT  07BH                                ; Call the Record Manager
```

OS/2 INTERFACES

NetWare Btrieve includes two interfaces for the C compiler for OS/2 (C2XBTRV.C and C2FXBTRV.C) that you can include with applications that run at OS/2 workstations. The C2XBTRV.C interface is for use with protected mode applications. The C2FXBTRV.C interface is for use with OS/2 FAPI programs.

If you are using a language that can call an Assembly Language routine, you can write an Assembly Language interface to Btrieve, using the source code from either interface as a guide. The section below contains guidelines for using the C interfaces and for writing your own Assembly Language interface.

C LANGUAGE

If you are using the C compiler supplied with OS/2, you have two options:

- You can compile either the C2XBTRV.C or C2FXBTRV.C interface separately from your program. Either include the resulting object module when you link your programs, or install the object in a library file that you include with your link.
- You can include the interface source contained in either C2XBTRV.C or C2FXBTRV.C with your application's source code when you compile your program.

ASSEMBLY LANGUAGE

If you are using Assembly Language, your application must use the OS/2 dynamic link mechanism to access the Btrieve routines. Use the following guidelines when you write an Assembly Language interface for OS/2:

- Pass all parameters on the stack.
- Use the AX register to receive the return code from Btrieve.
- Use the form *selector:offset* for all interface addresses.
- Access the dynamic link routine using a FAR CALL.
- Do not pop the return parameters off the stack.
- Specify the dynamic link routine name in upper case.

LINKING OS/2 APPLICATIONS

The external references to the dynamic link routines are resolved the same way in both C and Assembly Language.

You must provide the BTRCALLS.LIB library to the linker when you link your object files. This library contains the dynamic link definition records that provide the correspondence between the called routine and the BTRCALLS.DLL file. If the linker does not have access to BTRCALLS.LIB, it reports unresolved linkages for the dynamic link entry points.

NOTE:
For Family applications, you must use the C2FXBTRV.C interface, link your application, and then run the OS/2 BIND utility. The BUTIL.EXE program is an example of a FAPI application.

INTERFACING WITH BROUTER

If you are writing a VAP that uses Btrieve files, you must include an interface to BROUTER in your code. Follow the procedures described under "Interfacing With Btrieve In Assembly Language," beginning on page 5-26, with the following additions:

- Issue the NetWare "Get Interrupt Vector" function call to determine whether BROUTER is loaded. If the vector for the 7B interrupt is zero, BROUTER is not loaded.
- Store a unique client ID number in the AX register prior to performing the 7B interrupt.
- Identify every workstation that accesses your VAP with a unique 2-byte client ID number. You must provide this number to BROUTER for concurrency and transaction processing purposes. A convenient method for generating a client ID number is storing the workstation's connection number in the first byte, and a unique binary value in the second byte.
- Identify your VAP to BROUTER by using a unique 2-character, ASCII identifier in the BX register. This identifier distinguishes your VAP from any other VAP that accesses BROUTER. Contact the Novell Development Products Division to obtain the identifer for your VAP. The address of the Novell Development Products Division is:

  Novell Development Products Division
  6034 West Courtyard
  Suite 220
  Austin, TX 78730

In general, your interface to BROUTER must complete all of the following steps:

- Check to see if BROUTER is loaded by executing the NetWare "Get Interrupt Vector" function call.
- Store the Btrieve parameter block in memory in the format expected by the Record Manager.
- Store the address of the Btrieve parameter block in the DX register.
- Store a client ID number that is unique for each workstation in the AX register.
- Store the unique 2-character, ASCII identifier for the VAP in the BX register.
- Perform the 7B interrupt that transfers control from your VAP to BROUTER.

6 BTRIEVE RECORD OPERATIONS

This chapter describes each of the 36 operations your application can perform using Btrieve. For each operation, this chapter presents the following information:

- The purpose of the operation
- A table illustrating the parameter values that Btrieve expects from and returns to your application
- A description of what the operation does
- The prerequisites your application must satisfy before the operation will be successful
- The procedure for initializing the parameters required by the operation
- The results of both a successful and an unsuccessful operation
- The effect the operation has on your current position in the file

The parameter table lists six Btrieve parameters, including the position block and the data buffer. In BASIC, these two parameters are consolidated into a single FCB parameter in the BASIC interface. On every Btrieve call, your application must pass to Btrieve every parameter required for the language you are using, even when Btrieve does not expect a value for or return a value to that parameter.

The status parameter does not appear in the table since Btrieve handles it the same way for all operations. Btrieve doesn't expect your application to initialize the status before a call and always returns a status value to your application.

Program examples that illustrate each operation for Pascal, COBOL, C, and BASIC are included in Appendixes C, D, E, and F.

ABORT TRANSACTION (21)

Purpose:

Abort Transaction removes all operations performed since the beginning of an active transaction and terminates the transaction.

Parameter Usage:

	Operation	FCB Pos Block	FCB Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X					
Returned						

Description:

Your application can perform an Abort Transaction operation to terminate a transaction that has been interrupted. Abort Transaction removes all operations performed after the start of the previous Begin Transaction operation, and terminates the current transaction.

Prerequisites:

Before your application issues an Abort Transaction operation, the following prerequisites must be met:

- You must have specified a valid transaction control file (using the /T: start-up option) when you loaded Btrieve.
- You must have issued a successful Begin Transaction operation prior to issuing the Abort Transaction operation.

Procedure:

To perform an Abort Transaction operation, set the operation code to 21 before issuing the Btrieve call. Btrieve will ignore all other parameters on an Abort Transaction call.

ABORT TRANSACTION (21)
(Continued)

Result:

If the Abort Transaction operation is successful, Btrieve will return a status code of 0. All Insert (2), Update (3), and Delete (4) operations issued since the beginning of the transaction will be removed from the files.

If the Abort Transaction operation fails, Btrieve will return a non-zero status code indicating the reason. Common non-zero status codes for this operation include the following:

- 36 Not Configured For Transactions
- 39 No Begin Transaction

Current Positioning:

An Abort Transaction operation has no effect on positioning.

BEGIN TRANSACTION (19)

Purpose:

The Begin Transaction operation marks the beginning of a set of logically related Btrieve operations.

Parameter Usage:

	Operation	FCB		Data Buffer Length	Key Buffer	Key Number
		Pos Block	Data Buffer			
Expected	X					
Returned						

Description:

A Begin Transaction operation defines the start of a transaction. Transactions are useful when you need to perform multiple Btrieve operations to record a single event and your database would be inconsistent if all those operations were not completed. A transaction can include any number of Btrieve operations on as many as twelve different files. By enclosing a set of operations between Begin and End Transaction operations, you can ensure that Btrieve does not permanently record any of the intervening operations unless all of those operations are successfully completed.

Prerequisites:

Before your application issues a Begin Transaction operation, the following prerequisites must be met:

- You must have specified a valid transaction control file when you configured BSERVER.
- Your application must have ended or aborted any previous transaction.

BEGIN TRANSACTION (19)
(Continued)

Procedure:

To perform a Begin Transaction, set the operation code to 19 before calling Btrieve. Btrieve will ignore all other parameters on a Begin Transaction call.

Result:

If the Begin Transaction operation is successful, Btrieve will return a status code of 0.

If the operation fails for any reason, Btrieve will return a non-zero status code. The most common non-zero status codes for this operation are the following:

- 36 Not Configured for Transactions
- 37 Transaction Is Active

Current Positioning:

A Begin Transaction operation has no effect on positioning.

CLEAR OWNER (30)

Purpose:

The Clear Owner operation removes the owner name associated with a Btrieve file.

Parameter Usage:

		FCB				
	Operation	Pos Block	Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X				
Returned		X				

Description:

Clear Owner removes an owner name that you have assigned to a file with the Set Owner operation. If the data was previously encrypted, Btrieve decrypts the data during a Clear Owner operation.

Prerequisites:

Before your application issues a Clear Owner operation, the following prerequisites must be met:

- The Btrieve file must be open.
- An owner name must be assigned to the file.
- No transactions can be active.

CLEAR OWNER (30)
(Continued)

Procedure:

To perform a Clear Owner operation, set the Btrieve parameters as follows:

- Set the operation code to 30.
- Pass the position block that identifies the file you want to clear.

Result:

After a Clear Owner operation, Btrieve no longer requires the owner name when you attempt to open a file. If you have previously encrypted the data in the Btrieve file when you assigned the owner, Btrieve decrypts the data during a Clear Owner operation. The more data that Btrieve must decrypt, the longer the Clear Owner operation takes.

The most common non-zero status codes for this operation are the following:

3	File Not Open
41	Operation Not Allowed

Current Positioning:

Clear Owner has no effect on positioning.

CLOSE (1)

Purpose:

The Close operation closes a Btrieve file.

Parameter Usage:

		FCB				
	Operation	Pos Block	Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X				
Returned						

Description:

When an application has finished accessing a Btrieve file, it should perform a Close operation. This operation closes the file associated with the specified position block and releases any lock the application has executed for the file. After a Close operation, your application cannot access the file again until it issues another Open operation for that file.

Prerequisites:

Before your application issues a Close operation, the following prerequisites must be met:

- The file must be open.
- Any transactions must be ended or aborted.

Procedure:

To perform this operation, set the Btrieve parameters as follows:

- Set the operation code to 1.
- Pass a valid position block for the file you want to close.

CLOSE (1)

(Continued)

Result:

If the Close operation is successful, the following will occur:

- The position block for the closed file will no longer be valid. Your application can use it for another file, or can use the data area for other purposes.
- Any pre-image file associated with the closed data file will be deleted if no other workstations have the Btrieve file open.

If the Close operation fails, the file will remain open. The most common non-zero status code for this operation is status 3 (File Not Open).

Current Positioning:

Close removes any positioning information associated with the file.

CREATE (14)

Purpose:

The Create operation creates a Btrieve file with a specified set of characteristics.

Parameter Usage:

	Operation	FCB		Data Buffer Length	Key Buffer	Key Number
		Pos Block	Data Buffer			
Expected	X	X	X	X	X	X
Returned		X		X		

Description:

The Create operation allows you to generate a Btrieve file from within your application program. It performs essentially the same function as the CREATE utility described in Chapter 3. See Chapters 1 and 2 of this manual for more information about the file and key characteristics you need to specify when you create a file.

The following sections describe how to store the definition of the Btrieve file in the data buffer. A chart on the next page illustrates the order in which the different specifications for the file and the keys should be stored. The sections following the chart describe how to specify the following:

- The file specifications
- The key characteristics
- The alternate collating sequence
- The data buffer length

CREATE (14)

(Continued)

Btrieve expects the data buffer to be formatted as shown in Table 6.1.

	Description	Length
file specs	record length	2
	page size	2
	# of indexes	2
	not used	4
	file flags	2
	reserved word	2
	allocation	2

	Description	Length
key specs (repeated)	key position	2
	key length	2
	key flags	2
	not used	4
	extended key type	1
	null value	1
	reserved	4

Table 6.1
Data buffer structure for Create operation

File Specifications. Store the file specifications in the first 16 bytes of the data buffer. The bytes are numbered beginning with 0. Store the information for the record length, page size, and number of indexes as integers. To create a data-only file, set the number of indexes to zero.

You must allocate the "not used" and "reserved" areas of the data buffer even though Btrieve doesn't use them for a Create operation. Initialize the reserved areas to 0 to maintain compatibility with future releases of Btrieve.

CREATE (14)
(Continued)

The bit setting in the file flags word specifies whether a file allows variable length records, blank truncation, or data compression, and whether Btrieve should preallocate disk space for the file. Use the two high bits of the low order byte to specify the free space threshold for variable length pages.

Bits in the file flags word are numbered from 0 to 15, with 0 being the low order bit. Set the bits according to the following description:

- If bit 0 = 1, Btrieve will allow the file to contain variable length records.
- If bit 1 = 1, Btrieve will truncate trailing blanks in variable length records.
- If bit 2 = 1, Btrieve will preallocate the number of pages you specify in the allocation word.
- If bit 3 = 1, Btrieve will compress the data in the file.
- If bit 4 = 1, Btrieve will create the file as a key-only file.
- If bit 6 = 1, Btrieve will maintain a10% free space threshold on the variable length pages.
- If bit 7 = 1, Btrieve will maintain a 20% free space threshold on the variable length pages.
- If bit 6 = 1 and bit 7 = 1, Btrieve will maintain a 30% free space threshold on the variable length pages.

CREATE (14)
(Continued)

The following table shows the binary and decimal representations of the file flag values:

Values	Binary	Decimal
variable length	00000001	1
blank truncation	00000010	2
preallocation	00000100	4
data compression	00001000	8
key-only	00010000	16
10% free space	01000000	64
20% free space	10000000	128
30% free space	11000000	192

If you need to specify a combination of the file attributes, add their respective flag values. For example, to specify a file that allows variable length records and uses blank truncation, initialize the file flags to 3 (2 + 1). Btrieve ignores the blank truncation and free space threshold flags if the variable length flag is set to 0.

If you set the preallocation flag bit, use the allocation word to store an integer value that specifies the number of pages you want to preallocate to the file.

Key Characteristics. Place the key characteristics immediately after the file specification block. Allocate a 16-byte key specification block for each key segment in the file. The extended key type code and the null character are each 1 byte long.

Store the information for the key position and the key length as integers.

CREATE (14)
(Continued)

Set the key flags according to the following description to specify the attributes you want a key to have.

- If bit 0 = 1, the key allows duplicates.
- If bit 1 = 1, the key is modifiable.
- If bit 2 = 0 and bit 8 = 0, the key is string.
- If bit 2 = 1 and bit 8 = 0, the key is binary.
- If bit 3 = 1, the key has a null value.
- If bit 4 = 1, the key has another segment.
- If bit 5 = 1, the key is sorted by an alternate collating sequence.
- If bit 6 = 1, the key is sorted in descending order.
- Bit 7 is ignored for a Create operation.
- If bit 8 = 0, the key is a standard type.
- If bit 8 = 1, the key is an extended type.
- If bit 9 = 1, the key is manual.

Although Btrieve ignores bit 7 for a Create operation, you should initialize it to 0 when you create the file. When you issue a Stat operation (15) Btrieve will set bit 7 to 1 if the key is a supplemental index, and returns the key flags in the data buffer.

CREATE (14)

(Continued)

The following table shows the binary, hexadecimal, and decimal values for the key flags:

Attribute	Binary	Hexadecimal	Decimal
duplicate	00000001	01	1
modifiable	00000010	02	2
binary	00000100	04	4
null	00001000	08	8
segmented	00010000	10	16
alt col seq	00100000	20	32
descending	01000000	40	64
supplemental	10000000	80	128
extended type	1 00000000	100	256
manual	10 00000000	200	512

Assign the same duplicate, modifiable, and null attributes for all segments of the same key. If you specify the null attribute for the key, you may assign different null characters for individual segments.

The segmented key attribute is a flag indicating that the next key block in the data buffer refers to the next segment of the same key. In addition, you can make each key segment either ascending or descending, and specify any data type.

For example, to create a file with two keys, the first consisting of two segments and the second consisting of one segment, use bit 4 of the key flags as follows:

- In the first key block, set bit 4 of the key flags word to 1, indicating that another segment definition for that key follows.
- In the second key block, set bit 4 of the key flags word to 0, indicating that this key block defines the last segment of the first key.
- In the third key block, set bit 4 of the key flags word to 0, indicating that the second key has only one segment.

CREATE (14)
(Continued)

Specify the extended key type as a binary value in byte 10 of the key specification block. The values for the extended key types are shown below:

Type	Value
string	0
integer	1
float	2
date	3
time	4
decimal	5
money	6
logical	7
numeric	8
bfloat	9
lstring	10
zstring	11
unsigned binary	14
autoincrement	15

As with the file flags, you can specify a combination of key attributes by adding their respective flag values. For example, if the key is an extended type, is part of a segmented key, and is to be collated in descending order, you would store 150H (336 decimal) in the flags word.

NOTE:

You can define the string and unsigned binary extended key types as either standard types or extended types. This maintains compatibility with applications that were developed under earlier versions of Btrieve, while allowing new applications to use extended key types exclusively.

CREATE (14)

(Continued)

Alternate Collating Sequence. You may specify an alternate collating sequence for sorting any number of key segments in the file. However, you may specify only *one* alternate collating sequence for the entire file. You may designate that some segments of a single key are to be sorted with the standard ASCII collating sequence and that other segments are to be sorted with the alternate sequence.

You can specify an alternate collating sequence for the lstring, zstring, and string key types. If you do set the alternate collating sequence flag for any key or key segment in the file, place the definition for the collating sequence immediately after the last key specification block.

That is, the actual collating sequence itself must follow the key specification block instead of the name of a file containing that sequence. The alternate collating sequence definition consists of nine header bytes followed by 256 characters as described under "Alternate Collating Sequence Files" on page 4-15.

NOTE:

If you create multiple files with different alternate collating sequences, use a different name for each sequence.

CREATE (14)

(Continued)

Data Buffer Length. The data buffer length must be long enough to include the file specifications, the key characteristics, and an alternate collating sequence, if one is defined. Do <u>not</u> specify the file's record length in this parameter.

For example, to create a file that has two keys of one segment each and an alternate collating sequence, the data buffer for the Create operation should be at least 313 bytes long, as shown below.

File Spec	+	Key1 Spec	+	Key2 Spec	+	Alt Col		
16	+	16	+	16	+	265	=	313

Key Number. You can use the key number parameter to specify whether you want Btrieve to warn you if a file of the same name already exists. Specify a value for the key number as follows:

- *If you do not want Btrieve to create a new file over an existing file,* set the key number parameter to –1. If a file of the same name already exists, Btrieve will return a nonzero status and will not create a new file.

- *If you want Btrieve to create a new file over an existing file, or if you do not want to check for the presence of an existing file,* set the key number parameter to a non-negative value, preferably 0.

CREATE (14)
(Continued)

Prerequisites:

If you are creating an empty Btrieve file over a pre-existing Btrieve file, be sure the file is closed before executing the Create operation.

Procedure:

To perform the Create operation, set the Btrieve parameters as follows:

- Set the operation code to 14.
- Specify the file specifications, key characteristics, and any alternate collating sequence in the data buffer. All of the values for the file specifications and key characteristics you store in the data buffer must be in binary format.
- Specify the length of the data buffer.
- Set the key number parameter to –1 if you want Btrieve to warn you that a file of the same name already exists. Otherwise, set the key number parameter to 0.
- Specify the name for the file in the key buffer. Be sure to terminate the filename with a blank or a binary zero. You may specify the device name and path name for the file, including any number of directory levels.

CREATE (14)
(Continued)

Result:

If the operation is successful, Btrieve will either warn you of the existence of a file with the same name, or will create the new file according to your specifications. The new file will not contain any records. The Create operation does not open the file. Your application must perform an Open operation before it can access the file.

If the operation is unsuccessful, Btrieve will return a non-zero status code informing you of the reason. Common non-zero codes include the following:

• 2	File I/O Error
• 22	Data Buffer Too Short
• 24	Page Size Error
• 25	Create I/O Error
• 26	Number of Keys
• 27	Invalid Key Position
• 28	Invalid Record Length
• 29	Invalid Key Length
• 48	Invalid Alternate Sequence Definition
• 49	Key Type Error
• 59	File Already Exists

Refer to Appendix B for an explanation of these status codes.

Current Positioning:

A Create operation does not establish any positioning information.

CREATE SUPPLEMENTAL INDEX (31)

Purpose:

The Create Supplemental Index operation adds a supplemental index to an existing Btrieve file.

Parameter Usage:

		FCB				
	Operation	Pos Block	Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X	X	X		
Returned						

Description:

Use the Create Supplemental Index operation to add an index to a file at any time after the file has been created.

Prerequisites:

Before your application issues a Create Supplemental Index operation, the following prerequisites must be met:

- The Btrieve file must be open.
- The number of existing key segments in the file must be less than or equal to the following formula:

  24 – (# of segments to be added)

- Be certain that the key flags, the position, and the length of the new index are appropriate for the file to which you are adding the index.
- No transactions can be active .

CREATE SUPPLEMENTAL INDEX (31)
(Continued)

Procedure:

To create a supplemental index, set the Btrieve parameters as follows:

- Set the operation code to 31.
- Pass Btrieve the position block for the file to which you want to add the index.
- Store the key specifications for the new index in the data buffer. The data buffer consists of a 16-byte key specification block for each segment of the supplemental index you are creating. Use the same structure as the key specification block you use in the Create operation (14).
- Set the data buffer length parameter to the number of bytes in the data buffer. For a new index with no alternate collating sequence, use the following formula to determine the correct data buffer length:

  16 * (# of segments)

  If the new key has an alternate collating sequence, use the following formula to determine the correct data buffer length:

  16 * (# of segments) +265

Result:

Btrieve will immediately begin to add the new index to the file. The time required for this operation depends on the total number of records to be indexed, the size of the file, and the length of the new index.

The key number of the new index is one higher than the previous highest key number. You can use the new index to access your data as soon as the operation completes.

CREATE SUPPLEMENTAL INDEX (31)
(Continued)

If Btrieve is unable to create the supplemental index for any reason, it will return a non-zero status indicating the reason and drop the portion of the supplemental index which it has already built. The file pages allocated to the supplemental index prior to the error will be placed on a list of free space for the file and re-used when you insert records or create another supplemental index.

Common errors include the following:

- 22 Data Buffer Too Short
- 27 Invalid Key Position
- 41 Operation Not Allowed
- 45 Inconsistent Key Flags
- 49 Key Type Error
- 56 Incomplete Index

If a power failure or system reset occurs during the creation of a supplemental index, you can access the data in the file through the file's other indexes. However, Btrieve will return a non-zero status if you try to access data by the incomplete index. In this case, drop the incomplete index with a Drop Supplemental Index operation (32) and re-issue the Create Supplemental Index operation.

Current Positioning:

The Create Supplemental Index operation has no effect on positioning.

DELETE (4)

Purpose:

Delete removes an existing record from a Btrieve file.

Parameter Usage:

		FCB				
	Operation	Pos Block	Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X		x		X
Returned		X		X		

Description:

You can remove an existing record from a file using the Delete operation. After deletion the space in the file where the deleted record was stored is placed on a list of free space.

Prerequisites:

Before your application issues a Delete operation, the following prerequisites must be met:

- The data file must be open.
- You cannot issue a call to the file containing the record to be deleted between the retrieval of the record and its deletion.

Procedure:

To perform the Delete operation, set the Btrieve parameters as follows:

- Specify an operation code of 4.
- Pass Btrieve the position block of the file from which the record is to be deleted.

DELETE (4)

(Continued)

- Initialize the data buffer length parameter to the length of the record to be deleted.
- Store the key number used to retrieve the record in the key number parameter.

Result:

If the Delete operation is successful, Btrieve will

- Completely remove the record from the file;
- Adjust all key indexes to reflect the deletion;
- Set the data buffer length to the length of the deleted record.

If it is unable to successfully delete the record, Btrieve will return a non-zero status code. Common error codes include the following:

- 7 Different Key Number
- 8 Invalid Positioning

Current Positioning:

After a Delete operation, Btrieve erases any existing positioning information and establishes its position in the file as follows:

- If a duplicate exists, the **next** record becomes the first duplicate following the deleted record. Otherwise, the next record becomes the first record for the key value greater than the deleted key value.
- If a duplicate exists, the **previous** record becomes the previous duplicate of the key value. Otherwise, the previous record becomes the last data record for the previous key value.

DROP SUPPLEMENTAL INDEX (32)

Purpose:

The Drop Supplemental Index operation removes a supplemental index from an existing Btrieve file.

Parameter Usage:

		FCB				
	Operation	Pos Block	Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X				X
Returned						

Description:

Use the Drop Supplemental Index operation to remove a supplemental index from a file.

Prerequisites:

Before your application issues a Drop Supplemental Index operation, the following prerequisites must be met:

- The file must be open.
- A supplemental index must exist in the file.
- No transactions can be active.

DROP SUPPLEMENTAL INDEX (32)

(Continued)

Procedure:

To drop a supplemental index, set the Btrieve parameters as follows:

- Set the operation code to 32.
- Pass the position block of the Btrieve file.
- Store the key number for the supplemental index you want to drop in the key number parameter.

Result:

If the operation is successful, Btrieve will

- Place the file pages that were allocated to that index on a list of free space for later use;
- Decrement (reduce by one) the key numbers of any other supplemental indexes with key numbers higher than the index that was dropped.

If the operation is unsuccessful, Btrieve will return a non-zero status to your application. Common non-zero status codes for this operation include the following:

- 6 Invalid Key Number
- 41 Operation Not Allowed

If processing is interrupted while Btrieve is dropping an index, you can access the data in the file through the file's other indexes. Btrieve will return a status code 56 (Incomplete Index) if you try to access the file by the incomplete index. In this case, re-issue the Drop Supplemental Index operation.

Current Positioning:

The Drop Supplemental Index operation has no effect on positioning.

END TRANSACTION (20)

Purpose:

End Transaction completes a transaction and makes the appropriate changes to affected data files.

Parameter Usage:

	Operation	FCB Pos Block	FCB Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X					
Returned						

Description:

An End Transaction marks the completion of a set of logically related Btrieve operations.

Prerequisites:

Before your application issues an End Transaction operation, it must have issued a successful Begin Transaction operation (19).

Procedure:

To perform an End Transaction operation, set the operation code to 20. Btrieve will ignore all other parameters on an End Transaction call.

Result:

If the End Transaction operation is successful, all of the operations bracketed within the transaction will be recorded in your database. Your application cannot abort a transaction after an End Transaction operation.

END TRANSACTION (20)
(Continued)

If the operation is unsuccessful, Btrieve will return a non-zero status. The most common non-zero status code is code 38 (Transaction Control File Error), which can result if the transaction control file has been deleted or cannot be written to for some reason.

Current Positioning:

An End Transaction operation does not affect positioning.

EXTEND (16)

Purpose:

Extend divides a Btrieve file over two logical disk drives.

Parameter Usage:

	Operation	FCB Pos Block	FCB Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X			X	X
Returned		X				

Description:

The Extend operation allows your application to expand a single Btrieve file to a second logical disk drive.

Prerequisites:

Before your application issues an Extend operation, the following prerequisites must be met:

- The file to be extended must be open.
- Btrieve must have access to the volume to which the file will be extended.
- No transactions can be active.

Procedure:

To perform an Extend operation, set the Btrieve parameters as follows:

- Set the operation code to 16.
- Pass the position block for the file to be extended.

EXTEND (16)

(Continued)

- Store the name of the extension file in the key buffer. Specify the device name and the full path name for the file. Terminate the name of the extension file with a blank or a binary zero.
- Specify a value of –1 in the key number parameter when you perform the Extend operation if you want Btrieve to begin storing data in the extension file immediately. Normally, Btrieve does not store data in the extension file until the drive containing the original file is full.

Result:

If the Extend operation is successful, Btrieve will extend the file across two logical volumes. To access an extended file, use the following guidelines:

- Immediately after the file is extended, your application must close and reopen the file before it can access the extension.
- Both the original drive and the extension drive must be online whenever your program accesses the extended file. Btrieve must be able to find the extension file on the logical disk drive you specified.
- Once you have created the extended file, you cannot move it to a different drive. When you extend a file, Btrieve writes the full path name specified for the extension to an address in the original data file. Therefore, every workstation must use the same drive designator to refer to the drive containing the extended file.

If the operation is unsuccessful, Btrieve will return a non-zero status. Common errors returned from an Extend operation include the following:

- 31 File Already Extended
- 32 Extend I/O Error
- 34 Invalid Extension Name

Current Positioning:

An Extend operation does not affect current positioning.

GET DIRECT (23)

Purpose:

Get Direct retrieves the data record positioned at a specified physical address in the Btrieve file.

Parameter Usage:

		FCB				
	Operation	Pos Block	Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X	X	X		X
Returned		X	X	X	X	

Description:

The Get Direct operation allows your application to retrieve a record using its physical location in the file instead of using one of the defined index paths. You can use Get Direct in the following ways:

- You can retrieve a record faster by using its physical location instead of its key value.
- You can use the Get Position operation to retrieve the 4-byte location of a record, save the location, and then later use Get Direct to return directly to that location.
- You can use the 4-byte location to retrieve a record in a chain of duplicates without rereading all the records from the beginning of the chain.
- You can change the current access path. A Get Position operation, followed by a Get Direct operation with a different key number, establishes positioning for the current record in a different index tree. A subsequent Get Next will return the next record in the file based on the new access path.

GET DIRECT (23)

(Continued)

Prerequisites:

Before your application issues a Get Direct operation, the following prerequisites must be met:

- The file must be open.
- Your application must have previously retrieved the 4-byte location of the record by issuing a Get Position operation.

Procedure:

To perform a Get Direct operation, set the Btrieve parameters as follows:

- Set the operation code to 23.
- Store the 4-byte position of the requested record in the first four bytes of the data buffer.
- Specify the total length of the data buffer so that Btrieve can determine whether the record will fit in your buffer.
- Specify the access path for which Btrieve is to establish positioning in the key number parameter.

Result:

If the Get Direct operation is successful, Btrieve will

- Store the requested record in the data buffer, overwriting the 4-byte offset stored in the first four bytes;
- Store the actual length of the record in the data buffer length parameter;
- Store the key value for the specified access path in the key buffer.

GET DIRECT (23)
(Continued)

If Btrieve cannot return the requested record, it will return a non-zero status. Common non-zero status codes include the following:

- 22 Data Buffer Too Short
- 43 Invalid Data Record Address

Current Positioning:

After a Get Direct operation, Btrieve erases any existing positioning information and establishes the current position according to the key number specified.

- The **next** record becomes the next duplicate of the key value returned if a duplicate exists. Otherwise it becomes the first record for the key value greater than the one returned.
- The **previous** record becomes either the previous duplicate for the key returned or the last duplicate of the key value less than the one returned.

GET DIRECTORY (18)

Purpose:

Get Directory retrieves the current directory.

Parameter Usage:

		FCB				
	Operation	Pos Block	Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X					X
Returned					X	

Description:

The Get Directory operation returns the current directory for a specified logical drive.

Prerequisites:

Your application can issue a Get Directory operation immediately after loading the Record Manager. The key buffer should be at least 65 characters long.

Procedure:

To retrieve the current directory, set the Btrieve parameters as follows:

- Set the operation code to 18
- Store the logical drive number in the key number parameter before calling Btrieve. Specify the drive as 1 for A, 2 for B, etc. To use the default drive, specify 0.

GET DIRECTORY (18)
(Continued)

Result:

Btrieve will return the current directory, terminated by a binary 0, to the key buffer.

Current Positioning:

A Get Directory operation does not affect current positioning.

GET EQUAL (5)

Purpose:

Get Equal retrieves a record corresponding to a specified key value.

Parameter Usage:

	Operation	FCB Pos Block	FCB Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X		X	X	X
Returned		X	X	X		

Description:

Using the Get Equal operation, your application can retrieve a record based on the key value specified in the key buffer.

Prerequisites:

Before your application issues a Get Equal operation, the following prerequisites must be met:

- The file must be open.
- The file cannot be a data-only file with no indexes defined.

Procedure:

To perform the operation, set the Btrieve parameters as follows:

- Initialize the operation code to 5.
- Pass the position block for the file.
- Specify the desired key value in the key buffer.

GET EQUAL (5)
(Continued)

- Set the key number to the correct access path.
- Initialize the data buffer length to a value equal to the length of the record you want to retrieve.

Result:

If the Get Equal operation is successful, Btrieve will

- Return the requested record in the data buffer.
- Return the length of the record in bytes in the data buffer length parameter.

If the Get Equal operation is not successful, Btrieve will return a non-zero status code indicating the reason. Common non-zero status codes include the following:

- 3 File Not Open
- 4 Key Value Not Found
- 22 Data Buffer Too Short

Current Positioning:

After a Get Equal operation, Btrieve erases any existing positioning information and establishes its position in the index as follows:

- If a duplicate exists, the **next** record becomes the first duplicate of the key value returned. Otherwise, the next record becomes the first record for the key value greater than the one returned.
- The **previous** record becomes the last duplicate of the key value less than the one returned. If no duplicates exist for that key value, the previous record becomes the only record for the key value less than the one returned.

GET FIRST (12)

Purpose:

Get First retrieves the record corresponding to the first key value for a specified access path.

Parameter Usage:

		FCB				
	Operation	Pos Block	Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X		X		X
Returned		X	X	X	X	

Description:

The Get First operation enables your application to retrieve the record which corresponds to the first key value for a specified key number.

Prerequisites:

Before your application issues a Get First operation, the following prerequisites must be met:

- The file must be open.
- The file cannot be a data-only file with no indexes defined.

Procedure:

To perform this operation, set the Btrieve parameters as follows:

- Initialize the operation code to 12.
- Pass the position block for the file.
- Indicate the key number for the access path.
- Specify the length of the data buffer.

GET FIRST (12)
(Continued)

Result:

If the Get First operation is successful, Btrieve will

- Return the requested record in the data buffer;
- Store the corresponding key value in the key buffer;
- Return the length of the record in the data buffer length parameter.

If the Get First operation is not successful, Btrieve will return a non-zero status indicating the reason. Common non-zero status codes include the following:

- 3 File Not Open
- 6 Invalid Key Number
- 22 Data Buffer Too Short

Current Positioning:

After a Get First operation, Btrieve erases any existing positioning information and establishes its position in the index as follows:

- The **previous** record points beyond the beginning of the file.
- The **next** record becomes the next duplicate of the key value returned, or, if no duplicates exist, the first record for the key value greater than the one returned.

GET GREATER (8)

Purpose:

Get Greater retrieves a record corresponding to a key value greater than the specified key value.

Parameter Usage:

		FCB				
	Operation	Pos Block	Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X		X	X	X
Returned		X	X	X	X	

Description:

Using the Get Greater operation your application can ascend the access path specified by the key number to find the first key value greater than the one specified in the key buffer.

Prerequisites:

Before your application issues a Get Greater operation, the following prerequisites must be met:

- The file must be open.
- The file cannot be a data-only file.

Procedure:

To perform a Get Greater operation, set the Btrieve parameters as follows:

- Set the operation code to 8.
- Pass the position block for the file.

GET GREATER (8)
(Continued)

- Store the desired key value in the key buffer parameter.
- Set the key number parameter to correspond to the correct access path.
- Specify the length of the data buffer.

Result:

If the operation is successful, Btrieve will

- Return the corresponding record in the data buffer;
- Return the length of the record in the data buffer length parameter.

If the operation is unsuccessful, Btrieve will return a non-zero status indicating the reason. Common non-zero status codes include the following:

- 3 File Not Open
- 6 Invalid Key Number
- 22 Data Buffer Too Short

Current Positioning:

After a Get Greater operation, Btrieve erases any existing positioning information and establishes its position in the index as follows:

- If a duplicate exists, the **next** record becomes the first duplicate of the key value returned. Otherwise, the next record becomes the first record for the key value greater than the one returned.
- The **previous** record becomes the last duplicate of the key value less than the one returned. Otherwise, the previous record becomes the only record for the key value less than the one returned.

GET GREATER OR EQUAL (9)

Purpose:

Get Greater Or Equal retrieves a record with a key value greater than or equal to a specific key value.

Parameter Usage:

		FCB				
	Operation	Pos Block	Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X		X	X	X
Returned		X	X	X	X	

Description:

The Get Greater Or Equal operation allows your application to retrieve a record that is either equal to or greater than a specified key value. Btrieve first searches for a key value that is equal to the value you specify. If Btrieve cannot find an equal key value, it ascends the access path until it finds the record with the next higher key value.

Prerequisites:

Before your application issues a Get Greater Or Equal operation, the following prerequisites must be met:

- The file must be open.
- The file cannot be a data-only file.

Procedure:

To perform a Get Greater Than Or Equal operation, set the Btrieve parameters as follows:

- Set the operation code to 9.
- Pass the position block for the file.

GET GREATER OR EQUAL (9)

(Continued)

- Store the desired key value in the key buffer parameter.
- Set the key number parameter to correspond to the correct access path.
- Specify the length of the data buffer.

Result:

If Btrieve finds a record in the file that satisfies the requirements of the Get Greater Or Equal operation, it will

- Store the record in the data buffer;
- Return the length of the record in the data buffer length parameter.

If Btrieve is unable to return a record, it will return a non-zero status indicating the reason. Common non-zero status codes include the following:

- 3 File Not Open
- 6 Invalid Key Number
- 22 Data Buffer Too Short

Current Positioning:

After a Get Greater Or Equal operation, Btrieve erases any existing positioning information and establishes the current position as follows:

- The **next** record becomes the first duplicate of the key value returned if a duplicate exists. Otherwise it becomes the first record for the key value greater than the one returned.
- The **previous** record becomes the last duplicate of the key value less than the one returned, or, if no duplicates exist, the only record for the key value less than the one returned.

GET KEY (+50)

Purpose:

Get Key allows you to perform a Get operation without actually retrieving a data record. You can use Get Key to detect the presence of a value in a file. A Get Key operation is generally faster than its corresponding Get operation. The Get Key operation can be used with any of the following Get operations:

- GET EQUAL (5)
- GET NEXT (6)
- GET PREVIOUS (7)
- GET GREATER (8)
- GET GREATER OR EQUAL (9)
- GET LESS THAN (10)
- GET LESS THAN OR EQUAL (11)
- GET FIRST (12)
- GET LAST (13)

Parameter Usage:

The parameters are the same as those for the corresponding Get operation except that Btrieve ignores the data buffer length and does not return a record in the data buffer.

Prerequisites:

The prerequisites for a Get Key operation are the same as those for the corresponding Get operation.

GET KEY (+50)
(Continued)

Procedure:

To perform a Get Key operation, set the Btrieve parameters as you would for the corresponding Get operation. You do not need to initialize the data buffer length.

You must add 50 to the operation code for the Get operation you want to perform. For example, to perform a Get Key (operation code 50) with the Get Equal operation (operation code 5), use 55 as the operation code.

If Btrieve finds the requested key, it will return the key in the key buffer and a status of 0. Otherwise, Btrieve will return a non-zero status indicating why it cannot find the key.

Current Positioning:

A Get Key operation establishes the current positioning exactly as the corresponding Get operation does, except that Get Next Key and Get Previous Key do not return duplicates.

GET LAST (13)

Purpose:

Get Last retrieves the record corresponding to the last key value for a specified access path.

Parameter Usage:

		FCB				
	Operation	Pos Block	Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X		X		X
Returned		X	X	X	X	

Description:

Using the Get Last operation, your application can retrieve a record which corresponds to the last key value for a specified key number. If duplicates exist for the last key value, the record returned is the last duplicate.

Prerequisites:

Before your application issues a Get Last operation, the following prerequisites must be met:

- The file must be open.
- The file cannot be a data-only file with no indexes defined.

Procedure:

To perform this operation, set the Btrieve parameters as follows:

- Initialize the operation code to 13.
- Pass the position block for the file.
- Specify the length of the data buffer.
- Specify the key number for the access path.

GET LAST (13)
(Continued)

Result:

If the operation is successful, Btrieve will

- Return the requested record in the data buffer;
- Store the corresponding key value in the key buffer;
- Return the length of the record in the data buffer length parameter.

If Btrieve is unable to return a record, it will return a non-zero status code indicating the reason. Common non-zero status codes include the following:

- 3 File Not Open
- 6 Invalid Key Number
- 22 Data Buffer Too Short

Current Positioning:

After a Get Last operation, Btrieve erases any existing positioning information and establishes its position in the index as follows:

- The **next** record points beyond the end of the file.
- The **previous** record becomes the previous duplicate of the key value returned, or, if no duplicates exist, the last duplicate for the key value less than the one returned.

GET LESS THAN (10)

Purpose:

Get Less Than retrieves a record corresponding to the key value which is less than a specified key value.

Parameter Usage:

		FCB				
	Operation	Pos Block	Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X		X	X	X
Returned		X	X	X	X	

Description:

Using the Get Less Than operation, your application can retrieve a record that corresponds to the first key value which is less than a specified key value. Btrieve descends the access path specified by the key number to find the first key value less than the one requested. Once it locates the correct key value, it returns the corresponding data record in the data buffer.

Prerequisites:

Before your application issues a Get Less Than operation, the following prerequisites must be met:

- The file must be open.
- The file cannot be a data-only file with no indexes defined.

Procedure:

To perform a Get Less Than operation, set the Btrieve parameters as follows:

- Set the operation code to 10.
- Pass the position block for the file.

GET LESS THAN (10)

(Continued)

- Store the desired key value in the key buffer parameter.
- Set the key number parameter to the desired access path.
- Specify the length of the data buffer.

Result:

If the operation is successful, Btrieve will

- Return the record in the data buffer;
- Return the key value for the record in the key buffer;
- Return the length of the record in the data buffer length parameter.

If Btrieve is unable to return a record, it will return a non-zero status indicating the reason. Common non-zero status codes include the following:

- 3 File Not Open
- 6 Invalid Key Number
- 22 Data Buffer Too Short

Current Positioning:

After a Get Less Than operation, Btrieve erases any existing positioning information and establishes its position in the index as follows:

- If a duplicate exists, the **next** record becomes the first duplicate of the key value returned. Otherwise, the next record becomes the first record for the key value greater than the one returned.
- If a duplicate exists, the **previous** record becomes the last duplicate of the key value less than the one returned. Otherwise, the previous record becomes the only record for the key value less than the one returned.

GET LESS THAN OR EQUAL (11)

Purpose:

Get Less Than Or Equal retrieves a record with a key value that is less than or equal to the key value specified in the key buffer.

Parameter Usage:

		FCB				
	Operation	Pos Block	Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X		X	X	X
Returned		X	X	X	X	

Description:

Using the Get Less Than Or Equal operation, your application can retrieve a record that is equal to or less than a specified key value. Btrieve first searches the access path for the specified key value. If it does not find the value, it descends the access path specified by the key number to find the first key value less than the one requested. Once it locates the correct key value, it returns the corresponding data record in the data buffer.

Prerequisites:

Before your application issues a Get Less Than Or Equal operation, the following prerequisites must be met:

- The file must be open.
- The file cannot be a data-only file with no indexes defined.

GET LESS THAN OR EQUAL (11)
(Continued)

Procedure:

To perform a Get Less Than Or Equal Operation, set the Btrieve parameters as follows:

- Set the operation code to 11.
- Pass the position block for the file.
- Store the desired key value in the key buffer parameter.
- Set the key number parameter to the desired access path.
- Specify the length of the data buffer.

Result:

If the operation is successful, Btrieve will

- Return the record in the data buffer;
- Return the key value for the record in the key buffer;
- Return the length of the record in the data buffer length parameter.

If Btrieve is unable to return a record, it will return a non-zero status indicating the reason. Common non-zero status codes include the following:

- 3 File Not Open
- 6 Invalid Key Number
- 22 Data Buffer Too Short

GET LESS THAN OR EQUAL (11)
(Continued)

Current Positioning:

After a Get Less Than Or Equal operation, Btrieve erases any existing positioning information and establishes the current position as follows:

- If a duplicate exists, the **next** record becomes the first duplicate of the key value returned. Otherwise, the next record becomes the first record for the key value greater than the one returned.
- If a duplicate exists, the **previous** record becomes the last duplicate of the key value less than the one returned. Otherwise, the previous record becomes the only record for the key value less than the one returned.

GET NEXT (6)

Purpose:

Get Next retrieves the record from a Btrieve file that follows the current record in the key path.

Parameter Usage:

		FCB				
	Operation	Pos Block	Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X		X	X	X
Returned		X	X	X	X	

Description:

Using the Get Next operation, your application can retrieve records in order according to a specified access path. Only the Get First, Get Next, Get Previous, and Get Last operations allow an application to retrieve records for duplicate key values.

Prerequisites:

Before your application issues a Get Next operation, the following prerequisites must be met:

- The file must be open.
- The file cannot be a data-only file with no indexes defined.
- Your application must have established a position in the index on the Btrieve call immediately preceding the Get Next operation.

GET NEXT (6)
(Continued)

Procedure:

To perform a Get Next operation, set the Btrieve parameters as follows:

- Set the operation code to 6.
- Pass the position block for the file.
- Store the key value from the previous operation in the key buffer. Pass the key buffer exactly as Btrieve returned it on the previous call since Btrieve may need the information previously stored there to determine its current position in the file.
- Set the key number parameter to the access path used on the previous call. You cannot change access paths using a Get Next operation.
- Specify the length of the data buffer.

Result:

If the operation is successful, Btrieve will

- Return the record in the data buffer;
- Return the key value for the record in the key buffer;
- Return the length of the record in the data buffer length parameter.

If Btrieve is unable to return a record, it will return a non-zero status indicating the reason. Common non-zero status codes include the following:

• 3	File Not Open
• 6	Invalid Key Number
• 7	Different Key Number
• 9	End of File

GET NEXT (6)

(Continued)

- 22 Data Buffer Too Short
- 82 Lost Position

Current Positioning:

Btrieve uses the positioning established by the previous call to perform a Get Next operation, updating that positioning as follows:

- If a duplicate exists, the **next** record becomes the first duplicate of the key value returned. Otherwise, the next record becomes the first record for the key value greater than the one returned.
- If a duplicate exists, the **previous** record becomes the last duplicate of the key value less than the one returned. Otherwise, the previous record becomes the only record for the key value less than the one returned.

GET POSITION (22)

Purpose:

Get Position returns the physical position of the current record.

Parameter Usage:

	Operation	FCB Pos Block	FCB Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X		x		
Returned		X	X	X		

Description:

Using the Get Position operation, your application can obtain the 4-byte position of the current record within a Btrieve file. To establish the current record, your application may perform any of the other Get operations, an Insert operation, or an Update operation. Your application can then issue a Get Position operation to retrieve the record's address. Once your application knows a record's address, it can use the Get Direct operation to retrieve that record directly by its physical location in the file.

Btrieve does not perform any disk I/O to process a Get Position request.

Prerequisites:

Before your application issues a Get Position operation, the following prerequisites must be met:

- The file must be open.
- The Btrieve call to the file immediately prior to the Get Position call must have retrieved a record. You cannot issue a call using the same position block between the retrieval of the record and the Get Position call.

GET POSITION (22)
(Continued)

Procedure:

To perform a Get Position operation, set the Btrieve parameters as follows:

- Set the operation code to 22.
- Pass the position block for the file.
- Indicate a data buffer that is long enough to store the 4-byte position.
- Set the data buffer length to at least four bytes.

Result:

If the operation is successful, Btrieve will

- Return the position of the record in the data buffer. The position is a 4-byte binary value (most significant word first) that indicates the record's offset (in bytes) into the file.
- Set the data buffer length to four bytes.

If Btrieve cannot determine the current record or if it cannot return the position, it will return a non-zero status code indicating the reason. The most common non-zero status that Btrieve returns is a status code 8 (Invalid Positioning).

Current Positioning:

A Get Position operation does not affect current positioning.

GET PREVIOUS (7)

Purpose:

Get Previous retrieves the record that precedes the current record in the key path.

Parameter Usage:

		FCB				
	Operation	Pos Block	Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X		X	X	X
Returned		X	X	X	X	

Description:

Using the Get Previous operation, your application can retrieve records in order according to a specified access path. Only the Get First, Get Next, Get Previous, and Get Last operations allow an application to retrieve records for duplicate key values.

Prerequisites:

Before your application issues a Get Previous operation, the following prerequisites must be met:

- The file must be open.
- The Btrieve call to the file immediately prior to the Get Previous call must have retrieved a record. You cannot issue a Btrieve call using the same position block between the retrieval of the record and the Get Previous call.

GET PREVIOUS (7)

(Continued)

Procedure:

To perform a Get Previous operation, set the Btrieve parameters as follows:

- Set the operation code to 7.
- Pass the position block for the file.
- Specify the correct key number.
- Specify the length of the data buffer.
- Pass the key buffer exactly as Btrieve returned it on the previous call. Btrieve may need the information previously stored in the key buffer to determine its current position in the file.

Result:

If the operation is successful, Btrieve will

- Update the key buffer with the key value for the new record;
- Return the previous record in the data buffer;
- Return the length of the record in the data buffer length parameter.

If the operation is not successful, Btrieve will return a non-zero status indicating the reason. Common non-zero status codes include the following:

- 3 File Not Open
- 9 End of File
- 22 Data Buffer Too Short

GET PREVIOUS (7)
(Continued)

Current Positioning:

Btrieve uses the positioning established by the previous call to perform a Get Previous operation as follows.

- The record that was the current record when the call was initiated becomes the **next** record.
- If a duplicate exists, the **previous** record becomes the last duplicate of the key value less than the one returned. Otherwise, the previous record becomes the only record for the key value less than the one returned.

INSERT (2)

Purpose:

The Insert operation inserts a record into a file.

Parameter Usage:

		FCB				
	Operation	Pos Block	Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X	X	X		X
Returned		X			X	

Description:

Your application can use the Insert operation to insert a new record into a file. Btrieve adjusts all the key indexes to reflect the key values for the new record at the time the record is inserted.

Prerequisites:

Before your application issues an Insert operation, the following prerequisites must be met:

- The file must be open.
- The record to be inserted must be the proper length, and the key values must conform to the keys defined for the file.

Procedure:

To perform the Insert operation, set the Btrieve parameters as follows:

- Specify an operation code of 2.
- Store the new data record in the data buffer.

INSERT (2)

(Continued)

- Specify the length of the data buffer. This value must be at least as long as the fixed-length portion of the record.
- Specify the key number for which you want Btrieve to maintain position.

Result:

If the Insert operation is successful, Btrieve will

- Place the new record in the file;
- Update all index information to reflect the new record;
- Return the key value for the current access path.

If the Insert operation is not successful, Btrieve will return a non-zero status indicating the reason. Common non-zero status codes include the following:

• 2	I/O Error
• 3	File Not Open
• 5	Duplicates Error
• 14	Pre-Image Open Error
• 15	Pre-Image I/O Error
• 18	Disk Full
• 21	Key Buffer Too Short
• 22	Data Buffer Too Short

INSERT (2)
(Continued)

Current Positioning:

An Insert operation erases any existing positioning information. Based on the key number you specify, Btrieve establishes its position in the index as follows:

- The first data record for the key value greater than the key just inserted becomes the **next** record.
- The last data record for the key value less than the key just inserted becomes the **previous** record.

LOCKS

Purpose:

Locks allow you to control access to records and files, preventing workstations from performing conflicting operations on the database.

Parameters:

With the exception of the operation code, the parameters for a lock operation are the same as those for the corresponding record operation.

Description:

Btrieve recognizes two distinct kinds of record locks: single locks and multiple locks. You can specify either wait or nowait type locks for either single or multiple record locks.

You can specify a lock with any Get, Step, Open, or Begin Transaction operation. Adding a lock bias to a Begin Transaction operation only specifies whether you want a wait or nowait transaction. It does not cause Btrieve to use record locks during the transaction.

Single Record Locks. When a workstation uses single record locks, it can only lock one record in a file at a time. Btrieve releases a single record lock when the workstation issues another Get operation with a lock for the same file, updates or deletes the locked record, or issues an Unlock operation.

Multiple Record Locks. Multiple record locks allow an application to lock multiple records in a file, and then update or delete those records as necessary. When you use multiple record locks, Btrieve still locks only one record for each Get operation. However, it does <u>not</u> unlock the record when you update a locked record or issue another Get operation with a multiple lock. Your application can release one or all of the multiple record locks using the Unlock operation.

LOCKS
(Continued)

Wait locks. If another workstation has the record locked or has a transaction pending for the file when you issue a lock with the wait option, Btrieve will wait until the record is available before returning control to the application.

Nowait locks. If another workstation has the record locked or has a transaction pending for the file when you issue a lock with the nowait option, Btrieve will immediately return a status of 84 or 85 to the application, indicating that the record is busy.

Prerequisites:

With the exception of the operation code, the parameters required for a record lock operation are identical to those required for the corresponding operation with no lock.

Procedure:

To specify a record lock, your application adds a value (called a lock bias) to any Get, Step, Open, or Begin Transaction operation code. The following table illustrates the lock bias values:

Value	Lock Type
+100	single record wait lock
+200	single record nowait lock
+300	multiple record wait lock
+400	multiple record nowait lock

Use the lock bias values as follows:

- *To specify a single record lock,* add 100 (wait) or 200 (nowait) to the operation code.
- *To specify a multiple record lock,* add 300 (wait) or 400 (nowait) to the operation code.

LOCKS
(Continued)

For example, to issue a Get Equal with a single record wait lock, the operation code is (100 + 5) or 105. For the same operation with a multiple record wait lock, the operation code is (300 + 5) or 305. To issue a Get Last with a single record nowait lock, the operation code is (13 + 200) or 213. For the same operation with a multiple record nowait lock, the operation code is (13 + 400), or 413.

To specify a wait transaction, set the operation code to 119 or 319. In this case, the lock code specifies that you want Btrieve to wait if it encounters a file or a record that is locked by another workstation. Specifying 319 for a Begin Transaction operation is equivalent to specifying either 19 or 119. See the description of transaction control in Chapter 2 for more information.

You may also issue a nowait Begin Transaction request with an operation code of 219 or 419. In this case, Btrieve will return a status of 84 or 85 if your application attempts to access a locked record or file within a transaction. See the description of transaction control in Chapter 2 for more information.

Opening Locked Files. If a file is locked when a workstation attempts to open it, Btrieve normally waits until the file is available before executing the Open operation. This is equivalent to a wait lock.

You can specify a nowait open call by passing either 200 or 400 as the operation code for the Open operation (0 + 200 or 0 + 400). If the file you are attempting to open is locked, Btrieve will return a status code 85 (File In Use) to the application. You can then retry the operation until the file is available.

Releasing Multiple Record Locks. As mentioned earlier, Btrieve does not automatically release a multiple lock as it does a single record lock. Records that you lock with a multiple record lock remain locked until you do one of the following:

- Explicitly release the lock by issuing a Btrieve Unlock operation (27).
- Delete the record.
- Issue a Btrieve Reset operation (28).

LOCKS
(Continued)

- Close the file.
- Access the file within a transaction.

Result:

You cannot mix single and multiple locks in the same file from one workstation. If a single record lock (+100/+200) is in effect when a workstation issues a multiple lock request (+300/+400), Btrieve will return an incompatible lock error. The reverse situation will result in the same error. In either case, Btrieve will not lock the record. This does not mean that a workstation is limited to only one type of lock for a particular file. Btrieve will only return the error if one type of lock is *currently in effect* when the workstation attempts a lock of the other type.

If a workstation attempts to unlock a multiple record lock when it does not hold one for that position, Btrieve will return a lock error status.

In addition, Btrieve will return a lock error status if your application attempts to lock more records than you specified when you configured BSERVER. Refer to the description of the NetWare Btrieve initialization options in Chapter 3 for more information.

Common nonzero status codes that Btrieve returns from unsuccessful lock operations include the following:

• 81	Lock Error
• 84	Record in Use
• 85	File in Use
• 93	Incompatible Lock Type

Positioning:

Lock operations have no effect on Btrieve's position in an index.

OPEN (0)

Purpose:

The Open operation makes a file available for access.

Parameter Usage:

		FCB				
	Operation	Pos Block	Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X		X	X	X	X
Returned		X				

Description:

Your application cannot access a Btrieve file unless it first performs an Open operation. The file does not have to reside in the current directory as long as you specify the full path name.

Prerequisites:

Before your application issues an Open operation, the following prerequisites must be met:

- The file to be opened must exist on an accessible drive. If the file has an extension, both of the storage devices on which the file exists must be accessible.
- A file handle must be available for the file.

OPEN (0)
(Continued)

Procedure:

To perform an Open operation, set the Btrieve parameters as follows:

- Specify an operation code of 0.
- Place the name of the file to be opened in the key buffer parameter. Terminate the filename with a blank or binary 0. If the file is not in the current directory, specify the device name and path name for the file, including the directory levels.
- If the file has an owner, specify the owner name, terminated by a binary 0, in the data buffer.
- Specify the length of the owner name, including the binary 0, in the data buffer length parameter.
- Specify one of the following mode specifications in the key number parameter.

Mode	Description
–1	Accelerated In accelerated mode, your application can disable Btrieve's automatic data recovery capability in order to increase update performance. See "Accelerated Access" beginning on page 2-25 for a more complete description of this option. Btrieve locks a buffer in cache for each file opened in accelerated mode. The number of files you can open at once in accelerated mode depends on the memory and page size options you specify when you load Btrieve.

OPEN (0)
(Continued)

Mode	Description
–2	Read-Only This mode allows an application to open a damaged file that Btrieve cannot automatically recover. When Btrieve opens a file in the read-only mode, your application can only read the file; it cannot perform updates. If the file's indexes have been damaged, the records can be retrieved by opening the file in read-only mode and then using using the Step Next operation.
–3	Verify Verify mode only applies to files that are located on local DOS disks. If your application opens a local file in verify mode, Btrieve enables the DOS Verify option during each operation. After each write to the disk, the operating system rereads the data to ensure that it has been recorded correctly. Although disk recording errors are very rare, Btrieve provides this function in case you want to verify the proper recording of critical data.
–4	Exclusive Exclusive mode gives a workstation exclusive access to a file on a shared device. No other workstation can open that file until the workstation that has exclusive access to the file closes it. Exclusive mode is only valid for files that are located on a shared device. If you request exclusive mode for a file on a local disk, Btrieve opens the file in normal mode.
Other	Normal

OPEN (0)
(Continued)

To access a Btrieve file from BASIC, two steps are necessary. First, your application should issue a BASIC OPEN operation for the device NUL in order to use the FIELD statement for the file's data buffer. (See "Calling Btrieve From BASIC" beginning on page 5-5 for more information.) Second, the application must perform a Btrieve Open operation. Other languages do not require the first step.

Btrieve allows as many as 255 open files for compiled BASIC, Pascal, COBOL, or C applications. When multiple files are open at the same time, Btrieve uses the current position block to determine which file to access for a particular call.

NOTE:
Although Btrieve allows an application to open up to 255 files, the BASIC interpreter and some BASIC compilers allow a maximum of only 15 open files. To access more than three files, BASIC requires that you specify the /files parameter when you initiate the BASIC interpreter. When you open multiple files at the same time from BASIC, Btrieve uses the FCB to determine which file to access for a particular call. Refer to the documentation supplied with your BASIC interpreter or compiler for more information.

Result:

If the Open operation is successful, Btrieve will

- Assign a file handle to the file;
- Reserve the position block passed on the Open call for the newly opened file;
- Make the file available for access.

OPEN (0)
(Continued)

If the operation is unsuccessful, Btrieve will return a non-zero status code. Common non-zero status codes for the Open operation include the following:

- 2 I/O Error
- 46 Access To File Denied
- 85 File In Use
- 86 File Table Full
- 87 Handle Table Full

Current Positioning:

An Open operation does not establish any positioning information.

RESET (28)

Purpose:

Reset releases all resources held by a workstation, such as locks left pending by an abnormal termination of an application.

Parameter Usage:

	Operation	FCB Pos Block	FCB Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X				X	X
Returned						

Description:

An application can perform a Reset operation to release all Btrieve resources held by a workstation on the network. This operation aborts any transactions the workstation has pending, releases all locks, and closes any open files for the workstation.

Prerequisites:

Your application can issue a Reset operation at any time after the Record Manager is loaded.

Procedure:

To perform a Reset operation, set the Btrieve parameters as follows:

- Set the operation code to 28.
- Set the key number parameter to –1 if your application is releasing resources for another workstation on the network.
- Store the connection number of the workstation to be reset as an integer in the first 2 bytes of the key buffer.

RESET (28)
(Continued)

Result:

If the Reset operation is successful, Btrieve will

- Close all open files for the specified workstation;
- Release all locks held by the specified workstation;
- Abort any active transaction at the specified workstation.

If the operation fails for any reason, Btrieve will return a non-zero status.

Current Positioning:

The Reset operation destroys all positioning information because it closes any open files.

SET DIRECTORY (17)

Purpose:

The Set Directory operation sets the current directory to a specified value.

Parameter Usage:

		FCB				
	Operation	Pos Block	Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X				X	
Returned						

Description:

A Set Directory operation changes the current directory to the directory specified in the key buffer parameter.

Prerequisites:

Before your application issues a Set Directory operation, the target drive and directory must be accessible.

Procedure:

To set the current directory, set the Btrieve parameters as follows:

- Set the operation code to 17.
- Store the desired drive and directory path, terminated by a binary 0, in the key buffer. If you omit the drive name, Btrieve uses the default drive. If you do not specify the complete path for the directory, Btrieve appends the directory path specified in the key buffer to the current directory.

SET DIRECTORY (17)

(Continued)

Result:

If the Set Directory operation is successful, Btrieve will make the directory specified in the key buffer the current directory.

If the operation is not successful, Btrieve will leave the current directory unchanged and return a non-zero status.

Current Positioning:

Set Directory has no effect on positioning.

SET OWNER (29)

Purpose:

Set Owner assigns an owner name to a file.

Parameter Usage:

		FCB				
	Operation	Pos Block	Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X	X	X	X	X
Returned		X				

Description:

The Set Owner operation assigns an owner name to a file so that users who do not know the name cannot access the file. If an owner name has been set for a file, users or applications must specify the owner name each time they attempt to open the file. You can specify that an owner name be required for any access or just for update privileges.

When you assign an owner name to a file, you can also direct Btrieve to encrypt the file's data on the disk. If you specify data encryption, Btrieve will encrypt all the data during the Set Owner operation. The longer the file, the longer Set Owner takes to complete.

Prerequisites:

Before issuing a Set Owner operation, the following prerequisites must be met:

- The file must be open.
- No transactions can be active.
- An owner name can't already be assigned to the file.

SET OWNER (29)
(Continued)

Procedure:

To perform a Set Owner operation, set the Btrieve parameters as follows:

- Set the operation code to 29.
- Pass the position block that identifies the file you want to protect.
- Store the owner name in both the data buffer and the key buffer and pass the data buffer length. Btrieve requires the name to be in both buffers to avoid the possibility of accidentally specifying an incorrect value. The owner name can be up to eight characters long and must end with a binary 0.
- Set the key number parameter to an integer that specifies the type of access restrictions you want to define for the file and whether data should be encrypted. Table 6.2 lists the values you can specify for the key number.

Value	Description
0	Requires an owner name for any access mode (no data encryption)
1	Permits read-only access without an owner name (no data encryption)
2	Requires an owner name for any access mode (with data encryption)
3	Permits read-only access without an owner name (with data encryption)

Table 6.2
Owner name and data encryption codes

SET OWNER (29)
(Continued)

Result:

If the Set Owner operation is successful, Btrieve will

- Not allow access to the file unless an owner name is specified;
- Encrypt the data in the file, if encryption is specified.

Once your application sets an owner name, it remains in effect until your application issues a Clear Owner operation.

If the Set Owner operation is unsuccessful, Btrieve will return a non-zero status. Common non-zero status codes include the following:

- 41 Operation Not Allowed
- 50 Owner Already Set
- 51 Invalid Owner Name

Current Positioning:

Set Owner has no effect on positioning.

STAT (15)

Purpose:

Stat retrieves the characteristics for a specified file.

Parameter Usage:

	Operation	FCB Pos Block	FCB Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X	X	X	X	
Returned			X	X	X	

Description:

Using the Stat operation, your application can determine the characteristics specified for a file when it was first created. In addition, the Stat operation returns the number of records in the file, the number of unique key values stored for each index in the file, the number of unused pages in the file, and any supplemental indexes defined for the file.

Btrieve returns the file characteristics in the data buffer in the same binary format as it does for a Create operation. For 4-byte variables (number of keys and records), Btrieve returns the low part of the number in the first 2 bytes followed by the high part of the number in the last 2 bytes. The reserved areas are allocated even though Btrieve ignores them on a Stat operation.

STAT (15)

(Continued)

The file flags appear as shown below:

- If bit 0 = 1, the file allows variable length records.
- If bit 1 = 1, Btrieve truncates trailing blanks in variable length records.
- If bit 2 = 1, Btrieve preallocated pages for the file.
- If bit 3 = 1, Btrieve compresses the data in the file.
- If bit 4 = 1, Btrieve created the file as a key-only file.
- If bit 6 = 1, Btrieve maintains a10% free space threshold.
- If bit 7 = 1, Btrieve maintains a 20% free space threshold.
- If bit 6 = 1 and bit 7 = 1, Btrieve maintains a 30% free space threshold.

The key specifications appear immediately after the file specifications and are repeated for each key segment in the file. Btrieve sets the key flags as follows:

If bit 0 = 1, the key allows duplicates.

If bit 1 = 1, the key is modifiable.

If bit 2 = 0 and bit 8 = 0, the key is string type.

If bit 2 = 1 and bit 8 = 0, the key is binary type.

If bit 3 = 1, the key has a null value.

If bit 4 = 1, the key has another segment.

If bit 5 = 1, the key is sorted by an alternate collating sequence.

If bit 6 = 1, the key is sorted in descending order.

If bit 7 = 1, the key is a supplemental index.

If bit 8 = 1, the key is an extended type.

If bit 9 = 1, the key is manual.

See the Create operation in this chapter for a table illustrating the decimal values for these flags.

STAT (15)
(Continued)

If you specify an alternate collating sequence for any of the keys or key segments in the file, Btrieve will return the definition of that sequence immediately after the last key specification block. Btrieve will return the data buffer in the format shown in Table 6.3.

	Description	Length
file specs	record length	2
	page size	2
	# of indexes	2
	# of records	4
	file flags	2
	reserved word	2
	unused pages	2

	Description	Length
key specs (repeated)	key position	2
	key length	2
	key flags	2
	# of keys	4
	extended key type	1
	null value	1
	reserved	4

Table 6.3
Data Buffer for Stat Operation

STAT (15)

(Continued)

Prerequisites:

Before performing a Stat operation, your application must first open the Btrieve file.

Procedure:

To perform a Stat operation, set the Btrieve parameters as follows:

- Set the operation code to 15.
- Pass the position block for the file.
- Indicate a data buffer (to hold the file and key statistics), and an alternate collating sequence, if one is defined.
- Specify the length of the data buffer.
- Indicate a key buffer that is at least 64 characters long.

Result:

If the Stat operation is successful, Btrieve will

- Return the file and key characteristics to the data buffer;
- Store the name of the file's extension, terminated by a binary zero, in the key buffer if you have previously extended the file. Otherwise, Btrieve will initialize the first byte of the key buffer to zero.

If the operation is not successful, Btrieve will return a non-zero status. Common non-zero status codes include the following:

• 3	File Not Open
• 22	Data Buffer Too Short

Current Positioning:

The Stat operation does not affect positioning.

STEP FIRST (33)

Purpose:

Step First retrieves the record in the first physical location in the file.

Parameter Usage:

	Operation	FCB Pos Block	FCB Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X		X		
Returned		X	X	X		

Description:

Step First allows your application to retrieve the record in the first physical location in the file. Btrieve does not use an index path to retrieve the record.

Prerequisites:

Before your application can perform a Step First operation, the file must be open.

Procedure:

To perform a Step First operation, set the Btrieve parameters as follows:

- Specify an operation code of 33.
- Pass the position block for the file.
- Indicate a data buffer in which to store the returned record.
- Set the data buffer length parameter to the length of the data buffer.

STEP FIRST (33)
(Continued)

Result:

If the operation is successful, Btrieve will

- Return the first physical record in the file to your application's data buffer;
- Set the data buffer length parameter to the number of bytes it returned in the data buffer.

If the operation is not successful, Btrieve will return a non-zero status indicating the reason. Common non-zero status codes include the following:

- 3 File Not Open
- 9 End of File
- 22 Data Buffer Too Short

Current Positioning:

The Step First operation establishes no positioning in an index.

STEP LAST (34)

Purpose:

Step Last retrieves the record in the last physical location in the file.

Parameter Usage:

	Operation	FCB		Data Buffer Length	Key Buffer	Key Number
		Pos Block	Data Buffer			
Expected	X	X		X		
Returned		X	X	X		

Description:

Step Last allows your application to retrieve the last physical record in the file. Btrieve does not use an index path to retrieve a record for a Step Last operation.

Prerequisites:

Before your application can perform a Step Last operation, the file must be open.

Procedure:

To perform a Step Last operation, set the Btrieve parameters as follows:

- Specify an operation code of 34.
- Pass the position block for the file.
- Indicate a data buffer in which to store the returned record.
- Set the data buffer length parameter to the length of the data buffer.

STEP LAST (34)
(Continued)

Result:

If the operation is successful, Btrieve will

- Return the last physical record in the file to your application's data buffer;
- Set the data buffer length parameter to the number of bytes it returned in the data buffer.

If the operation is unsuccessful, Btrieve will return a non-zero status indicating the reason. Common non-zero status codes include the following:

- 3 File Not Open
- 9 End of File
- 22 Data Buffer Too Short

Current Positioning:

The Step Last operation establishes no positioning in an index.

STEP NEXT (24)

Purpose:

Step Next retrieves a record from the location physically following the current record.

Parameter Usage:

	Operation	FCB		Data Buffer Length	Key Buffer	Key Number
		Pos Block	Data Buffer			
Expected	X	X		X		
Returned		X	X	X		

Description:

Step Next allows your application to retrieve records in the order in which they are physically stored. Btrieve does not use an index path to retrieve a record for a Step Next operation. A Step Next operation issued immediately after an Open operation returns the first record in the file. A Step Next operation issued immediately after any Get or Step operation returns the record physically following the record retrieved by the previous operation.

Your application cannot predict the order in which records will be returned by a Step Next operation.

Prerequisites:

Before your application can perform a Step Next operation, the file must be open.

Procedure:

To perform a Step Next operation, set the Btrieve parameters as follows:

- Specify an operation code of 24.
- Pass the position block for the file.

STEP NEXT (24)
(Continued)

- Indicate a data buffer in which to store the returned record.
- Specify the length of the data buffer.

Result:

If the operation is successful, Btrieve will

- Return the last physical record in the file to your application's data buffer;
- Set the data buffer length parameter to the number of bytes it returned in the data buffer.

If the operation is unsuccessful, Btrieve will return a non-zero status indicating the reason. Common non-zero status codes include the following:

• 3	File Not Open
• 9	End of File
• 22	Data Buffer Too Short

Current Positioning:

The Step Next operation establishes no positioning in an index.

STEP PREVIOUS (36)

Purpose:

Step Previous retrieves a record in the location physically preceding the current record.

Parameter Usage:

		FCB				
	Operation	Pos Block	Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X		X		
Returned		X	X	X		

Description:

Step Previous allows your application to retrieve records in the order in which they are physically stored. Btrieve does not use an index path to retrieve a record for a Step Previous operation. A Step Previous operation immediately after any Get or Step operation returns the record physically preceding the record retrieved by the previous operation.

Prerequisites:

Before your application can perform a Step Previous operation, the following prerequisites must be met:

- The file must be open.
- The previous operation must have been a successful Get or Step operation.

STEP PREVIOUS (35)
(Continued)

Procedure:

To perform a Step Previous operation, set the Btrieve parameters as follows:

- Specify an operation code of 35.
- Pass the position block for the file.
- Indicate a data buffer in which to store the returned record.
- Specify the length of the data buffer.

Result:

If the operation is successful, Btrieve will

- Return the last physical record in the file to your application's data buffer;
- Set the data buffer length parameter to the number of bytes it returned in the data buffer.

If the operation is unsuccessful, Btrieve will return a non-zero status indicating the reason. Common non-zero status codes include the following:

- 3 File Not Open
- 9 End of File
- 22 Data Buffer Too Short

Current Positioning:

The Step Previous operation establishes no positioning in an index.

STOP (25)

Purpose:

The Stop operation terminates the BREQUEST program and removes it from a workstation's memory.

Parameter Usage:

		FCB				
	Operation	Pos Block	Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X					
Returned						

Description:

Stop removes the Btrieve requestor program (BREQUEST) from a workstation's memory. A Btrieve application at that workstation cannot perform any other Btrieve operations until you restart BREQUEST.

The Stop operation removes memory only from the workstation where the Btrieve call is issued. You cannot stop BREQUEST at another workstation.

Prerequisites:

BREQUEST must be loaded before your application can issue a Stop operation.

Procedure:

To perform a Stop operation, your application specifies an operation code of 25.

STOP (25)

(Continued)

Result:

If the Stop operation is successful, Btrieve will

- Remove BREQUEST from memory at the workstation;
- Close all files previously open for the workstation;
- Abort any active transactions;
- Release all locks held by the workstation.

If the Stop operation is unsuccessful, Btrieve will return a non-zero status. The most common non-zero status code is 20 (BREQUEST Not Loaded).

Current Positioning:

The Stop operation does not establish any position.

UNLOCK (27)

Purpose:

The Unlock operation unlocks one or more records that were previously locked.

Parameter Usage:

		FCB				
	Operation	Pos Block	Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X	X	X		X
Returned						

Description:

Unlock explicitly releases one or more locked records for the file associated with the specified position block.

Prerequisites:

Before your application can issue an Unlock operation at a workstation, the workstation must hold at least one record lock.

Procedure:

To unlock a single record lock, set the Btrieve parameters as follows:

- Set the operation code to 27.
- Pass the position block for the file that contains the locked record.
- Set the key number to a non-negative value.

UNLOCK (27)
(Continued)

To unlock one multiple type record lock, first retrieve the 4-byte position of the record you want to unlock by issuing a Get Position operation (22) for that record. Then issue the Unlock operation, setting the Btrieve parameters as follows:

- Set the operation code to 27.
- Pass Btrieve the position block for the file that contains the locked record.
- Store (in the data buffer) the 4-byte position that Btrieve returns.
- Set the data buffer length to 4.
- Initialize the key number parameter to –1.

To unlock all of the multiple record locks on a file, you should set the Btrieve parameters as follows:

- Set the operation code to 27.
- Pass Btrieve the position block for the file which contains the multiple locks.
- Initialize the key number parameter to –2.

Result:

If the Unlock operation is successful, Btrieve will release all of the locks specified by the operation.

If the Unlock operation is not successful, Btrieve will return a non-zero status. The most common non-zero status code is code 81 (Lock Error).

Current Positioning:

An Unlock operation has no effect on positioning.

UPDATE (3)

Purpose:

The Update operation updates an existing record in a Btrieve file.

Parameter Usage:

		FCB				
	Operation	Pos Block	Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X	X	X	X		X
Returned		X			X	

Description:

The Update operation changes the information in an existing record.

Prerequisites:

Before your application can perform an Update operation, the following prerequisites must be met:

- The file must be open.
- The Btrieve call to the file immediately prior to the Update call must have retrieved the record to be updated. You cannot issue a call using the same position block between the time your application retrieves the record and the time it updates the record.

Procedure:

To perform an Update operation, set the Btrieve parameters as follows:

- Set the operation code to 3.
- Pass the position block for the file containing the record.
- Store the updated data record in the data buffer.

UPDATE (3)
(Continued)

- Set the data buffer length to the length of the updated record.
- Store the key number used for retrieving the record in the key number parameter.

Result:

If the Update operation is successful, Btrieve will

- Update the record stored in the file with the new value in the data buffer;
- Adjust the key indexes to reflect any change in the key values;
- Update the key buffer parameter, if necessary.

If the Update operation is not successful, Btrieve will return a non-zero status code. Common non-zero status codes include the following:

• 5	Duplicate Key Error
• 7	Different Key Number
• 8	Invalid Positioning
• 10	Modifiable Key Error
• 14	Pre-Image Open Error
• 15	Pre-Image I/O Error
• 22	Data Buffer Too Short
• 80	Conflict Error

UPDATE (3)
(Continued)

Current Positioning:

An Update operation changes positioning information only when a key value changes. In this case, Btrieve establishes its position in the index, based on the key number you specify, as follows:

- The first record with a key value greater than the updated key becomes the **next** record.
- The first record with a key value less than the updated key becomes the **previous** record.

VERSION (26)

Purpose:

The Version operation returns the current Btrieve version and revision numbers.

Parameter Usage:

	Operation	FCB Pos Block	FCB Data Buffer	Data Buffer Length	Key Buffer	Key Number
Expected	X			X		
Returned			X	X		

Description:

This operation returns the current Btrieve version and revision numbers.

Prerequisites:

The Btrieve Record Manager must be loaded before you can issue a Version operation.

Procedure:

To perform a Version operation, set the Btrieve parameters as follows:

- Specify an operation code of 26.
- Indicate a data buffer at least 5 bytes long.
- Set the data buffer length to 5.

VERSION (26)
(Continued)

Result:

If the Version operation is successful, Btrieve will return the data to the data buffer in the following format:

Size	Description
2	Integer containing version number
2	Integer containing revision number
1	Character containing "N" for NetWare Btrieve

If the Version operation is not successful, Btrieve will return a non-zero status.

Current Positioning:

The Version operation does not affect current positioning.

APPENDIX A: BTRIEVE OPERATION CODES

Code	Operation
0	Open
1	Close
2	Insert
3	Update
4	Delete
5	Get Equal
6	Get Next
7	Get Previous
8	Get Greater
9	Get Greater or Equal
10	Get Less Than
11	Get Less Than or Equal
12	Get First
13	Get Last
14	Create
15	Stat
16	Extend
17	Set Directory
18	Get Directory
19	Begin Transaction
20	End Transaction
21	Abort Transaction
22	Get Position
23	Get Direct
24	Step Direct
25	Stop
26	Version
27	Unlock
28	Reset
29	Set Owner
30	Clear Owner

Code	Operation
31	Create Supplemental Index
32	Drop Supplemental Index
33	Step First
34	Step Last
35	Step Previous

APPENDIX B: STATUS CODES AND MESSAGES

BTRIEVE STATUS CODES

The Btrieve Record Manager returns a status value after each operation an application performs. A value of 0 indicates that the operation was successful. The possible nonzero status codes that the Record Manager returns are described below.

01 INVALID OPERATION

The operation parameter specified in the Btrieve call is invalid.

02 I/O ERROR

An error occurred during disk read/write. This status may indicate that the file has been damaged and must be recreated, or that the filename specified on the open call was not created by Btrieve. This status may also occur if the application writes on the position block allocated for the file.

03 FILE NOT OPEN

An application must perform a successful Open operation before Btrieve can process any other operations. This status may also occur if the application writes on the position block allocated for the file or passes an invalid position block.

04 KEY VALUE NOT FOUND

Btrieve did not find the requested key value in the specified access path.

05 DUPLICATE KEY VALUE

An attempt was made to add a record with a key field containing a duplicate key value to an index that does not allow duplicate values.

06 INVALID KEY NUMBER

The value stored in the key number parameter is not valid for the file being accessed. The key number must correspond to one of the keys defined when the file was created, or to a supplemental index.

07 DIFFERENT KEY NUMBER

The key number parameter changed before a Get Next, Get Previous, Update, or Delete operation. The operation requested requires the same key number parameter as the previous operation because the Record Manager uses positioning information relative to the previous key number.

If you need to change key numbers between consecutive Get Next or get Previous operations, first use the Get Direct operation to re-establish positioning by the new access path.

08 INVALID POSITIONING

An attempt was made to update or delete a record without first performing a Get or Step operation to establish current position. This status may also occur if the application writes on the position block allocated for the file.

09 END OF FILE

An attempt was made to read past the end of file. When following an access path in ascending order (using Get Next operations) Btrieve returns the last record in the access path for the previous request. When following an access path in descending order (using Get Previous operations) Btrieve returns the first record in the access path for the previous request.

10 MODIFIABLE KEY VALUE ERROR

An attempt was made to modify a key field which is defined as nonmodifiable.

11 INVALID FILE NAME

The file name specified does not conform to file naming conventions.

12 FILE NOT FOUND

The file name requested does not exist. Check the key buffer parameter to make sure the file name is terminated with a blank or a binary zero and that it is being passed correctly to Btrieve.

13 EXTENDED FILE ERROR

Btrieve cannot find the extension file for a partitioned file which you have attempted to open. Extension files must be loaded on the disk drive specified when the extension was created with EXTEND. Both the primary file and its extension must be loaded to access a partitioned file.

14 PRE-IMAGE OPEN ERROR

The pre-image file could not be created/opened. There are three possible causes for this error.

- The Record Manager cannot create a new pre-image file to protect future operations because your disk directory is full. Btrieve must be able to create a pre-image file in order to operate.
- The Record Manager may be trying to open the pre-image file to restore file integrity. If the pre-image file has been erased or damaged, the Record Manager cannot restore the file's integrity. In this case, either use RECOVER to retrieve the damaged file's data records in a sequential file, or replace the file with its most recent backup.
- The Record Manager cannot assign a handle to the pre-image file because the Record Manager was not started by a user with access rights to the pre-image file.

15 PRE-IMAGE I/O ERROR

This status indicates an I/O error during the pre-imaging function. Either the disk is full or the pre-image file has been damaged.

- If the disk is full, erase any unnecessary files or use EXTEND to gain additional disk space.
- If the pre-image file has been damaged, the integrity of the Btrieve file cannot be ensured. Either use RECOVER to retrieve the damaged file's data records in a sequential file, or replace the Btrieve file with its most recent backup.

16 EXPANSION ERROR

The directory structure could not be flushed for the expanded file partition. Either the Record Manager cannot close the file or a new page was added to the file and the Record Manager cannot close and reopen the file to update the directory structure. Check for a damaged disk. This status may also occur if the application writes on the position block allocated for the file.

17 CLOSE ERROR

The directory structure could not be flushed for the Btrieve file. Either the Record Manager cannot close the file or a new page was added to the file and the Record Manager cannot close and reopen the file to update the directory structure. Check for a damaged disk. This status may also occur if the application writes on the position block allocated for the file.

18 DISK FULL

The disk is full and will not allow the file to be expanded to accommodate the insertion. Either erase any unnecessary files or use EXTEND to gain additional disk space.

19 UNRECOVERABLE ERROR

An unrecoverable error has occurred. File integrity cannot be guaranteed. Either use RECOVER to retrieve the damaged file's data records in a sequential file, or replace the Btrieve file with its most recent backup.

20 RECORD MANAGER INACTIVE

A request has been made before BREQUEST or the Record Manager has been started.

21 KEY BUFFER TOO SHORT

The key buffer parameter is not long enough to accommodate the key field for the access path requested. Verify that the length of the key buffer equals the defined length of the key specified by the key number parameter.

22 DATA BUFFER LENGTH

The data buffer parameter is not long enough to accommodate the length of the data record defined when the file was created. Verify that the length of the data buffer is as least as long as the file's defined record length.

- For *Get* or *Step* operations, if the data buffer is too short to contain the fixed length portion of the record, Btrieve will not return any data to the data buffer. If the record is a variable length record, and the data buffer is too short to contain the entire variable length portion of the record, Btrieve will return as much data as it can and a status 22, indicating that it could not return the entire record.
- For the *Insert* operation, Btrieve will not insert the record if the data buffer is shorter than the fixed length portion of the record.
- For the *Update* operation, if the data buffer is too short to contain the fixed length portion of any record, Btrieve will not update the record.
- For the *Create*, *Stat*, and *Create Supplemental Index* operations, a status 22 indicates that the data buffer is not long enough to contain all the file and key specifications, and the alternate collating sequence definition, if needed.

23 POSITION BLOCK LENGTH

The position block parameter is not 128 bytes long.

24 PAGE SIZE ERROR

The page size must be a multiple of 512 bytes but must be no larger than 4096 bytes. During a CREATE operation, page size is the first file specification Btrieve checks, and a status of 24 may indicate an invalid data buffer.

25 CREATE I/O ERROR

The filename specified could not be created. Possible causes are a full disk directory, a full disk, or a write-protected disk. If you are creating a file over an existing file, Btrieve can return a status code 25 because the existing file is open or is flagged transactional.

26 NUMBER OF KEYS

For standard Btrieve files with a page size of 512 bytes, the number of key segments for all key fields specified must be between 1 and 8. For larger page sizes, the number of key segments for all key fields must be between 1 and 24. You must define at least one key without the null attribute.

27 INVALID KEY POSITION

The key field position specified must not exceed the defined record length for the file. Either the key position is greater than the record length or the key position plus the key length exceeds the record length. For key-only files, the key must begin in the first byte of the record (position 1).

28 INVALID RECORD LENGTH

The record length specified (plus overhead for duplicates) is greater than the page size minus 6, or is less than 4 bytes long.

29 INVALID KEY LENGTH

The key length specified must be greater than 0 and cannot exceed 255. The length of a binary key must be even. Btrieve requires that each key page in the file be large enough to hold at least eight keys.

If the file's page size is too small to accommodate eight occurrences of the specified key length (plus overhead), either increase the file's page size or decrease the key length.

30 NOT A BTRIEVE FILE

The file name specified is not a valid Btrieve data file. Either the file was not created by Btrieve or it was created by an earlier version ofBtrieve. Use the program CONVERT4.EXE on the Btrieve Diskette to convert extended files created by Btrieve v3.x to the current format.

Another possibility is that the first page of the file, which contains the File Control Record, is damaged.

31 FILE ALREADY EXTENDED

The file name specified is already extended. A file can be extended only once.

32 EXTEND I/O ERROR

The file cannot be extended. Possible causes are that the disk directory is full, the disk is full, or the disk is write-protected.

34 INVALID EXTENSION NAME

The filename specified for the extended partition is not valid.

35 DIRECTORY ERROR

An error occurred while switching between the current directory and the directory which contains the Btrieve file. Either the current directory or the Btrieve file directory is invalid.

36 TRANSACTION ERROR

A Begin Transaction operation cannot be performed because no transactions were specified when BSERVER.VAP was initialized.

37 TRANSACTION IS ACTIVE

A Begin Transaction was issued while another transaction was active at that station. Transactions cannot be nested.

38 TRANSACTION CONTROL FILE I/O ERROR

An error occurred when the Record Manager tried to write to the transaction control file. Possible causes are that the disk is full, the disk is write-protected, or the transaction control file (which is created when the Record Manager is loaded) has been deleted.

39 END/ABORT TRANSACTION ERROR

An End or Abort Transaction operation was issued without a corresponding Begin Transaction operation.

40 TRANSACTION MAX FILES

An attempt was made to update more than the maximum number of files allowed within a transaction. The maximum number of different files that may be updated during a logical transaction is 12.

41 OPERATION NOT ALLOWED

Some operations are not allowed under certain operating conditions. For example, Btrieve will return this status if you attempt to perform a Step, Update, or Delete operation on a key-only file, or a Get operation on a data-only file.

Also, certain operations are prohibited during transactions because they have too great an effect on the pre-image file or on Btrieve's performance. These operations include Close, Set or Clear Owner, Extend, Create Supplemental Index, and Drop Supplemental Index. In addition, during a transaction you can only open files in Read-only mode.

42 INCOMPLETE ACCELERATED ACCESS

An attempt was made to open a file that was previously accessed in accelerated mode and never successfully closed. The file's integrity cannot be ensured. Either use RECOVER to build a new file or write your own recovery program that opens the file in recover mode and uses Step Direct to retrieve the data records.

43 INVALID RECORD ADDRESS

The record address specified on a Get Direct is invalid. Either the address is outside of the file's bounds, it is not on a record boundary within a data page, or it is not on a data page. The 4-byte address you specify on Get Direct should be one that was obtained by a Get Position operation.

44 NULL KEY PATH

An attempt has been made to use Get Direct to establish an access path for a key whose value is null in the corresponding record. Btrieve cannot establish positioning based on a null key value.

45 INCONSISTENT KEY FLAGS

The key flags specification on a Create operation is inconsistent. If a key has multiple segments, the duplicate, modifiable, and null attributes should be the same for each segment in the key.

46 ACCESS TO FILE DENIED

Your application opened a file in read-only mode and attempted to perform an Update, Delete, or Insert on that file. Inconsistent files that are opened in recover mode can be read but not updated. You should build a new file using either the RECOVER utility or the Step Direct operation.

Another possible cause is that the owner name required for updates was not specified correctly when your application opened the file.

47 MAXIMUM OPEN FILES

When a file is opened in accelerated mode, the Record Manager reserves one of its cache buffers for the file. The number of files opened in accelerated mode cannot exceed the number of buffers available in Btrieve's cache. Btrieve always reserves five empty buffers for index manipulation. Reconfigure the Record Manager with a smaller page size parameter to allocate more buffers.

48 INVALID ALTERNATE SEQUENCE DEFINITION

Btrieve returns this status if the first byte of an alternate collating sequence definition, the identification byte, does not contain the hexadecimal value AC.

49 KEY TYPE ERROR

You attempted to create a file or a supplemental index with an invalid extended key type, or you attempted to assign an alternate collating sequence to a binary key or key segment. You can only assign an alternate collating sequence to a string, lstring, or zstring type key.

Btrieve will also return this status if you defined a supplemental index requiring an alternate collating sequence, and no alternate collating sequence definition exists either in the file or in the key definition passed in the data buffer.

50 OWNER ALREADY SET

An attempt has been made to perform a set owner operation on a file that already has an owner. Use the clear owner operation to remove the previous owner before setting a new one.

51 INVALID OWNER

There are two possible causes for this status code.

- If your application receives this status code after a Set Owner operation, the owner names specified in the key buffer and data buffer do not match.

- If your application receives this status code after an Open operation, the file you are opening has had an owner name assigned to it. Your application must specify the correct owner name in the data buffer.

52 ERROR WRITING CACHE

While trying to make a cache buffer available, Btrieve attempted to write data to a disk from a file that was previously opened in accelerated mode. The operating system returned an I/O error during the write.

53 INVALID INTERFACE

An application attempted to access a file containing variable length records with a language interface from Btrieve v3.15 or earlier. To access files with variable length records, you must use a v4.xx interface.

54 VARIABLE PAGE ERROR

During a Step Direct operation Btrieve could not read all or part of the variable length portion of a record. In this case, Btrieve returns as much data as possible to your application. This error usually indicates file damage to one or more pages in the file.

55 AUTOINCREMENT ERROR

The application attempted to specify either the segmented or duplicates attribute for an autoincrement type key. An autoincrement key cannot be part of another key, and cannot allow duplicates.

56 INCOMPLETE INDEX

A supplemental index is damaged. This can occur if a Create Supplemental Index operation or a Drop Supplemental Index operation is interrupted and does not run to completion. Perform a Drop Supplemental Index operation to completely remove the index from the file.

58 COMPRESSION BUFFER TOO SHORT

The application attempted to read or write a record that is longer than the value you specified for the size of the compression buffer. Reconfigure BSERVER, specifying a higher value for the "Maximum Compressed Record Size" option, and restart the network.

59 FILE ALREADY EXISTS

Btrieve will return this status for a Create operation if you specified –1 in the key number parameter and the name of an existing file in the key buffer parameter.

80 CONFLICT

The Update or Delete operation cannot be performed because the record has been changed by another station since this station read it. Reread the record to perform the operation.

81 LOCK ERROR

This error can result from any one of three conditions:

- The Btrieve lock table is full. Specify a larger value for the locks parameter of BSERVER.
- The lock function call to the operating system failed.
- You attempted to unlock one record that was locked with a multiple record lock, and the record position stored in the data buffer did not correspond with any record that was locked in that file.

82 LOST POSITION

When performing a Get Next or Get Previous on a key with duplicates, an attempt has been made to retrieve a record that has been deleted or whose key value has been modified by another station. Re-establish positioning using a Get Equal or a Get Direct operation.

83 READ OUTSIDE TRANSACTION

An attempt has been made to delete or update a record within a transaction but the record was not read within the transaction. If you are going to update or delete a record within a transaction, you must read the record within the transaction to ensure you have first obtained exclusive access to the data.

84 RECORD IN USE

The application attempted to lock a record that is currently locked by another handle, or the application attempted to access a file within a transaction while another station held active record locks in that file.

85 FILE IN USE

The application attempted to open a file, lock a record, or access a record while another handle holds the transaction lock on the file or has the file open in accelerated mode.

86 FILE TABLE FULL

BSERVER's file table is full. Specify a larger number for the files parameter for BSERVER.

87 HANDLE TABLE FULL

BSERVER's handle table is full. Specify a larger number for the handles parameter for BSERVER. Another possibility is that NetWare cannot assign a handle to the file.

88 INCOMPATIBLE MODE ERROR

The application attempted to open a file in an incompatible mode. If the first handle to access a file opens it in accelerated mode, all others must open it in accelerated mode. The opposite is true for opens made in non-accelerated mode.

90 REDIRECTED DEVICE TABLE FULL

Either the redirection table or the server routing table is full. This error can occur if you attach to additional servers or map to more drives after you have loaded BREQUEST. Reload BREQUEST, specifying a larger number for the /R or /S option.

This error can also occur if you are attached to 8 servers, and then unattach a particular server and attach to a different server.Once a workstation has attached to a server, BREQUEST will not remove its name from the server table.

91 SERVER ERROR

BREQUEST cannot establish a session with the server. Either BSERVER has not been started for the device controlling the requested file, or the server is not active. Verify that BSERVER is active on the server to which the device has been redirected, and that the server is active.

92 TRANSACTION TABLE FULL

The number of active transactions you specified when you loaded BSERVER has been exceeded. Specify a larger number for the transactions parameter for BSERVER.

93 INCOMPATIBLE LOCK TYPE

The application attempted to mix single record locks (+100/+200) and multiple record locks (+300/+400) in the same file at the same time. All locks of one type must be released before a lock of the other type can be executed.

94 PERMISSION ERROR

A user attempted to open or create a file in a directory where he or she does not have those rights. Btrieve does not override the network privileges assigned to users.

95 SESSION NO LONGER VALID

A previously established session is no longer active due to an error either at the workstation or the file server, or on the network. Verify that the file server is still attached and then reload BREQUEST at the workstation.

96 COMMUNICATIONS ENVIRONMENT ERROR

The SPX connection table is full. Reload SPX, specifying a higher value for the connection table. Refer to the *NetWare Supervisor Reference* manual for more information.

97 DATA MESSAGE TOO SMALL

The application attempted to read or write a record which is longer than the value you specified with the record length parameter of BSERVER or the data message option of BREQUEST. Determine the correct maximum record length required by your application and reload the appropriate program, specifying a larger value.

- For an *Update*, *Insert*, or *Create* operation, the application will receive this error status if the data buffer length it specifies for the record exceeds the length specified for either the /D option of BREQUEST or record length parameter of BSERVER.
- For a *Get*, *Step*, or *Stat* operation, the application can receive this error status if the value specified for the data message parameter is shorter than the length of the data Btrieve returns, regardless of the data buffer length specified in the program.

98 INTERNAL TRANSACTION ERROR

An error has occurred during an operation within a transaction. All Insert, Update, and Delete operations since the last Begin Transaction operation have been rolled back. You should issue an Abort Transaction operation (21) to complete the rollback and exit the transaction.

BREQUEST STATUS CODES (OS/2)

BREQUEST.DLL can return the following run-time status codes at an OS/2 workstation.

2001 INSUFFICIENT MEMORY

BREQUEST cannot allocate enough memory for the parameters specified with the REQPARMS environment variable. Either reduce the size of the /D: option, or reduce the size of other memory resident routines loaded prior to BREQUEST.

2002 PARAMETER INVALID OR OUT OF RANGE

One of the parameters specified with the REQPARMS environment variable is either invalid (such as /P: instead of /D:) or the value specified for a parameter is out of range. Check the SET REQPARMS statement to make sure that it is correct.

2003 NO LOCAL ACCESS ALLOWED

The application attempted to access a file stored on a local drive. The version of BTRCALLS.DLL installed at the workstation does not allow access to local files.

BREQUEST ERROR MESSAGES (DOS)

BREQUEST.EXE can return the following error messages when it loads at a DOS workstation.

BSERVER NOT LOADED

The BSERVER program must be active at the server before BREQUEST can access files at that machine.

TOO MANY FILE SERVERS ATTACHED

The maximum number of file servers to which a workstation can attach is eight.

INCORRECT NETWARE VERSION

The NetWare shell for Advanced NetWare v2.1*x* or above must be installed at the workstation.

INCORRECT PARAMETER

An illegal parameter was specified. Refer to the section on starting BREQUEST for a list of valid parameters.

INSUFFICIENT MEMORY

BREQUEST cannot allocate enough memory for the parameters specified. Make sure that the workstation has enough memory to load all the programs it requires.

MUST HAVE DOS 2.00 OR GREATER

BREQUEST requires that DOS v2.00 or above be loaded at a DOS workstation.

PROGRAM ALREADY LOADED

BREQUEST is already loaded into memory.

REDIRECTION LIST NOT LARGE ENOUGH

BREQUEST cannot store all the redirected devices in its redirection list. Increase the value for the /R parameter.

SERVER ROUTING LIST IS NOT LARGE ENOUGH

The workstation is attached to more servers than were specified with the /S start-up option. Reload BREQUEST, specifying a higher number for the /S option.

SPX IS NOT LOADED

The NetWare SPX communications software must be loaded before an application can access BREQUEST.

BSERVER AND BROUTER ERROR MESSAGES

If either BSERVER or BROUTER encounters an error during initialization, it will display a message on the screen and will not load. You may receive the following messages when the VAPs load at the file server.

INSUFFICIENT MEMORY FOR PARAMETERS SPECIFIED

BSERVER cannot allocate the minimum number of buffers required. Reduce the size of the page size option.

Another possibility is that your server does not have enough available memory to load BSERVER at the requested size. You may need to add more memory to your server.

UNABLE TO ACCESS FILE FOR TRANSACTION RECOVERY

One possibility is that Btrieve cannot open one of the files involved in an incomplete transaction. Verify that all files involved in the transaction are online and restart the network.

Another cause for this message is that an I/O error occurred when Btrieve attempted to read or write one of the files. In order to recover an incomplete transaction, Btrieve must read and write the header record in each of the files involved in the transaction. The file may have been damaged. Check for media failure and replace all Btrieve files in the transaction with their most recent backup.

UNABLE TO ALLOCATE BROUTER BUFFER MEMORY

BROUTER cannot allocate the amount of memory specified for the data message length option. Use BSETUP to reconfigure and reinstall the Btrieve VAPs, specifying a smaller value for the data message length.

BUTIL ERROR MESSAGES

The BUTIL.EXE utility returns text messages if it encounters errors. The error messages are printed to STDOUT. You can use the DOS redirection symbol (>) to redirect them to a different device.

Most of the Butil error messages are self-explanatory. The messages listed here are those that may require additional explanation.

ERROR ACCESSING ALTERNATE SEQUENCE FILE

Butil received an error while it was reading the alternate collating sequence file you named with the "name=" keyword. This usually indicates an unexpected end-of-file.

ERROR ACCESSING DESCRIPTION FILE. EXPECTED KEYWORD *keyword* IN KEY DESCRIPTOR *n*

The description file contains a keyword or a description Butil does not recognize, or a definition in your description file is inconsistent. All keywords in the file must be spelled correctly, must be in lower case, and must appear in the correct order. Refer to the section "Rules for Description Files" in Chapter 4 for more information.

ERROR ACCESSING SEQUENTIAL FILE

Butil encountered an unexpected end-of-file on a read operation, or an error occurred while reading from or writing to the sequential file.

IMPROPER COMMAND LINE

You entered an improper switch parameter (i.e., –P instead of –O) or the command has too many parameters.

INVALID KEY TYPE

You specified an invalid key type in the description file.

INVALID LOAD FILE FORMAT. END OF RECORD MARKER NOT FOUND

Butil encountered an invalid record terminator in the sequential load file. Butil expects a carriage return/line feed at the end of each record in a load file. This error usually occurs because the length specified at the beginning of the sequential record is incorrect.

SIZE FOR *keytype* TYPE IS INVALID

Certain key types have restrictions on their length. You specified an invalid length for a key definition in the description file.

UNABLE TO CREATE/OPEN SEQUENTIAL FILE

Btrieve could not open the sequential file you specified for a LOAD operation. Receiving this error during a SAVE or RECOVER operation indicates that Btrieve could not create the sequential file you specified.

UNABLE TO OPEN ALTERNATE SEQUENCE FILE

Butil could not open the file you specified for the alternate collating sequence on a CREATE operation. Make sure the file name you specified is correct.

UNABLE TO OPEN DESCRIPTION FILE

Butil could not locate a file with the name you specified for the description file. If your description file is not in the current directory, you must specify a pathname.

APPENDIX C: PASCAL EXAMPLES

The following examples illustrate how to call Btrieve from a Pascal application. A different example is presented for each Btrieve operation.

PASCAL ABORT TRANSACTION

In the following example, an application must insert a record into an order header file and insert two records into an order detail file to record the sale of a box of diskettes and a ribbon to one customer. If an error occurs on any of the operations, the transaction is aborted so that the database remains consistent.

```
type
  ORDER_HDR = record
      case integer of
          1:  (HDR_ORD_NUM   :     integer;
               HDR_CUST      :     integer;
               HDR_DATE      :     string(6));
          2:  (ENTIRE        :     string(10));
      end;
  ORDER_DET = record
      case integer of
          1:  (DET_ORD_NUM   :     integer;
               DET_PART      :     integer;
               DET_QUAN      :     integer);
          2:  (ENTIRE        :     string(6));
      end;
var
  DET_BUF        : ORDER_DET;
  DET_KEY        : string(2);
  DET_LEN        : integer;
  DET_POS        : string(128);
  DUMY_BUF       : string(128);
  DUMY_LEN       : integer;
  HDR_BUF        : ORDER_HDR;
  HDR_KEY        : string(2);
  HDR_LEN        : integer;
  HDR_POS        : string(128);
  STATUS         : integer;
```

```
begin
{ Begin the transaction.                                                   }
STATUS := BTRV (B_BEGIN, DUMY_BUF, DUMY_BUF, DUMY_LEN, DUMY_BUF, 0);
  if STATUS <> 0 then
   begin
      writeln (OUTPUT, 'Error beginning transaction.Status = ', STATUS):
      return;
   end;
{     Insert the header record.                                            }
with HDR_BUF do
  begin
      HDR_ORD_NUM := 236;                                       {Order #236}
      HDR_CUST := 201;                                      {Customer #201}
      HDR_DATE := '061583';                                   {Date of sale}
  end;
HDR_LEN  := sizeof (HDR_BUF);
STATUS := BTRV (B_INSERT, HDR_POS, HDR_BUF.ENTIRE, HDR_LEN, HDR_KEY, 0);
  if STATUS <> 0 then
  begin
      writeln (OUTPUT, 'Error inserting header record.Status = ', STATUS);
      STATUS := BTRV (B_ABORT, DUMY_BUF, DUMY_BUF,DUMY_LEN, DUMY_BUF, 0);
      return;
  end;
{ Insert two detail records.                                               }
with DET_BUF do                                   {Insert the first detail record}
  begin
      DET_ORD_NUM := 236;                                       {Order #236}
      DET_PART := 1002;                          {Diskettes are part #1002}
      DET_QUAN := 1;                                      {Purchased 1 box};
  end;
DET_LEN  := sizeof (DET_BUF);
STATUS := BTRV ( B_INSERT, DET_POS, DET_BUF.ENTIRE, DET_LEN, DET_KEY, 0 );
if  STATUS <> 0 then
  begin
      writeln (OUTPUT, 'Error inserting detail record.Status = ', STATUS);
      STATUS := BTRV (B_ABORT, DUMY_BUF, DUMY_BUF, DUMY_LEN, DUMY_BUF, 0);
      return;
  end;
with DET_BUF do                                 {Insert the second detail record}
  begin
      DET_PART := 1024;                             {Ribbon is part #1024}
      DET_QUAN := 1;                                  {Purchased 1 ribbon};
  end;
```

```
STATUS := BTRV( B_INSERT, DET_POS, DET_BUF.ENTIRE, DET_LEN, DET_KEY, 0 );
  if STATUS <> 0 then
   begin
      writeln (OUTPUT, 'Error inserting detail record.Status = ', STATUS);
      STATUS := BTRV(B_ABORT, DUMY_BUF, DUMY_BUF, DUMY_LEN, DUMY_BUF, 0);
      return;
   end;
{ End the transaction.                                                  }
STATUS := BTRV (B_END, DUMY_BUF, DUMY_BUF, DUMY_LEN, DUMY_BUF, 0 );
  if STATUS <> 0 then
      begin
          writeln (OUTPUT, 'Error ending transaction.Status = ', STATUS);
          return;
      end;
```

PASCAL BEGIN TRANSACTION

See Abort Transaction and End Transaction.

PASCAL CLOSE

The following example illustrates the code required to close a Btrieve file from Pascal.

```
var
  BUF_LEN       : integer;
  DATA_BUF : string(80);
  KEY_BUF       : string(30);
  POS_BLOCK   : string(128);
  STATUS        : integer;

begin
STATUS := BTRV (B_CLOSE, POS_BLOCK, DATA_BUF, BUF_LEN, KEY_BUF, 0);
if STATUS <> 0 then writeln (OUTPUT, 'Btrieve status = ', STATUS);
end.
```

PASCAL CREATE

In the following example, an application creates a Btrieve file with two keys. Key 0 is an integer key, 2-bytes long. Key 1 allows duplicates, is modifiable, and consists of two segments. The first segment is a 2-byte string, to be sorted in descending order, and the second is a zero terminated 30-byte string. The records have a fixed length of 80.

```
const
  B_CREATE     = 14;
  DUPLICATES   = 1;
  MODIFIABLE   = 2;
  SEGMENTED    = 16;
  DESCENDING   = 64;
  EXTTYPE      = 256;
  BINTEGER     = 1;
  BSTRING      = 0;
  BZSTRING     = 11;

type
  byte_type = 0..255;
  KEY_SPEC = record
      KEY_POS     :   integer;
      KEY_LEN     :   integer;
      KEY_FLAGS   :   integer;
      NOT_USED    :   string(4);
      KEY_RSV     :   array [1..6] of byte_type;          { IBM Pascal word aligns }
  end;
FILE_SPEC = record
  case integer of
      1:  (REC_LEN      : integer;
           PAGE_SIZE    : integer;
           NDX_CNT      : integer;
           NOT_USED     : string(4);
           VARIABLE     : integer;
           RESERVED     : string(2);
           PRE_ALLOC    : integer;
           KEY_BUF      : array [0..2] of KEY_SPEC);
      2:  (SPEC_BUF     : integer);
  end;
var
  BUF_LEN        : integer;
  FILE_BUF       : FILE_SPEC;
  FILE_NAME      : string 17;
  POS_BLK        : string(128);
  STATUS         : integer;
```

```
begin
with FILE_BUF do
  begin
      REC_LEN := 80;
      PAGE_SIZE := 1024;
      VARIABLE := 0;
      NDX_CNT := 2;
      PRE_ALLOC := 0;
      KEY_BUF[0].KEY_POS := 1;
      KEY_BUF[0].KEY_LEN := 2;
      KEY_BUF[0].KEY_FLAGS := EXTTYPE;
      KEY_BUF[0].KEY_RSV[1] := BINTEGER;
      KEY_BUF[1].KEY_POS := 3;
      KEY_BUF[1].KEY_LEN := 2;
      KEY_BUF[1].KEY_FLAGS :=
        DUPLICATES + MODIFIABLE + SEGMENTED + EXTTYPE + DESCENDING;
      KEY_BUF[1].KEY_RSV[1] := BSTRING;
      KEY_BUF[2].KEY_POS := 5;
      KEY_BUF[2].KEY_LEN := 30;
      KEY_BUF[2].KEY_FLAGS := DUPLICATES + MODIFIABLE + EXTTYPE;
      KEY_BUF[2].KEY_RSV[1] := BZSTRING;
  end;
FILE_NAME := '\DATA\CREATE.TST ';
BUF_LEN := sizeof (FILE_BUF);
STATUS := BTRV (B_CREATE, POS_BLK, FILE_BUF.SPEC_BUF, BUF_LEN, FILE_NAME, 0);
if STATUS <> 0
  then
      writeln (OUTPUT, 'Error creating file. Status = ',STATUS)
  else
      writeln (OUTPUT, 'File created successfully.');
```

PASCAL CREATE SUPPLEMENTAL INDEX

In the following example, an application adds an index to a Btrieve file. The first segment of the key is a 10-byte string. It allows duplicates and is modifiable. The second segment is a 2-byte integer.

```
const
  B_CREIDX     = 31;
  DUPLICATES   = 1;
  MODIFIABLE   = 2;
  SEGMENTED    = 16;
  EXTTYPE      = 256;
  BINTEGER     = 1;
  BSTRING      = 0;

type
  byte_type  =   0..255;
  KEY_SPEC =     record
      KEY_POS        : integer;
      KEY_LEN        : integer;
      KEY_FLAGS      : integer;
      NOT_USED       : string(4);
      KEY_RSV        : array [1..6] of byte_type;          { IBM Pascal word aligns }
  end;
  ALL_KEY_SPEC = record
      case integer of
          1:  (KEY_BUF      : array [0..1] of KEY_SPEC);
          2:  (SPEC_BUF     : string (32));
      end;
var
  BUF_LEN       : integer;
  AKEY_BUF      : ALL_KEY_SPEC;
  FILE_NAME     : string (17);
  POS_BLK       : string (128);
  DUMY          : string (80);
  STATUS        : integer;

begin
BUF_LEN := 0;
FILE_NAME := '\DATA\CREATE.TST ';
STATUS := BTRV (B_OPEN, POS_BLK, DUMY, BUF_LEN, FILE_NAME, 0);
if STATUS <> 0 then
  writeln (OUTPUT, 'Error opening file. Status = ', STATUS);
```

```
with AKEY_BUF do
  begin
      KEY_BUF[0].KEY_POS := 40;
      KEY_BUF[0].KEY_LEN := 10;
      KEY_BUF[0].KEY_FLAGS := DUPLICATES + MODIFIABLE + SEGMENTED + EXTTYPE;
      KEY_BUF[0].KEY_TYPE := BSTRING;
      KEY_BUF[1].KEY_POS := 50;
      KEY_BUF[1].KEY_LEN := 2;
      KEY_BUF[1].KEY_FLAGS := DUPLICATES + MODIFIABLE + EXTTYPE;
      KEY_BUF[1].KEY_TYPE := BINTEGER;
  end
BUF_LEN := sizeof (AKEY_BUF);
STATUS := BTRV (B_CREIDX, POS_BLK, AKEY_BUF.SPEC_BUF, BUF_LEN, DUMY, 0);
if STATUS <> 0 then
  writeln (OUTPUT, 'Error adding supplemental index to file.Status = ', STATUS)
else
  writeln (OUTPUT,'Supplemental index added successfully.');
```

PASCAL DELETE

In the following example, a Pascal application for an airline company uses the Delete operation to reflect the fact that a passenger has cancelled a flight reservation.

```
type
RESERVATION = record
  case integer of
      1:  (FLIGHT_NO      :   string(3);
           PASSENGER      :   string(15);
           AMOUNT_PAID    :   string(6);
           ISSUE_DATE     :   string(6));
      2:  (ENTIRE         :   string(32));
  end;
var
  BUF_LEN      :   integer;
  DATA_BUF     :   RESERVATION;
  KEY_BUF      :   string(15);
  POS_BLOCK    :   string(128);
  STATUS       :   integer;
```

```
begin
BUF_LEN := sizeof (DATA_BUF);
KEY_BUF := 'Martin, Dave H.';
STATUS := BTRV (B_GET_EQ, POS_BLOCK, DATA_BUF.ENTIRE, BUF_LEN, KEY_BUF, 0);
if STATUS <> 0 then
  begin
      writeln (OUTPUT, 'Btrieve status = ', STATUS);
      return;
  end;
STATUS := BTRV (B_DELETE, POS_BLOCK, DATA_BUF.ENTIRE, BUF_LEN, KEY_BUF, 0);
if STATUS <> 0 then
  writeln (OUTPUT, 'Btrieve status = ', STATUS);
```

After the Delete operation, Btrieve's current position in the file is as follows:

326	Crawley, Joe J.	179.85	061582	
711	Howell, Susan	259.40	052382	← Previous record
326	Peters, John H.	445.80	061782	← Next record
840	White, Rosemary	397.00	060282	

↑
Access Path

PASCAL DROP SUPPLEMENTAL INDEX

In the following example, an application drops a supplemental index in a Btrieve file because the file no longer needs to be accessed by that index. The index number is 3.

```
const
  B_DROP    = 32;
  INV_KNUM = 6;

var
  BUF_LEN  :   integer;
  POS_BLK  :   string(128);
  STATUS   :   integer;
  DUMYDB   :   string(1);

begin
  BUF_LEN := 1;
  STATUS := BTRV (B_DROP, POS_BLK, DUMYDB, BUF_LEN, DUMYDB, 3);
  if STATUS = INV_KNUM then
      writeln (OUTPUT, 'Key number to drop is not a supplemental index.')
  else if  STATUS <> 0 then
      writeln (OUTPUT, 'Error dropping supplemental index.Status = ', STATUS);
```

PASCAL END TRANSACTION

The following application uses transaction control to ensure that one account in a ledger file is not debited unless another is also credited.

```
type
  LEDGER_KEY = record
      case integer of
          1:  (KEY_VAL  : integer);
          2:  (KEY_STR  : string(2));
      end;
  LEDGER_REC = record
      case integer of
          1:  (ACCT_ID  :    integer;
               DESC     :    string(40);
               BALANCE  :    real);
          2:  (ENTIRE   :    string(46));
      end;
var
  BUF_LEN       :    integer;
  DATA_BUF      :    LEDGER_REC;
  DUMY_BUF      :    string(128);
  DUMY_LEN      :    integer;
  KEY_BUF       :    LEDGER_KEY;
  POS_BLK       :    string(128);
  STATUS        :    integer;

begin
{ Begin the transaction.                                                    }
STATUS := BTRV (B_BEGIN, DUMY_BUF, DUMY_BUF, DUMY_LEN, DUMY_BUF, 0);
  if  STATUS <> 0 then
      begin
          writeln (OUTPUT, 'Error beginning transaction.Status = ', STATUS);return;
      end;
{ Retrieve and update the cash account record.                              }
KEY_BUF.KEY_VAL := 101;                                {Cash is account #101}
BUF_LEN:= sizeof (DATA_BUF);
STATUS := BTRV (B_GET_EQ, POS_BLK, DATA_BUF.ENTIRE, BUF_LEN,
                         KEY_BUF.KEY_STR, 0);
  if STATUS <> 0 then
      begin
          writeln (OUTPUT, 'Error retrieving record. Status = ', STATUS);
          STATUS := BTRV (B_ABORT, DUMY_BUF, DUMY_BUF, DUMY_LEN, DUMY_BUF, 0);
          return;
      end;
```

```
DATA_BUF.BALANCE := DATA_BUF.BALANCE – 250;
STATUS := BTRV (B_UPDATE, POS_BLK, DATA_BUF.ENTIRE, BUF_LEN,
                KEY_BUF.KEY_STR, 0);
if  STATUS <> 0 then
  begin
     writeln (OUTPUT, 'Error updating record. Status = ', STATUS);
     STATUS := BTRV (B_ABORT, DUMY_BUF, DUMY_BUF, DUMY_LEN, DUMY_BUF, 0);
     return;
  end;
{  Retrieve and update the office expense account record                 }
KEY_BUF.KEY_VAL := 511;                      {Office expense is account #511}
STATUS := BTRV (B_GET_EQ, POS_BLK, DATA_BUF.ENTIRE, BUF_LEN,
                KEY_BUF.KEY_STR, 0);
if STATUS <> 0 then
  begin
     writeln (OUTPUT, 'Error retrieving record. Status = ', STATUS);
     STATUS := BTRV (B_ABORT, DUMY_BUF, DUMY_BUF, DUMY_LEN, DUMY_BUF, 0);
     return;
  end;
DATA_BUF.BALANCE := DATA_BUF.BALANCE + 250;
STATUS := BTRV (B_UPDATE, POS_BLK, DATA_BUF.ENTIRE, BUF_LEN,
                KEY_BUF.KEY_STR, 0);
if  STATUS <> 0 then
  begin
     writeln (OUTPUT, 'Error updating record. Status = ', STATUS);
     STATUS := BTRV (B_ABORT, DUMY_BUF, DUMY_BUF, DUMY_LEN, DUMY_BUF, 0);
     return;
  end;
{  End the transaction.                                                  }
STATUS := BTRV (B_END, DUMY_BUF, DUMY_BUF, DUMY_LEN, DUMY_BUF,0);
if STATUS <> 0 then
  begin
     writeln (OUTPUT, 'Error ending transaction.Status = ', STATUS);
     return;
  end.
```

PASCAL EXTEND

The following example illustrates how an application might use the Extend operation to expand a Btrieve file in order to gain more disk space.

```
type
ADDRESS_REC = record                                {Structure of address file entry}
      case integer of
          1:  (NAME      :   string(30);
               STREET    :   string(30);
               CITY      :   string(30);
               STATE     :   string(2);
               ZIP       :   string(5));
          2: (ENTIRE     :   string(98));
      end;
var
  BUF_LEN       :   integer;
  DATA_BUF      :   ADDRESS_REC;
  EXT_NAME      :   string(14);
  FILE_NAME     :   string(14);
  KEY_BUF       :   string(30);
  POS_BLOCK     :   string(128);
  STATUS        :   integer;

begin
BUF_LEN   := sizeof (DATA_BUF);
FILE_NAME := 'ADDRESS.BTR   ';
STATUS := BTRV (B_OPEN, POS_BLOCK, DATA_BUF.ENTIRE, BUF_LEN, FILE_NAME, 0);
if STATUS <> 0 then
  begin
      writeln (OUTPUT, 'Error opening file. Status = ',STATUS);
      return;
  end;
EXT_NAME := 'B:\ADDRESS.EXT ';
STATUS := BTRV (B_EXTEND, POS_BLOCK, DATA_BUF.ENTIRE, BUF_LEN, EXT_NAME, 0);
if STATUS <> 0 then
  begin
      writeln (OUTPUT, 'Error extending file. Status = ',STATUS);
      return;
  end;
STATUS := BTRV (B_CLOSE, POS_BLOCK, DATA_BUF.ENTIRE, BUF_LEN,  KEY_BUF,  0);
STATUS := BTRV (B_OPEN, POS_BLOCK, DATA_BUF.ENTIRE, BUF_LEN, FILE_NAME,  0);
if STATUS <> 0 then
  writeln (OUTPUT, 'Error reopening the file. Status = ', STATUS)
else
  writeln (OUTPUT, 'File reopened successfully.');
```

PASCAL GET DIRECT

The following example illustrates how an application can use the Get Direct operation to sort the records in a Btrieve file by an external index. (See the description of Get Position for an example of how to build the external index file.)

```
type
  ADDR_REC = record
      case integer of
          1:  (NAME       :   string(20);
               STREET     :   string(20);
               CITY       :   string(10);
               STATE      :   string(2);
               ZIP        :   string(5));
          2: (REC_POS     :   string(4));
          3: (ENTIRE      :   string(57));
      end;
type
  INDX_REC = record
      case integer of
          1:  (INDX_POS      :   string(4);
               INDX_STATE    :   string(2));
          2: (ENTIRE         :   string(6));
      end;

var
  FILE_DATA     :   ADDR_REC;
  FILE_LEN      :   integer;
  FILE_POS      :   string(128);
  INDX_DATA     :   INDX_REC;
  INDX_LEN      :   integer;
  INDX_POS      :   string(128);
  NAME_KEY      :   string(20);
  STATE_KEY     :   string(2);
  STATUS        :   integer;
```

```
begin
FILE_LEN:= sizeof (FILE_DATA);
INDX_LEN:= sizeof (INDX_DATA);
STATUS := BTRV (B_GET_LOW, INDX_POS, INDX_DATA.ENTIRE, INDX_LEN, STATE_KEY, 0);
while STATUS <> EOF_ERR do                                    {Read until end of file}
  begin
      if (STATUS <> 0) then
          begin
              writeln (OUTPUT, 'Error reading file. Status = ', STATUS);
              return;
          end;
      FILE_DATA.REC_POS := INDX_DATA.INDX_POS;
      STATUS := BTRV (B_GET_DIRECT, FILE_POS, FILE_DATA.ENTIRE, FILE_LEN,
                          NAME_KEY, 0);
      if (STATUS <> 0) then
          begin
              writeln (OUTPUT, 'Error reading record. Status = ', STATUS);
              return;
          end;
      with FILE_DATA do
          writeln (OUTPUT, NAME, STREET, CITY, STATE,ZIP);
          STATUS := BTRV (B_GET_NEXT, INDX_POS, INDX_DATA.ENTIRE,
                              INDX_LEN, STATE_KEY, 0);
  end;
```

PASCAL GET DIRECTORY

The following example illustrates how an application can use the Get and Set Directory operations to retrieve the current directory at the beginning of the program, and to restore it before terminating.

```
var
  DIR_PATH      :   string(64);
  DUMY_BUF      :   string(128);
  DUMY_LEN      :   integer;
  STATUS        :   integer;

begin
STATUS := BTRV (B_GET_DIR, DUMY_BUF, DUMY_BUF, DUMY_LEN, DIR_PATH, 0);
if STATUS <> 0 then
  begin
      writeln (OUTPUT, 'Error getting current dir. Status = ', STATUS);
      return;
  end;
STATUS := BTRV (B_SET_DIR, DUMY_BUF, DUMY_BUF, DUMY_LEN, DIR_PATH, 0);
if STATUS <> 0 then
  begin
      writeln (OUTPUT, 'Error restoring current dir. Status = ', STATUS);
      return;
  end;
```

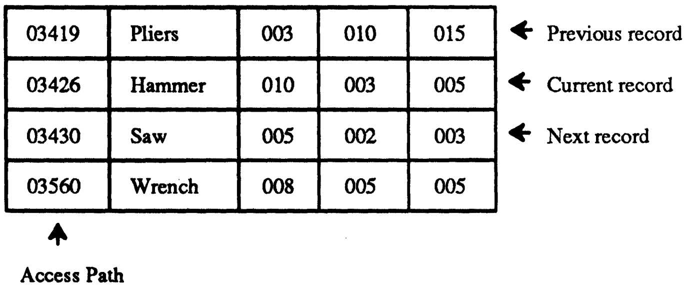
PASCAL GET EQUAL

In the following example, part number is a key in an inventory file. The application program retrieves the record containing inventory information for a particular part number with a single Get Equal operation, in order to determine whether or not to reorder that part number.

```
type
  INVENTORY = record
      case integer of
          1:  (PART_NUM         :      string(5);
              PART_DESC         :      string(10);
              QUAN_ON_HAND      :      string(3);
              REORDER_POINT     :      string(3);
              REORDER_QUAN      :      string(3));
          2:  (ENTIRE           :      string(28));
      end;
```

```
var
  DATA_BUF :   INVENTORY;
  DATA_LEN :   integer;
  KEY_BUF  :   string(5);
  POS_BLK  :   string(128);
  STATUS   :   integer;

begin
DATA_LEN:= sizeof (DATA_BUF);
KEY_BUF := '03426';                                     {Part number to find}
STATUS := BTRV (B_GET_EQ, POS_BLK, DATA_BUF.ENTIRE, DATA_LEN, KEY_BUF, 0);
if STATUS <> 0 then
  begin
      writeln (OUTPUT, 'Error reading file. Status = ',STATUS);
      return;
  end;
with DATA_BUF do
  if QUAN_ON_HAND < REORDER_POINT then
      writeln (OUTPUT, 'Time to order ',REORDER_QUAN,'units of ', PART_DESC);
```

The table below shows Btrieve's current position in the file after the Get Equal operation.



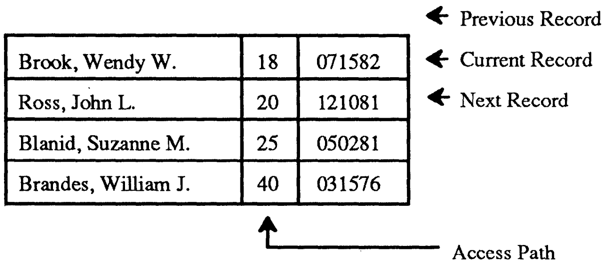
03419	Pliers	003	010	015
03426	Hammer	010	003	005
03430	Saw	005	002	003
03560	Wrench	008	005	005

PASCAL GET FIRST

The following example illustrates how an application might use the Get First operation to find the youngest employee in the company. Age is key number 2 in the employee file.

```
type
  EMP_REC = record
      case integer of
      1:  (NAME          :   string(20);
           AGE           :   string(2);
           HIRE_DATE     :   string(6));
      2:(ENTIRE          :   string(28));
  end;

var
  DATA_BUF    :   EMP_REC;
  DATA_LEN    :   integer;
  KEY_BUF     :   string(2);
  POS_BLOCK   :   string(128);
  STATUS      :   integer;

begin
DATA_LEN:= sizeof (DATA_BUF);
STATUS := BTRV (B_GET_FIRST, POS_BLOCK, DATA_BUF.ENTIRE, DATA_LEN, KEY_BUF, 2);
if STATUS <> 0 then
  writeln (OUTPUT, 'Error reading file. Status = ', STATUS)
else
  writeln (OUTPUT, 'Youngest employee is ', DATA_BUF.NAME);
```

After the Get First operation, Btrieve's current position in the file is as follows:



Brook, Wendy W.	18	071582
Ross, John L.	20	121081
Blanid, Suzanne M.	25	050281
Brandes, William J.	40	031576

PASCAL GET GREATER

The following example indicates how a Pascal application for an insurance company might use the Get Greater operation to determine which policyholders have more than 3 traffic violations. The number of traffic violations is key 2 in the policy file.

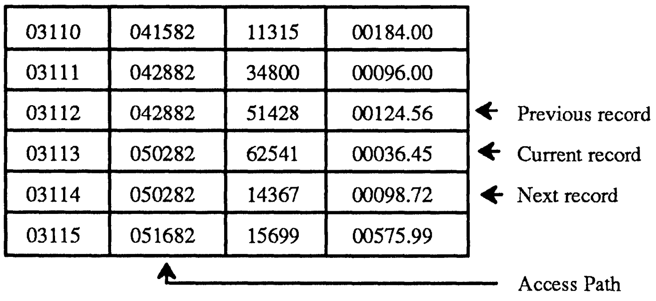
```
type
  POLICY = record
      case integer of
          1:  (POLICY_NUM :   string(10);
              NAME         :   string(20);
              EFFECT_DATE:   string(6);
              VIOLATIONS   :   string(2));
          2: (ENTIRE       :   string(38));
      end;


var
  DATA_BUF    :   POLICY;
  DATA_LEN    :   integer;
  KEY_BUF     :   string(2);
  POS_BLOCK   :   string(128);
  STATUS      :   integer;


begin
DATA_LEN:= sizeof (DATA_BUF);
KEY_BUF := '03';                                              {Start search above 3}
STATUS := BTRV (B_GET_GT, POS_BLOCK, DATA_BUF.ENTIRE, DATA_LEN, KEY_BUF, 2);
if STATUS <> 0 then
  begin
      writeln (OUTPUT, 'Error reading file. Status = ', STATUS);
      return;
  end;
while STATUS <> EOF_ERR do                                    {Read until end of file}
  begin
      writeln (OUTPUT, DATA_BUF.NAME, 'has ', DATA_BUF.VIOLATIONS,'traffic violations');
      STATUS := BTRV (B_GET_NEXT, POS_BLOCK, DATA_BUF.ENTIRE,
                          DATA_LEN, KEY_BUF,2);
      if (STATUS <> 0) and (STATUS <> EOF_ERR) then
          writeln (OUTPUT, 'Error reading file. Status = ', STATUS);
  end;
```

PASCAL GET GREATER OR EQUAL

If the date of a sale is a key in an invoice file, an application might use the Get Greater Or Equal operation to retrieve the invoice record for the first sale made in May, 1982.

```
type
  INVOICE = record
      case integer of
          1:  (INV_NUM        :      string(5);
               DATE_OF_SALE   :      string(6);
               CUST_NUM       :      string(5);
               TOTAL_PRICE    :      string(8));
          2:  (ALL            :      string(26));
      end;

var
  DATA_BUF     :   INVOICE;
  DATA_LEN     :   integer;
  KEY_BUF      :   string(6);
  POS_BLOCK    :   string(128);
  STATUS       :   integer;

begin
DATA_LEN:= sizeof (DATA_BUF);
KEY_BUF := '050182';                                   {Start search at May 1, 1982}
STATUS := BTRV (B_GET_GE, POS_BLOCK, DATA_BUF.ALL, DATA_LEN, KEY_BUF, 1);
if STATUS <> 0 then
  writeln (OUTPUT, 'Error reading file. Status = ', STATUS)
else
  writeln (OUTPUT, 'First sale in May was to ', DATA_BUF.CUST_NUM,'for',
          DATA_BUF.TOTAL_PRICE);
```

After the Get Greater Or Equal Operation, Btrieve's current position in the file is as follows:



03110	041582	11315	00184.00
03111	042882	34800	00096.00
03112	042882	51428	00124.56
03113	050282	62541	00036.45
03114	050282	14367	00098.72
03115	051682	15699	00575.99

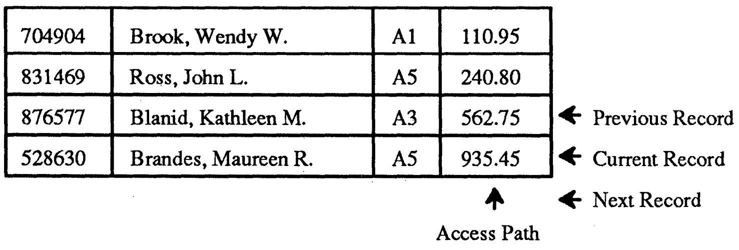
Previous record
Current record
Next record
Access Path

PASCAL GET LAST

The following example illustrates how an application might use the Get Last operation to determine which employee had the highest commission last month.

```
type
  EMP_REC = record
      case integer of
           1:  (EMP_NUM        :      string(6);
                EMP_NAME       :      string(20);
                EMP_DEPT       :      string(2);
                EMP_TOT_COM    :      string(6);
                EMP_CUR_COM    :      string(6));
           2:  (ENTIRE         :      string(40));
      end;

var
  BUF_LEN      :   integer;
  DATA_BUF     :   EMP_REC;
  KEY_BUF      :   string(6);
  POS_BLOCK    :   string(128);
  STATUS       :   integer;

begin
BUF_LEN:= sizeof (DATA_BUF);
STATUS := BTRV (B_GET_LAST, POS_BLOCK, DATA_BUF.ENTIRE, BUF_LEN, KEY_BUF, 1);
if STATUS <> 0 then
  writeln (OUTPUT, 'Error reading file. Status = ', STATUS)
else
  writeln (OUTPUT, 'Employee with highest commissions last month was', DATA_BUF.EMP_NAME);
```

After the Get Last operation, Btrieve's current positioning in the file is as follows:



704904	Brook, Wendy W.	A1	110.95	
831469	Ross, John L.	A5	240.80	
876577	Blanid, Kathleen M.	A3	562.75	← Previous Record
528630	Brandes, Maureen R.	A5	935.45	← Current Record
				← Next Record

PASCAL GET LESS THAN

The following Pascal example indicates how an application may use Get Less Than, followed by Get Previous, to find the names of all customers whose magazine subscriptions have less than three issues left before they run out. The number of issues remaining is key 2 in the subscription file.

```
type
  SUBSCRIPTION = record
      case integer of
          1:  (CUST_NAME            : string(20);
               DATE_SUBSCRIBED      : string(6);
               DATE_PAID            : string(6);
               ISSUES_PURCH         : string(3);
               ISSUES_REMAIN        : string(3));
          2:  (ENTIRE               : string(40));
      end;

var
  BUF_LEN      :    integer;
  DATA_BUF     :    SUBSCRIPTION;
  KEY_BUF      :    string(3);
  POS_BLOCK    :    string(128);
  STATUS       :    integer;

begin
BUF_LEN := sizeof (DATA_BUF);
KEY_BUF := '003';                                                {Start search below 3}
STATUS := BTRV (B_GET_LT, POS_BLOCK, DATA_BUF.ENTIRE, BUF_LEN, KEY_BUF, 2);
if STATUS <> 0 then
  begin
      writeln (OUTPUT, 'Error reading file. Status = ',STATUS);
      return;
  end;
while STATUS <> EOF_ERR do                                      {Read until start of file}
  begin
      writeln (OUTPUT, 'Send reorder form to ', DATA_BUF.CUST_NAME);
      STATUS := BTRV (B_GET_PREV, POS_BLOCK, DATA_BUF.ENTIRE, BUF_LEN,
                          KEY_BUF, 2);
        if (STATUS <> 0) and (STATUS <> EOF_ERR) then
           begin
               writeln (OUTPUT, 'Error reading file. Status = ', STATUS);
               return;
           end;
  end;
```

PASCAL GET LESS THAN OR EQUAL

In the following example, an application uses the Get Less Than Or Equal operation to retrieve the first house that falls within a prospective customer's price limit of $110,000.

```
type
  HOME = record
      case integer of
          1:  (PRICE              :     string(7);
               ADDRESS            :     string(20);
               SQUARE_FEET        :     string(6);
               YEAR_BUILT         :     string(4));
          2:  (ALL                :     string(38));
      end;

var
  DATA_BUF    :   HOME;
  DATA_LEN    :   integer;
  POS_BLOCK   :   string(128);
  PRICE_KEY   :   string(7);
  STATUS      :   integer;

begin
DATA_LEN  := sizeof (DATA_BUF);
PRICE_KEY := '0110000';                                   {Start search at 110,000}
STATUS := BTRV (B_GET_LE, POS_BLOCK, DATA_BUF.ALL, DATA_LEN, PRICE_KEY, 0);
if STATUS <> 0 then
  writeln (OUTPUT, 'Error reading file. Status = ', STATUS)
else
  writeln (OUTPUT, 'The home at ', DATA_BUF.ADDRESS,'sells for', DATA_BUF.PRICE);
```

After the Get Less Than or Equal operation, Btrieve's current position is as follows:

0050000	330 N. 31st	002200	1960	
0055000	11132 Maple Ave.	002000	1965	
0070000	624 Church Street	002300	1968	← Previous Record
0105000	3517 N. Lakes Avenue	002500	1975	← Current Record
0220000	4500 Oceanfront Ave.	003000	1980	← Next Record

Access Path →

PASCAL GET NEXT

In the following example, an application uses a Get Next operation to generate a set of mailing labels sorted according to zip code. The zip code (ZIP) is key 1 in the file.

```
type
  ADDRESS_REC = record
      case integer of
          1:  (NAME      :   string(20);
               STREET    :   string(20);
               CITY      :   string(10);
               STATE     :   string(2);
               ZIP       :   string(5));
          2:  (ENTIRE    :   string(58));
      end;
var
  DATA_BUF     :   ADDRESS_REC;
  DATA_LEN     :   integer;
  POS_BLOCK    :   string(128);
  PRINTER      :   text;
  STATUS       :   integer;
  ZIP_KEY      :   string(5);
begin
ASSIGN (PRINTER,'LPT1');                         {Initialize output variable for printer}
REWRITE (PRINTER);
DATA_LEN:= sizeof (DATA_BUF);
STATUS := BTRV (B_GET_FIRST, POS_BLOCK, DATA_BUF.ENTIRE, DATA_LEN, ZIP_KEY, 1);
if STATUS <> 0 then
  begin
      writeln (OUTPUT, 'Error reading address file. Status = ', STATUS);
      return;
  end;
with DATA_BUF do
  while STATUS <> EOF_ERR do                                  {Read until end of  file}
      begin
          page (PRINTER);                                          {Start new label}
          writeln (PRINTER, NAME);                                      {Print name}
          writeln (PRINTER, STREET);                                  {Print street}
          writeln (PRINTER, CITY,', ',STATE,' ',ZIP);        {Print city and state}
          STATUS:=BTRV (B_GET_NEXT, POS_BLOCK, DATA_BUF.ENTIRE,
                              DATA_LEN, ZIP_KEY, 1);
          if (STATUS <> 0) and (STATUS <> EOF_ERR) then
              begin
                  writeln (OUTPUT,'Error reading address file.Status = ', STATUS);
                  return;
              end;
      end;
```

PASCAL GET POSITION

The following example illustrates how Get Position can be used to construct an external index for an existing Btrieve file. Once an external index is created, the application can read the external index file from lowest to highest and use Get Direct to sort the records in a Btrieve file by some field that was not originally defined as a key field.

```
type
  ADDR_REC = record
      case integer of
          1:  (NAME       :   string(20);
               STREET     :   string(20);
               CITY       :   string(10);
               STATE      :   string(2);
               ZIP        :   string(5));
          2:  (REC_POS :      string(4));
          3:  (ENTIRE     :   string(57));
      end;
type
  INDX_REC = record
      case integer of
          1:  (INDX_POS        :    string(4);
               INDX_STATE      :    string(2));
          2: (ENTIRE           :    string(6));
      end;
var
  FILE_DATA    :   ADDR_REC;
  FILE_LEN     :   integer;
  FILE_POS     :   string(128);
  INDX_DATA    :   INDX_REC;
  INDX_LEN     :   integer;
  INDX_POS     :   string(128);
  NAME_KEY     :   string(20);
  STATE_KEY    :   string(2);
  STATUS       :   integer;
```

```
begin
FILE_LEN:=sizeof (FILE_DATA);
INDX_LEN:=sizeof (INDX_DATA);
STATUS := BTRV (B_GET_FIRST, FILE_POS, FILE_DATA.ENTIRE, FILE_LEN, NAME_KEY, 0);
while STATUS <> EOF_ERR do   {Read until end of file}
  begin
     if (STATUS <> 0) then
         begin
             writeln (OUTPUT, 'Error reading file. Status = ', STATUS);
             return;
         end;
     INDX_DATA.INDX_STATE := FILE_DATA.STATE;
     STATUS := BTRV (B_GET_POS, FILE_POS, FILE_DATA.ENTIRE, FILE_LEN,
                         NAME_KEY, 0);
     INDX_DATA.INDX_POS := FILE_DATA.REC_POS;
     STATUS := BTRV (B_INSERT, INDX_POS, INDX_DATA.ENTIRE, INDX_LEN,
                         STATE_KEY, 0);
     if (STATUS <> 0) then
         begin
             writeln (OUTPUT, 'Error inserting record.Status = ', STATUS);
             return;
         end;
     STATUS := BTRV (B_GET_NEXT, FILE_POS, FILE_DATA.ENTIRE, FILE_LEN,
                         NAME_KEY, 0);
  end;
```

PASCAL GET PREVIOUS

The following example illustrates how an application can use Get Previous to list corporations and their total sales dollars for the year, beginning with the corporation having the highest sales and continuing in descending order of sales dollars. Total sales is key number 1 in the company file.

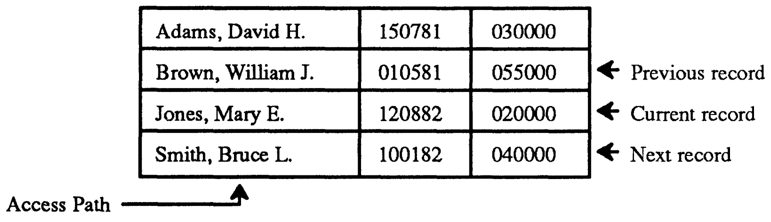
```
type
  COMPANY_REC = record
      case integer of
          1:  (NAME          :   string(30);
               TOTAL_SALES :   string(10));
          2:  (ALL           :   string(40));
      end;

var
  DATA_BUF    :   COMPANY_REC;
  DATA_LEN    :   integer;
  POS_BLOCK   :   string(128);
  SALES_KEY   :   string(10);
  STATUS      :   integer;

begin
DATA_LEN:= sizeof (DATA_BUF);
STATUS := BTRV (B_GET_LAST, POS_BLOCK, DATA_BUF.ALL, DATA_LEN, SALES_KEY, 1);
if STATUS <> 0 then
  begin
      writeln (OUTPUT, 'Error reading file. Status = ',STATUS);
      return;
  end;
while STATUS <> EOF_ERR do   {Read until end of file}
  begin
      writeln (OUTPUT, DATA_BUF.NAME, DATA_BUF.TOTAL_SALES);
      STATUS := BTRV (B_GET_PREV, POS_BLOCK, DATA_BUF.ALL, DATA_LEN,
                              SALES_KEY, 1);
      if (STATUS <> 0) and (STATUS <> EOF_ERR) then
          begin
              writeln (OUTPUT, 'Error reading file. Status = ', STATUS);
              return;
          end;
  end;
```

PASCAL INSERT

The following example shows how an application might use the Insert operation to add a new employee to the employee file.

```
type
  EMP_REC = record
      case integer of
          1:  (NAME          :    string(20);
               HIRE_DATE     :    string(6);
               ANNUAL_SAL    :    string(6));
          2:  (ENTIRE        :    string(32));
      end;

var
  DATA_BUF     :   EMP_REC;
  DATA_LEN     :   integer;
  KEY_BUF      :   string(20);
  POS_BLOCK    :   string(128);
  STATUS       :   integer;

begin
with DATA_BUF do                                                {Initialize data record}
  begin
      NAME := 'Jones, Mary E. ';
      HIRE_DATE := '120882';
      ANNUAL_SAL := '020000';
  end;
DATA_LEN:= sizeof(DATA_BUF);
STATUS := BTRV (B_INSERT, POS_BLOCK, DATA_BUF.ENTIRE, DATA_LEN, KEY_BUF, 0);
if STATUS <> 0 then
  writeln (OUTPUT, 'Btrieve status = ', STATUS);
```

After an Insert operation, Btrieve's current position in the file is as follows:



Adams, David H.	150781	030000
Brown, William J.	010581	055000
Jones, Mary E.	120882	020000
Smith, Bruce L.	100182	040000

PASCAL OPEN

The following example illustrates the code required to open a Btrieve file from Pascal.

```
var
  BUF_LEN     :  integer;
  DATA_BUF    :  string(92);
  FILE_NAME   :  string(20);
  POS_BLOCK   :  string(128);
  STATUS      :  integer;

begin
  FILE_NAME := 'C:\DATA\EMPLOYE.BTR ';
  STATUS := BTRV (B_OPEN, POS_BLOCK, DATA_BUF, BUF_LEN, FILE_NAME, 0);
  if STATUS <> 0 then
       writeln (OUTPUT, 'Btrieve status = ', STATUS);
end;
```

PASCAL SET DIRECTORY

In the following example, an application sets the current directory before performing an Open operation.

```
type
  ADDRESS_REC = record                                    {Structure of address file entry}
      case integer of
          1:  (NAME      :   string(30);
               STREET    :   string(30);
               CITY      :   string(30);
               STATE     :   string(2);
               ZIP       :   string(5));
          2:  (ENTIRE    :   string(98));
      end;
var
  BUF_LEN      :   integer;
  DATA_BUF     :   ADDRESS_REC;
  DIR_PATH     :   string(6);
  FILE_NAME    :   string(14);
  KEY_BUF      :   string(30);
  POS_BLOCK    :   string(128);
  STATUS       :   integer;

begin
BUF_LEN  := sizeof(DATA_BUF);
DIR_PATH := '\DATA ';
DIR_PATH[6] := chr(0);
STATUS := BTRV (B_SET_DIR, POS_BLOCK, DATA_BUF.ENTIRE, BUF_LEN, DIR_PATH, 0);
if STATUS <> 0 then
  begin
      writeln (OUTPUT, 'Unable to set current directory.Status = ', STATUS);
      return;
  end;
FILE_NAME := 'ADDRESS.BTR   ';
STATUS := BTRV (B_OPEN, POS_BLOCK, DATA_BUF.ENTIRE, BUF_LEN, FILE_NAME, 0);
if STATUS <> 0 then
  begin
      writeln (OUTPUT, 'Error opening file. Status = ', STATUS);
      return;
  end;
```

PASCAL STAT

In the following example, an application uses the Stat and Create operations to empty a Btrieve file.

```
type
  byte_type   =   0..255;
  KEY_SPEC =    record
      KEY_POS      :   integer;
      KEY_LEN      :   integer;
      KEY_FLAGS    :   integer;
      NOT_USED     :   string(4);
      KEY_RSV      :   array [1..6] of byte_type;          { IBM Pascal word aligns }
  end;
  FILE_SPEC = record
      case integer of
          1:  (REC_LEN      :   integer;
               PAGE_SIZE    :   integer;
               NDX_CNT      :   integer;
               NOT_USED     :   string(4);
               VARIABLE     :   integer;
               RESERVED     :   string(2);
               PRE_ALLOC    :   integer;
               KEY_BUF      :   array [0..2] of KEY_SPEC);
          2:  (SPEC_BUF     :   integer);
      end;

var
  DATA_BUF         :   string(44);
  DATA_BUF_LEN     :   integer;
  FILE_BUF         :   FILE_SPEC;
  FILE_BUF_LEN     :   integer;
  FILE_NAME        :   string(14);
  KEY_BUF          :   string(64);
  POS_BLK          :   string(128);
  STATUS           :   integer;

begin
DATA_BUF_LEN := sizeof(DATA_BUF);
FILE_BUF_LEN := sizeof(FILE_BUF);
FILE_NAME := 'LEDGER.BTR ';
STATUS := BTRV (B_OPEN, POS_BLK, DATA_BUF, DATA_BUF_LEN, FILE_NAME, 0);
if STATUS <> 0 then
  begin
      writeln (OUTPUT, 'Error opening file. Status = ',STATUS);
      return;
  end;
```

```
STATUS := BTRV (B_STAT, POS_BLK, FILE_BUF.SPEC_BUF, FILE_BUF_LEN, KEY_BUF, 0);
if STATUS <> 0 then
  begin
      writeln (OUTPUT, 'Error retrieving file stats. Status = ', STATUS);
      return;
  end;
STATUS := BTRV (B_CLOSE, POS_BLK, DATA_BUF, DATA_BUF_LEN, FILE_NAME, 0);
if STATUS <> 0 then
  begin
      writeln (OUTPUT, 'Error closing file. Status = ',STATUS);
      return;
  end;
FILE_BUF_LEN := sizeof(FILE_BUF);
STATUS := BTRV (B_CREATE, POS_BLK, FILE_BUF.SPEC_BUF, FILE_BUF_LEN,
                        FILE_NAME, 0);
if STATUS <> 0 then
  writeln (OUTPUT, 'Error recreating file. Status = ', STATUS)
else
  writeln (OUTPUT, 'File emptied successfully.');
```

PASCAL STEP FIRST

See Step Next.

PASCAL STEP LAST

See Step Previous.

PASCAL STEP NEXT

The following example illustrates how an application might use the Step First and Step Next operations to recover a file whose indexes have been damaged by a system failure.

```
type
  EMP_REC = record
      case integer of
          1:  (NUM       :   string(6);
               NAME      :   string(30);
               ADDR      :   string(50));
          3:  (ENTIRE    :   string(86));
  end;

var
  BUF_LEN  :   integer;
  OLD_BUF  :   EMP_REC;
  OLD_KEY  :   string(6);
  OLD_POS  :   string(128);
  NEW_BUF  :   EMP_REC;
  NEW_KEY  :   string(6);
  NEW_POS  :   string(128);
  STATUS   :   integer;

begin
BUF_LEN:= sizeof(OLD_BUF);
STATUS := BTRV (B_STEP_FST, OLD_POS, OLD_BUF.ENTIRE, BUF_LEN, OLD_KEY, 0);
while STATUS <> EOF_ERR do                                  {Read until end of file}
  begin
      if (STATUS <> 0) then
          begin
              writeln (OUTPUT, 'Error reading file. Status = ', STATUS);
              return;
          end;
      NEW_BUF.NUM := OLD_BUF.NUM;
      NEW_BUF.NAME := OLD_BUF.NAME;
      NEW_BUF.ADDR := OLD_BUF.ADDR;
      STATUS := BTRV (B_INSERT, NEW_POS, NEW_BUF.ENTIRE, BUF_LEN,
                      NEW_KEY, 0);
      if (STATUS <> 0) then
          begin
              writeln (OUTPUT, 'Error inserting record.Status = ', STATUS);
              return;
          end;
      STATUS := BTRV (B_STEP_NXT, OLD_POS, OLD_BUF.ENTIRE, BUF_LEN,
                      OLD_KEY, 0);
  end;
```

PASCAL STEP PREVIOUS

The following example illustrates how an application might use the Step Last and Step Previous operations to recover a file whose indexes have been damaged by a system failure.

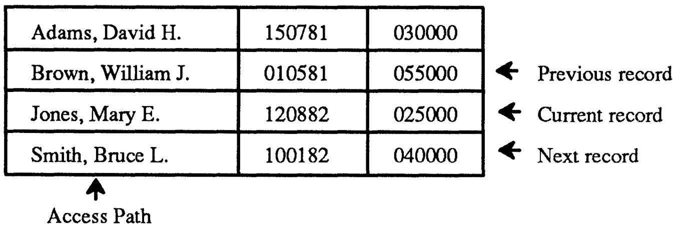
```
type
  EMP_REC = record
      case integer of
          1:  (NUM       :   string(6);
               NAME      :   string(30);
               ADDR      :   string(50));
          3:  (ENTIRE    :   string(86));
      end;

var
  BUF_LEN  :   integer;
  OLD_BUF  :   EMP_REC;
  OLD_KEY  :   string(6);
  OLD_POS  :   string(128);
  NEW_BUF  :   EMP_REC;
  NEW_KEY  :   string(6);
  NEW_POS  :   string(128);
  STATUS   :   integer;

begin
BUF_LEN:= sizeof(OLD_BUF);
STATUS := BTRV (B_STEP_LST, OLD_POS, OLD_BUF.ENTIRE, BUF_LEN, OLD_KEY, 0);
while STATUS <> EOF_ERR do                                    {Read until end of file}
  begin
      if (STATUS <> 0) then
          begin
              writeln (OUTPUT, 'Error reading file. Status = ', STATUS);
              return;
          end;
      NEW_BUF.NUM := OLD_BUF.NUM;
      NEW_BUF.NAME := OLD_BUF.NAME;
      NEW_BUF.ADDR := OLD_BUF.ADDR;
      STATUS := BTRV (B_INSERT, NEW_POS, NEW_BUF.ENTIRE, BUF_LEN,
                          NEW_KEY, 0);
      if (STATUS <> 0) then
          begin
              writeln (OUTPUT, 'Error inserting record.Status = ', STATUS);
              return;
          end;
      STATUS := BTRV (B_STEP_PREV, OLD_POS, OLD_BUF.ENTIRE, BUF_LEN,
                          OLD_KEY, 0);
  end;
```

PASCAL UPDATE

The following example shows how an application might use the Update operation to reflect the fact that an employee just received a raise.

```
type
  EMP_REC = record
      case integer of
          1:  (NAME         :   string(20);
               HIRE_DATE    :   string(6);
               ANNUAL_SAL   :   string(6));
          2:  (ALL          :   string(32));
      end;
var
  DATA_BUF     :   EMP_REC;
  DATA_LEN     :   integer;
  NAME_KEY     :   string(20);
  POS_BLK      :   string(128);
  STATUS       :   integer;
begin
DATA_LEN := sizeof(DATA_BUF);
NAME_KEY := 'Jones, Mary E.      ';
STATUS := BTRV (B_GET_EQ, POS_BLK, DATA_BUF.ALL, DATA_LEN, NAME_KEY, 0);
if STATUS <> 0 then
  begin
      writeln ('Btrieve status = ', STATUS);
      return;
  end;
DATA_BUF.ANNUAL_SAL := '025000';
STATUS := BTRV (B_UPDATE, POS_BLK, DATA_BUF.ALL, DATA_LEN, NAME_KEY, 0);
if STATUS <> 0 then
  writeln ('Btrieve status = ', STATUS);
```

After the Update operation, Btrieve's position in the file is as follows:



Adams, David H.	150781	030000	
Brown, William J.	010581	055000	← Previous record
Jones, Mary E.	120882	025000	← Current record
Smith, Bruce L.	100182	040000	← Next record
↑ Access Path			

APPENDIX D: COBOL EXAMPLES

The following examples illustrate how to call Btrieve from a COBOL application. A different example is presented for each Btrieve operation.

COBOL ABORT TRANSACTION

In the following example, an application must insert a record into an order header file and insert two records into an order detail file to record the sale of a box of diskettes and a ribbon to one customer. If an error occurs on any of the operations, the transaction is aborted so that the database remains consistent.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  B-ABORT           PIC 99 COMP-0 VALUE 21.
77  B-BEGIN           PIC 99 COMP-0 VALUE 19.
77  B-END             PIC 99 COMP-0 VALUE 20.
77  B-INSERT          PIC 99 COMP-0 VALUE 2.
77  B-OPEN            PIC 99 COMP-0 VALUE 0.
77  KEY-NUM           PIC 99 COMP-0 VALUE 0.
01  B-STATUS          PIC 99 COMP-0.
01  DSP-STATUS        PIC 99999.
01  HDR-BUFFER.
    02   HDR-ORD-NUM  PIC 99 COMP-0.
    02   HDR-CUST     PIC 99 COMP-0.
    02   HDR-DATE     PIC X(6).
01  DET-BUFFER.
    02   DET-ORD-NUM  PIC 99 COMP-0.
    02   DET-PART     PIC 99 COMP-0.
    02   DET-QUAN     PIC 99 COMP-0.
01  DET-KEY           PIC 99 COMP-0.
01  DET-LEN           PIC 99 COMP-0 VALUE 6.
01  DET-POS-BLK       PIC X(128).
01  DUMY-BUF          PIC X(128).
01  DUMY-LEN          PIC 99 COMP-0 VALUE 128.
01  HDR-KEY           PIC 99 COMP-0.
01  HDR-LEN           PIC 99 COMP-0 VALUE 10.
01  HDR-POS-BLK       PIC X(128).
```

```
PROCEDURE DIVISION.
BEGIN.
* Start the transaction.
*
     CALL 'BTRV' USING B–BEGIN, B–STATUS, DUMY–BUF, DUMY–BUF,
                              DUMY–LEN, DUMY–BUF, KEY–NUM.
     IF B–STATUS NOT = 0
           MOVE B–STATUS TO DSP–STATUS
           DISPLAY (5, 1) "Error beginning tran. Status = " DSP–STATUS
           STOP RUN.

* Insert the header record.
*
     MOVE 236 TO HDR–ORD–NUM.
     MOVE 201 TO HDR–CUST.
     MOVE "061583" TO HDR–DATE.
     CALL 'BTRV' USING B–INSERT, B–STATUS, HDR–POS–BLK,
                              HDR–BUFFER, HDR–LEN, HDR–KEY, KEY–NUM.
     IF B–STATUS NOT = 0
           MOVE B–STATUS TO DSP–STATUS
           DISPLAY (5, 1) "Error inserting headerrecord. Status =" DSP–STATUS
           CALL 'BTRV' USING B–ABORT, B–STATUS, DUMY–BUF,
                              DUMY–BUF, DUMY–LEN, DUMY–BUF, KEY–NUM.
           STOP RUN.

* Insert two detail records.
*
     MOVE 236 TO DET–ORD–NUM.
     MOVE 1002 TO DET–PART.
     MOVE 1 TO DET–QUAN.
     CALL 'BTRV' USING B–INSERT, B–STATUS, DET–POS–BLK,
                              DET–BUFFER, DET–LEN, DET–KEY, KEY–NUM.
     IF B–STATUS NOT = 0
           MOVE B–STATUS TO DSP–STATUS
           DISPLAY "Error inserting detail record.Status =" DSP–STATUS
           CALL 'BTRV' USING B–ABORT, B–STATUS, DUMY–BUF,
                              DUMY–BUF, DUMY–LEN, DUMY–BUF, KEY–NUM.

     STOP RUN.
     MOVE 1024 TO DET–PART.
     MOVE 1 TO DET–QUAN.

     CALL 'BTRV' USING B–INSERT, B–STATUS, DET–POS–BLK,
                              DET–BUFFER, DET–LEN, DET–KEY, KEY–NUM.
```

```
         IF B–STATUS NOT = 0
              MOVE B–STATUS TO DSP–STATUS
              DISPLAY "Error inserting detail record.Status =" DSP–STATUS
              CALL 'BTRV' USING B–ABORT, B–STATUS, DUMY–BUF,
                                DUMY–BUF, DUMY–LEN, DUMY–BUF, KEY–NUM.
         STOP RUN.

   *
   * End the transaction.
   *
         CALL 'BTRV' USING B–END, B–STATUS, DUMY–BUF, DUMY–BUF,
                                DUMY–LEN, DUMY–BUF, KEY–NUM.
         IF B–STATUS NOT = 0
              MOVE B–STATUS TO DSP–STATUS
              DISPLAY "Error ending tran. Status = " DSP–STATUS
              STOP RUN.
```

COBOL BEGIN TRANSACTION

See Abort Transaction and End Transaction.

COBOL CLOSE

The following example illustrates the code required to close a Btrieve file from COBOL.

```
   DATA DIVISION.
   WORKING–STORAGE SECTION.
   77  B–CLOSE             PIC 99 COMP–0 VALUE 1.
   77  NAME–KEY            PIC 99 COMP–0 VALUE 0.
   77  B–STATUS            PIC 99 COMP–0.
   01  BUF–LEN             PIC 99 COMP–0 VALUE 80.
   01  DATA–BUFFER         PIC X(80).
   01  DSP–STATUS          PIC 99999.
   01  KEY–BUFFER          PIC X(30).
   01  POSITION–BLOCK      PIC X(128).
   PROCEDURE DIVISION.

   BEGIN.
         CALL 'BTRV' USING B–CLOSE, B–STATUS, POSITION–BLOCK,
              DATA–BUFFER, BUF–LEN, KEY–BUFFER, NAME–KEY.
         IF B–STATUS NOT = 0
              MOVE B–STATUS TO DSP–STATUS
              DISPLAY "Btrieve Status = " DSP–STATUS.
              STOP RUN.
```

COBOL CREATE

In the following example, an application creates a Btrieve file with two keys. Key 0 is an integer key, 2 bytes long. Key 1 allows duplicates, is modifiable, and consists of two segments. The first segment is a 2-byte string, to be sorted in descending order, and the second is a zero terminated 30-byte string. The records have a fixed length of 80 bytes.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  DUPLICATES                  PIC 9  COMP-0 VALUE 1.
77  MODIFIABLE                  PIC 9  COMP-0 VALUE 2.
77  SEGMENTED                   PIC 99 COMP-0 VALUE 16.
77  DESC                        PIC 99 COMP-0 VALUE 64.
77  EXTTYPE                     PIC 999 COMP-0 VALUE 256.
77  B-INTEGER                   PIC 9  COMP-0 VALUE 1.
77  B-STRING                    PIC 9  COMP-0 VALUE 0.
77  B-ZSTRING                   PIC 99 COMP-0 VALUE 11.
77  B-CREATE                    PIC 99 COMP-0 VALUE 14.
77  KEY-NUMBER                  PIC 99 COMP-0 VALUE 0.
77  B-STATUS                    PIC 99 COMP-0 VALUE 0.
01  BUF-LEN                     PIC 99 COMP-0 VALUE 64.
01  DSP-STATUS                  PIC 99999.

01  DATA-BUFFER.
    02   RECORD-LENGTH          PIC 99 COMP-0 VALUE 80.
    02   PAGE-SIZE              PIC 9(4) COMP-0 VALUE 1024.
    02   NUMBER-OF-INDEXES      PIC 99 COMP-0 VALUE 2.
    02   NOT-USED-1             PIC X(4).
    02   VAR-REC-LEN            PIC 99 COMP-0 VALUE 0.
    02   NOT-USED-1A            PIC X(2).
    02   PRE-ALLOC              PIC 99 COMP-0 VALUE 0.
    02   KEY SPECS OCCURS 3 TIMES.
         05  KEY-POSITION       PIC 99 COMP-0.
         05  KEY-LENGTH         PIC 99 COMP-0.
         05  KEY-FLAG           PIC 99 COMP-0.
         05  NOT-USED-2         PIC X(4).
         05  KEY-TYPE           PIC 9 COMP-0 VALUE 0.
         05  FILLER             PIC X(5).
01  FILE-NAME                   PIC X(17) VALUE "\data\create.tst ".
01  POSITION-BLOCK              PIC X(128) VALUE SPACES.
```

```
PROCEDURE DIVISION.
BEGIN.
    MOVE 64 TO BUF–LEN.
    MOVE 1 TO KEY–POSITION(1).
    MOVE 2 TO KEY–LENGTH(1).
    MOVE EXTTYPE TO KEY–FLAG(1).
    MOVE B–INTEGER TO KEY–TYPE(1).
    MOVE 3 TO KEY–POSITION(2).
    MOVE 2 TO KEY–LENGTH(2).
    ADD DUPLICATES MODIFIABLE SEGMENTED EXTTYPE DESC GIVING
         KEY–FLAG(2).
    MOVE B–STRING TO KEY–TYPE(2).
    MOVE 5 TO KEY–POSITION(3).
    MOVE 30 TO KEY–LENGTH(3).
    ADD DUPLICATES MODIFIABLE EXTTYPE GIVING KEY–FLAG(3).
    MOVE B–ZSTRING TO KEY–TYPE(3).
    CALL 'BTRV' USING B–CREATE, B–STATUS, POSITION–BLOCK,
         DATA–BUFFER, BUF–LEN, FILE–NAME, KEY–NUMBER.
    IF B–STATUS NOT = 0
    MOVE B–STATUS TO DSP–STATUS
    DISPLAY "Error creating file. Status = " DSP–STATUS.
```

COBOL CREATE SUPPLEMENTAL INDEX

In the following example, an application adds an index to a Btrieve file. The first segment of the key is a 10-byte string. It allows duplicates and is modifiable. The second segment is a 2-byte integer.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  DUPLICATES          PIC 9  COMP-0 VALUE 1.
77  MODIFIABLE          PIC 9  COMP-0 VALUE 2.
77  SEGMENTED           PIC 99 COMP-0 VALUE 16.
77  EXTTYPE             PIC 999 COMP-0 VALUE 256.
77  B-INTEGER           PIC 9  COMP-0 VALUE 1.
77  B-STRING            PIC 9  COMP-0 VALUE 0.
77  B-OPEN              PIC 99 COMP-0 VALUE 0.
77  B-CLOSE             PIC 99 COMP-0 VALUE 1.
77  B-CREIDX            PIC 99 COMP-0 VALUE 31.
77  KEY-NUMBER          PIC 99 COMP-0 VALUE 0.
77  B-STATUS            PIC 99 COMP-0 VALUE 0.
01  BUF-LEN             PIC 99 COMP-0 VALUE 64.
01  DSP-STATUS          PIC 99999.
01  DATA-BUFFER.
    02  KEY-SPECS OCCURS 2 TIMES.
        05              KEY-POSITION     PIC 99 COMP-0.
        05              KEY-LENGTH       PIC 99 COMP-0.
        05              KEY-FLAG         PIC 99 COMP-0.
        05              NOT-USED-2       PIC X(4).
        05              KEY-TYPE         PIC 9 COMP-0 VALUE 0.
        05              FILLER  PIC X(5).
01  FILE-NAME           PIC X(17) VALUE"\data\create.tst ".
01  POSITION-BLOCK      PIC X(128) VALUE SPACES.

PROCEDURE DIVISION.
BEGIN.
    MOVE 32 TO BUF-LEN.
    CALL 'BTRV' USING B-OPEN, B-STATUS, POSITION-BLOCK,
        DATA-BUFFER, BUF-LEN, FILE-NAME, KEY-NUMBER.
    IF B-STATUS NOT = 0
        MOVE B-STATUS TO DSP-STATUS
        DISPLAY "Error opening file. Status = "DSP-STATUS
        STOP RUN.
    MOVE 32 TO BUF-LEN.
    MOVE 40 TO KEY-POSITION(1).
    MOVE 10 TO KEY-LENGTH(1).
    ADD DUPLICATES MODIFIABLE SEGMENTED EXTTYPE GIVING
        KEY-FLAG(1).
    MOVE B-STRING TO KEY-TYPE(1).
```

```
        MOVE 50 TO KEY–POSITION(2).
        MOVE 2 TO KEY–LENGTH(2).
        ADD DUPLICATES MODIFIABLE EXTTYPE GIVING KEY–FLAG(2).
        MOVE B–INTEGER TO KEY–TYPE(2).
        CALL ‘BTRV’ USING B–CREIDX, B–STATUS, POSITION–BLOCK,
             DATA–BUFFER, BUF–LEN, FILE–NAME,  KEY–NUMBER.
        IF B–STATUS NOT = 0
             MOVE B–STATUS TO DSP–STATUS
             DISPLAY "Error creating supplemental index. Status = " DSP–STATUS
             STOP RUN.
```

COBOL DELETE

In the following example, a COBOL application for an airline company uses the Delete operation to reflect the fact that a passenger has cancelled a flight reservation.

```
   DATA DIVISION.
   WORKING–STORAGE SECTION.
   77  B–DELETE                   PIC 99 COMP–0 VALUE 4.
   77  B–GET–EQUAL                PIC 99 COMP–0 VALUE 5.
   77  NAME–KEY                   PIC 99 COMP–0 VALUE 0.
   77  B–STATUS                   PIC 99 COMP–0.
   01  DISPLAY–STATUS             PIC 99999.
   01  BUF–LEN                    PIC 99 COMP–0 VALUE 30.
   01  RESERVATION–RECORD.
       02   FLIGHT–NUMBER         PIC X(3).
       02   PASSENGER             PIC X(15).
       02   AMOUNT–PAID           PIC X(6).
       02   ISSUE–DATE            PIC X(6).
   01  KEY–BUFFER                 PIC X(15).
   01  POSITION–BLOCK             PIC X(128).

   PROCEDURE DIVISION.
   BEGIN.
       MOVE "Martin, Dave H." TO KEY–BUFFER.
       CALL ‘BTRV’ USING B–GET–EQUAL, B–STATUS, POSITION–BLOCK,
            RESERVATION–RECORD, BUF–LEN, KEY–BUFFER, NAME–KEY.
       IF B–STATUS NOT = 0
            MOVE B–STATUS TO DISPLAY–STATUS
            DISPLAY "Btrieve Status = " DISPLAY–STATUS
            STOP RUN.
       CALL ‘BTRV’ USING B–DELETE, B–STATUS, POSITION–BLOCK,
            RESERVATION–RECORD, BUF–LEN, KEY–BUFFER, NAME–KEY.
       IF B–STATUS NOT = 0
            MOVE B–STATUS TO DISPLAY–STATUS
            DISPLAY "Btrieve Status = " DISPLAY–STATUS.
            STOP RUN.
```

After the Delete operation, Btrieve's current position in the file is as follows:

326	Crawley, Joe J.	179.85	061582	
711	Howell, Susan	259.40	052382	← Previous record
326	Peters, John H.	445.80	061782	← Next record
840	White, Rosemary	397.00	060282	

Access Path →

COBOL DROP SUPPLEMENTAL INDEX

In the following example, an application drops a supplemental index in a Btrieve file because the file no longer needs to be accessed by that index. The index number is 3.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  B-DROP               PIC 99 COMP-0 VALUE 32.
77  B-STATUS             PIC 99 COMP-0 VALUE 0.
77  INV-KNUM             PIC 99 COMP-0 VALUE 6.
01  BUF-LEN              PIC 99 COMP-0 VALUE 64.
01  POSITION-BLOCK       PIC X(128) VALUE SPACES.
01  DSP-STATUS           PIC 99999.
01  KEY-NUMBER           PIC 99 COMP-0 VALUE 3.
01  DUMY-BUF             PIC X(2) VALUE SPACES.
PROCEDURE DIVISION.
BEGIN.
    MOVE 1 TO BUF-LEN.
    CALL 'BTRV' USING B-DROP, B-STATUS, POSITION-BLOCK,
         DUMY-BUF, BUF-LEN, DUMY-BUF, KEY-NUMBER.
    IF B-STATUS = INV-KNUM
         DISPLAY "Key Number to drop is not a supplemental index"
    ELSE
    IF B-STATUS NOT = 0
         MOVE B-STATUS TO DSP-STATUS
         DISPLAY "Error creating supplemental index.Status = " DSP-STATUS
         STOP RUN.
    IF B-STATUS NOT = 0
         IF B-STATUS = INV-KNUM
           DISPLAY "Key Number to drop is not a supplemental index"
         ELSE
         MOVE B-STATUS TO DSP-STATUS
         DISPLAY "Error creating supplemental index.Status = " DSP-STATUS
         STOP RUN
```

COBOL END TRANSACTION

The following application uses transaction control to ensure that one account in a ledger file is not debited unless another is also credited.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  B-ABORT             PIC 99 COMP-0 VALUE 21.
77  B-BEGIN             PIC 99 COMP-0 VALUE 19.
77  B-END               PIC 99 COMP-0 VALUE 20.
77  B-GET-EQ            PIC 99 COMP-0 VALUE 5.
77  B-OPEN              PIC 99 COMP-0 VALUE 0.
77  B-UPDATE            PIC 99 COMP-0 VALUE 3.
77  KEY-NUM             PIC 99 COMP-0 VALUE 0.
01  B-STATUS            PIC 99 COMP-0.
01  DSP-STATUS          PIC 99999.
01  BUF-LEN             PIC 99 COMP-0 VALUE 47.
01  LEDGER-REC.
    02  ACCT-ID         PIC 9(5) COMP-0.
    02  DESC            PIC X(40).
    02  BALANCE         PIC 9(8) COMP-3.
01  KEY-BUF             PIC 9(5) COMP-0.
01  POS-BLK             PIC X(128).
01  DUMY-BUF            PIC X(128).
01  DUMY-LEN            PIC 99 COMP-0 VALUE 128.

PROCEDURE DIVISION.
BEGIN.
*
* Start the transaction.
*
    CALL 'BTRV' USING B-BEGIN, B-STATUS, DUMY-BUF, DUMY-BUF,
                      DUMY-LEN, DUMY-BUF, KEY-NUM.
    IF B-STATUS NOT = 0
        MOVE B-STATUS TO DSP-STATUS
        DISPLAY "Error beginning tran. Status = "DSP-STATUS
        STOP RUN.

*
*  Retrieve and update the cash account record.
*
    MOVE 101 TO KEY-BUF.
    CALL 'BTRV' USING B-GET-EQ, B-STATUS, POS-BLK, LEDGER-REC,
                      BUF-LEN, KEY-BUF, KEY-NUM.
        IF B–STATUS NOT = 0
              MOVE B–STATUS TO DSP–STATUS
              DISPLAY "Error retrieving cash record.
                 Status =" DSP–STATUS
              CALL 'BTRV' USING B–ABORT, B–STATUS, DUMY–BUF,
                                DUMY–BUF, DUMY–LEN, DUMY–BUF, KEY–NUM
              STOP RUN.
        SUBTRACT 250 FROM BALANCE.
        CALL 'BTRV' USING B–UPDATE, B–STATUS, POS–BLK, LEDGER–REC,
                                BUF–LEN, KEY–BUF, KEY–NUM.
        IF B–STATUS NOT = 0
              MOVE B–STATUS TO DSP–STATUS
              DISPLAY "Error updating cash record. Status =" DSP–STATUS
              CALL 'BTRV' USING B–ABORT, B–STATUS, DUMY–BUF,
                                DUMY–BUF, DUMY–LEN, DUMY–BUF, KEY–NUM.
              STOP RUN.
   *
   *  Retrieve and update the office expense account record.
   *
        MOVE 511 TO KEY–BUF.
        CALL 'BTRV' USING B–GET–EQ, B–STATUS, POS–BLK, LEDGER–REC,
                                BUF–LEN, KEY–BUF, KEY–NUM.
        IF B–STATUS NOT = 0
              MOVE B–STATUS TO DSP–STATUS
              DISPLAY "Error retrieving expense record.Status =" DSP–STATUS
              CALL 'BTRV' USING B–ABORT, B–STATUS, DUMY–BUF,
                                DUMY–BUF, DUMY–LEN, DUMY–BUF, KEY–NUM.
              STOP RUN.
        ADD 250 TO BALANCE.
        CALL 'BTRV' USING B–UPDATE, B–STATUS, POS–BLK, LEDGER–REC,
                                BUF–LEN, KEY–BUF, KEY–NUM.
        IF B–STATUS NOT = 0
              MOVE B–STATUS TO DSP–STATUS
              DISPLAY "Error updating expense record.Status =" DSP–STATUS
              CALL 'BTRV' USING B–ABORT, B–STATUS, DUMY–BUF, DUMY–BUF,
                                DUMY–LEN, DUMY–BUF; KEY–NUM
              STOP RUN.
   *
   * End the transaction.
   *    CALL 'BTRV' USING B–END, B–STATUS, DUMY–BUF, DUMY–BUF,
              DUMY–LEN,DUMY–BUF, KEY–NUM.
        IF B–STATUS NOT = 0
              MOVE B–STATUS TO DSP–STATUS
              DISPLAY "Error ending tran. Status = "DSP–STATUS
        STOP RUN.
```

COBOL EXTEND

The following example illustrates how an application might use the Extend operation to expand a Btrieve file in order to gain more disk space.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  B-CLOSE             PIC 99 COMP-0 VALUE 1.
77  B-EXTEND            PIC 99 COMP-0 VALUE 16.
77  B-OPEN              PIC 99 COMP-0 VALUE 0.
77  NAME-KEY            PIC 99 COMP-0 VALUE 0.
01  B-STATUS            PIC 99 COMP-0.
01  DSP-STATUS          PIC 99999.
01  BUF-LEN             PIC 99 COMP-0 VALUE 97.
01  DATA-BUFFER.
    02   NAME           PIC X(30).
    02   STREET         PIC X(30).
    02   CITY           PIC X(30).
    02   STATE          PIC XX.
    02   ZIP            PIC 9(5).
01  FILE-NAME           PIC X(14) VALUE "ADDRESS.BTR  ".
01  EXT-NAME            PIC X(15) VALUE "B:\ADDRESS.EXT ".
01  KEY-BUFFER          PIC X(30).
01  POSITION-BLOCK      PIC X(128) VALUE SPACES.

PROCEDURE DIVISION.
BEGIN.
    CALL 'BTRV' USING B-OPEN, B-STATUS, POSITION-BLOCK,
                DATA-BUFFER, BUF-LEN, FILE-NAME, NAME-KEY.
    IF B-STATUS NOT = 0
         MOVE B-STATUS TO DSP-STATUS
         DISPLAY "Error opening file. Status = " DSP-STATUS
         STOP RUN.
         CALL 'BTRV' USING B-EXTEND, B-STATUS, POSITION-BLOCK,
                     DATA-BUFFER, BUF-LEN, EXT-NAME, NAME-KEY.
    IF B-STATUS NOT = 0
         MOVE B-STATUS TO DSP-STATUS
         DISPLAY "Error extending file. Status = "DSP-STATUS
    STOP RUN.
    CALL 'BTRV' USING B-CLOSE, B-STATUS, POSITION-BLOCK,
                DATA-BUFFER,BUF-LEN, KEY-BUFFER, NAME-KEY.
    CALL 'BTRV' USING B-OPEN, B-STATUS, POSITION-BLOCK,
                DATA-BUFFER,BUF-LEN, FILE-NAME, NAME-KEY.
    IF B-STATUS NOT = 0
         MOVE B-STATUS TO DSP-STATUS
         DISPLAY "Error reopening file. Status = "DSP-STATUS
    ELSE DISPLAY "File extended successfully.".
```

COBOL GET DIRECT

The following example illustrates how an application can use the Get Direct operation to sort the records in a Btrieve file by an external index. (See the description of Get Position for an example of how to build the external index file.)

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  B-GET-DIRECT        PIC 99 COMP-0 VALUE 23.
77  B-GET-LOW           PIC 99 COMP-0 VALUE 12.
77  B-GET-NEXT          PIC 99 COMP-0 VALUE 6.
77  KEY-NUM             PIC 99 COMP-0 VALUE 0.
77  B-STATUS            PIC 99 COMP-0.
    88   B-EOF          VALUE 9.
01  DSP-STAT            PIC 99999.
01  ADDR-REC.
    02   NAME           PIC X(20).
    02   STREET         PIC X(20).
    02   CITY           PIC X(10).
    02   STATE          PIC X(2).
    02   ZIP            PIC X(5).
01  POS-BUF REDEFINES ADDR-REC.
02  REC-POS             PIC X(4).
01  FILE-LEN            PIC 99 COMP-0 VALUE 57.
01  INDX-REC.
    02   INX-POS        PIC X(4).
    02   INX-STATE      PIC X(2).
01  INDX-LEN            PIC 99 COMP-0 VALUE 6.
01  NAME-KEY            PIC X(20).
01  STATE-KEY           PIC X(2).
01  FILE-POS-BLK        PIC X(128).
.01 INDX-POS-BLK        PIC X(128).
```

```
PROCEDURE DIVISION.
BEGIN.
     CALL 'BTRV' USING B–GET–LOW, B–STATUS, INDX–POS–BLK,
           INDX–REC, INDX–LEN, STATE–KEY, KEY–NUM.
GET–NEXT.
     IF B–EOF GO TO XIT.
     IF B–STATUS NOT = 0
           MOVE B–STATUS TO DSP–STAT
           DISPLAY "Error reading file. Status = " DSP–STAT
           STOP RUN.
     MOVE INX–POS TO REC–POS.
     CALL 'BTRV' USING B–GET–DIRECT, B–STATUS, FILE–POS–BLK,
                             ADDR–REC,FILE–LEN, NAME–KEY, KEY–NUM.
     IF B–STATUS NOT = 0
           MOVE B–STATUS TO DSP–STAT
           DISPLAY "Error reading record. Status = " DSP–STAT
           STOP RUN.
     DISPLAY NAME, STREET, CITY, STATE, ZIP.
     CALL 'BTRV' USING B–GET–NEXT, B–STATUS, INDX–POS–BLK,
                             INDX–REC, INDX–LEN, STATE–KEY, KEY–NUM.
     GO TO GET–NEXT.
XIT.
```

COBOL GET DIRECTORY

The following example illustrates how an application can use the Get and Set Directory operations to retrieve the current directory at the beginning of the program, and to restore it before terminating.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  B-GET-DIR           PIC 99 COMP-0 VALUE 18.
77  B-SET-DIR           PIC 99 COMP-0 VALUE 17.
77  B-STATUS            PIC 99 COMP-0.
01  DSP-STATUS          PIC 99999.
01  DIR-PATH            PIC X(64).
01  DUMY-BUF            PIC X(128).
01  DUMY-LEN            PIC 99 COMP-0 VALUE 128.
01  KEY-NUM             PIC 99 COMP-0 VALUE 0.

PROCEDURE DIVISION.
BEGIN.
    CALL 'BTRV' USING B-GET-DIR, B-STATUS, DUMY-BUF, DUMY-BUF,
                        DUMY-LEN,DIR-PATH, KEY-NUM.
    IF B-STATUS NOT = 0
        MOVE B-STATUS TO DSP-STATUS
        DISPLAY "Error getting dir. Status = " DSP-STATUS
        STOP RUN.
    CALL 'BTRV' USING B-SET-DIR, B-STATUS, DUMY-BUF, DUMY-BUF,
                        DUMY-LEN,DIR-PATH, KEY-NUM.
    IF B-STATUS NOT = 0
        MOVE B-STATUS TO DSP-STATUS
        DISPLAY "Error setting dir. Status = " DSP-STATUS
        STOP RUN.
```

COBOL GET EQUAL

In the following example, part number is a key in an inventory file. The application program retrieves the record containing inventory information for a particular part number with a single Get Equal operation, in order to determine whether or not to reorder that part number.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  B-GET-EQUAL         PIC 99 COMP-0 VALUE 5.
77  PART-KEY            PIC 99 COMP-0 VALUE 0.
77  B-STATUS            PIC 99 COMP-0.
01  DSP-STAT            PIC 99999.
01  INVENTORY-RECORD.
    02    PART-NUM      PIC X(5).
    02    PART-DESC     PIC X(10).
    02    QUAN-ON-HAND          PIC X(3).
    02    REORDER-POINT         PIC X(3).
    02    REORDER-QUAN          PIC X(3).
01  DATA-LEN            PIC 99 COMP-0 VALUE 24.
01  KEY-BUFFER          PIC X(5).
01  POSITION-BLOCK      PIC X(128).
PROCEDURE DIVISION.
BEGIN.
    MOVE "03426" TO KEY-BUFFER.
    CALL 'BTRV' USING B-GET-EQUAL, B-STATUS, POSITION-BLOCK,
          INVENTORY-RECORD, DATA-LEN, KEY-BUFFER, PART-KEY.
    IF B-STATUS NOT = 0
          MOVE B-STATUS TO DSP-STAT
          DISPLAY "Error reading file. Status = " DSP-STAT
          STOP RUN.
    IF QUAN-ON-HAND < REORDER-POINT
          DISPLAY "Time to order " REORDER-QUAN " units of " PART-DESC.
```

The table below shows Btrieve's current position in the file after the Get Equal operation.

03419	Pliers	003	010	015	← Previous record
03426	Hammer	010	003	005	← Current record
03430	Saw	005	002	003	← Next record
03560	Wrench	008	005	005	

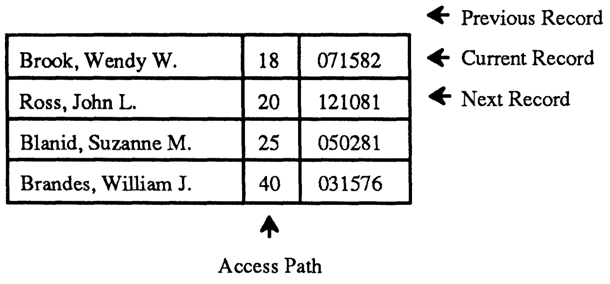
Access Path ↑

COBOL GET FIRST

The following example illustrates how an application might use the Get First operation to find the youngest employee in the company. Age is key number 2 in the employee file.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  B-GET-FIRST          PIC 99 COMP-0 VALUE 12.
77  AGE-KEY              PIC 99 COMP-0 VALUE 2.
77  B-STATUS             PIC 99 COMP-0.
01  DSP-STAT             PIC 99999.
01  EMP-REC.
    02   NAME            PIC X(20).
    02   AGE             PIC X(2).
    02   HIRE-DATE       PIC X(6).
01  DATA-LEN             PIC 99 COMP-0 VALUE 28.
01  KEY-BUFFER           PIC X(2).
01  POSITION-BLOCK       PIC X(128).

PROCEDURE DIVISION.
BEGIN.
    CALL 'BTRV' USING B-GET-FIRST, B-STATUS, POSITION-BLOCK,
                         EMP-REC,DATA-LEN, KEY-BUFFER, AGE-KEY.
    IF B-STATUS NOT = 0
         MOVE B-STATUS TO DSP-STAT
         DISPLAY "Error reading file. Status = " DSP-STAT
         STOP RUN.
    DISPLAY "Youngest employee is " NAME.
```

After the Get First operation, Btrieve's current position in the file is as follows:



Brook, Wendy W.	18	071582
Ross, John L.	20	121081
Blanid, Suzanne M.	25	050281
Brandes, William J.	40	031576

COBOL GET GREATER

The following example indicates how a COBOL application for an insurance company might use the Get Greater operation to determine which policy holders have more than three traffic violations. The number of traffic violations is key 2 in the policy file.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  B-GET-NEXT         PIC 99 COMP-0 VALUE 6.
77  B-GET-GT           PIC 99 COMP-0 VALUE 8.
77  VIOLATION-KEY      PIC 99 COMP-0 VALUE 2.
77  B-STATUS           PIC 99 COMP-0.
    88   B-EOF         VALUE 9.
01  DSP-STAT           PIC 99999.
01  POLICY-RECORD.
    02   POLICY-NUMBER          PIC X(10).
    02   NAME          PIC X(20).
    02   EFFECT-DATE   PIC X(6).
    02   VIOLATIONS    PIC X(2).
01  DATA-LEN           PIC 99 COMP-0 VALUE 38.
01  KEY-BUFFER         PIC X(2).
01  POSITION-BLOCK     PIC X(128).

PROCEDURE DIVISION.
BEGIN.
    MOVE "03" TO KEY-BUFFER.
    CALL 'BTRV' USING B-GET-GT, B-STATUS, POSITION-BLOCK,
         POLICY-RECORD,DATA-LEN, KEY-BUFFER, VIOLATION-KEY.
    IF B-STATUS NOT = 0
         MOVE B-STATUS TO DSP-STAT
         DISPLAY "Error reading file. Status = " DSP-STAT
         STOP RUN.
GET-NEXT.
    IF B-EOF GO TO XIT.
    DISPLAY NAME " has " VIOLATIONS " traffic violations".
    CALL 'BTRV' USING B-GET-NEXT, B-STATUS, POSITION-BLOCK,
         POLICY-RECORD, DATA-LEN, KEY-BUFFER, VIOLATION-KEY.

    IF B-STATUS NOT = 0 AND NOT B-EOF
         MOVE B-STATUS TO DSP-STAT
         DISPLAY "Error reading file. Status = " DSP-STAT
         STOP RUN.
    GO TO GET-NEXT.
```

COBOL GET GREATER OR EQUAL

If the date of a sale is a key in an invoice file, an application might use the Get Greater Or Equal operation to retrieve the invoice record for the first sale made in May, 1982.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  B-GET-GE             PIC 99 COMP-0 VALUE 9.
77  DATE-KEY             PIC 99 COMP-0 VALUE 1.
77  B-STATUS             PIC 99 COMP-0.
01  DSP-STAT             PIC 99999.
01  INVOICE-REC.
    02   INV-NUMBER      PIC X(5).
    02   DATE-SALE       PIC X(6).
    02   CUST-NUM        PIC X(5).
    02   TOTAL-PRICE     PIC X(8).
01  DATA-LEN             PIC 99 COMP-0 VALUE 24.
01  KEY-BUFFER           PIC X(6).
01  POSITION-BLOCK       PIC X(128).

PROCEDURE DIVISION.
BEGIN.
    MOVE "050182" TO KEY-BUFFER.
    CALL 'BTRV' USING B-GET-GE, B-STATUS, POSITION-BLOCK,
         INVOICE-REC, DATA-LEN, KEY-BUFFER, DATE-KEY.
    IF B-STATUS NOT = 0
         MOVE B-STATUS TO DSP-STAT
         DISPLAY "Error reading file. Status = " DSP-STAT
         STOP RUN.
    DISPLAY "First sale in May was to " CUST-NUM " for " TOTAL-PRICE.
```

After the Get Greater Or Equal Operation, Btrieve's current position in the file is as follows:

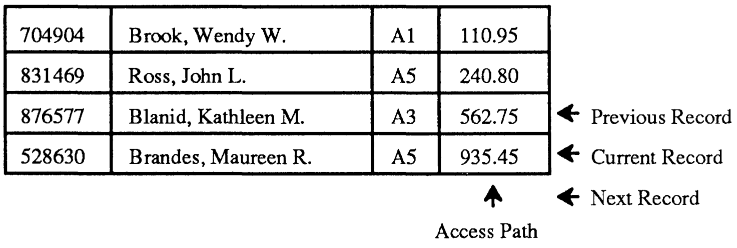
03110	041582	11315	00184.00	
03111	042882	34800	00096.00	
03112	042882	51428	00124.56	← Previous record
03113	050282	62541	00036.45	← Current record
03114	050282	14367	00098.72	← Next record
03115	051682	15699	00575.99	
	↑ Access Path			

COBOL GET LAST

The following example illustrates how an application might use the Get Last operation to determine which employee had the highest commissions last month.

```
DATA DIVISION.
WORKING–STORAGE SECTION.
77  B–GET–HIGH          PIC 99 COMP–0 VALUE 13.
77  COMM–KEY            PIC 99 COMP–0 VALUE 1.
77  B–STATUS            PIC 99 COMP–0.
01  DSP–STAT            PIC 99999.
01  EMP–REC.
    02   EMP–NUM        PIC X(6).
    02   EMP–NAME       PIC X(20).
    02   EMP–DEPT       PIC X(2).
    02   EMP–TOT–COM    PIC X(6).
    02   EMP–CUR–COM    PIC X(6).
01  BUF–LEN             PIC 99 COMP–0 VALUE 40.
01  KEY–BUFFER          PIC X(6).
01  POSITION–BLOCK      PIC X(128).

PROCEDURE DIVISION.
BEGIN.
    CALL 'BTRV' USING B–GET–LAST, B–STATUS, POSITION–BLOCK,
                        EMP–REC,BUF–LEN, KEY–BUFFER, COMM–KEY.
    IF B–STATUS NOT = 0
         MOVE B–STATUS TO DSP–STAT
         DISPLAY "Error reading file. Status = " DSP–STAT
         STOP RUN.
    DISPLAY "Employee with highest commissions last month was " EMP–NAME.
```

After the Get Last operation, Btrieve's current positioning in the file is as follows:



704904	Brook, Wendy W.	A1	110.95
831469	Ross, John L.	A5	240.80
876577	Blanid, Kathleen M.	A3	562.75
528630	Brandes, Maureen R.	A5	935.45

COBOL GET LESS THAN

The following COBOL example indicates how an application may use Get Less Than, followed by Get Previous, to find the names of all customers whose magazine subscriptions have less than three issues left before they run out. The number of issues remaining is key 2 in the subscription file.

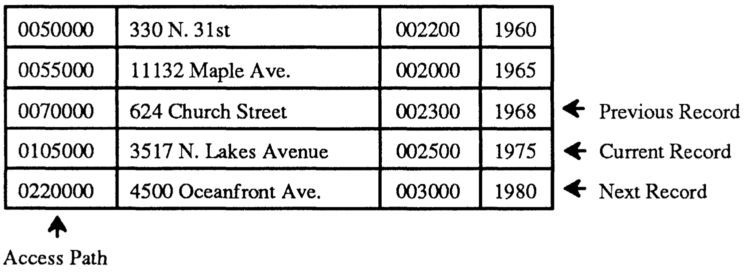
```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  B-GET-LT              PIC 99 COMP-0 VALUE 10.
77  B-GET-PREV            PIC 99 COMP-0 VALUE 7.
77   ISS-REM-KEY          PIC 99 COMP-0 VALUE 2.
77  B-STATUS              PIC 99 COMP-0.
    88    B-EOF           VALUE 9.
01  DSP-STAT              PIC 99999.
01  SUBSCRIPTION-REC.
    02    CUST-NAME       PIC X(20).
    02    DATE-SUB        PIC X(6).
    02    DATE-PAID       PIC X(6).
    02    ISSUES-PURCH PIC X(3).
    02    ISSUES-REM      PIC X(3).
01  BUF-LEN               PIC 99 COMP-0 VALUE 38.
01  KEY-BUFFER            PIC X(3).
01  POSITION-BLOCK        PIC X(128).

PROCEDURE DIVISION.
BEGIN.
    MOVE "003" TO KEY-BUFFER.
    CALL 'BTRV' USING B-GET-LT, B-STATUS, POSITION-BLOCK,
          SUBSCRIPTION-REC,  BUF-LEN, KEY-BUFFER, ISS-REM-KEY.
    IF B-STATUS NOT = 0 AND NOT B-EOF
          MOVE B-STATUS TO DSP-STAT
          DISPLAY "Error reading file. Status = " DSP-STAT
          STOP RUN.

GET-NEXT.
    IF B-EOF GO TO XIT.
    DISPLAY "Send reorder form to " CUST-NAME.
    CALL 'BTRV' USING B-GET-PREV, B-STATUS, POSITION-BLOCK,
          SUBSCRIPTION-REC, BUF-LEN, KEY-BUFFER, ISS-REM-KEY.
    IF B-STATUS NOT = 0 AND NOT B-EOF
          MOVE B-STATUS TO DSP-STAT
          DISPLAY "Error reading file. Status = " DSP-STAT
          STOP RUN.
    GO TO GET-NEXT.
XIT.
```

COBOL GET LESS THAN OR EQUAL

In the following example, an application uses the Get Less Than Or Equal operation to retrieve the first house that falls within a prospective customer's price limit of $110,000.

```
DATA DIVISION.
WORKING–STORAGE SECTION.
77  B–GET–LE           PIC 99 COMP–0 VALUE 11.
77  PRICE–KEY          PIC 99 COMP–0 VALUE 0.
77  B–STATUS           PIC 99 COMP–0.
01  DSP–STAT           PIC 99999.
01  HOME–REC.
    02   PRICE         PIC X(7).
    02   ADDRESS       PIC X(20).
    02   SQUARE–FEET   PIC X(6).
    02   YEAR–BUILT    PIC X(4).
01  DATA–LEN           PIC 99 COMP–0 VALUE 37.
01  KEY–BUFFER         PIC X(7).
01  POSITION–BLOCK     PIC X(128).

PROCEDURE DIVISION.
BEGIN.
    MOVE "0110000" TO KEY–BUFFER.
    CALL 'BTRV' USING B–GET–LE, B–STATUS, POSITION–BLOCK,
         HOME–REC, DATA–LEN, KEY–BUFFER, PRICE–KEY.
    IF B–STATUS NOT = 0
         MOVE B–STATUS TO DSP–STAT
         DISPLAY "Error reading file. Status = " DSP–STAT
         STOP RUN.
    DISPLAY "The home at " ADDRESS " sells for " PRICE.
```

After the Get Less Than Or Equal operation, Btrieve's current position is as follows:



0050000	330 N. 31st	002200	1960	
0055000	11132 Maple Ave.	002000	1965	
0070000	624 Church Street	002300	1968	← Previous Record
0105000	3517 N. Lakes Avenue	002500	1975	← Current Record
0220000	4500 Oceanfront Ave.	003000	1980	← Next Record

COBOL GET NEXT

In the following example, an application uses the Get Next operation to generate a set of mailing labels sorted according to zip code. The zip code (ZIP) is key 1 in the file.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  B-GET-NEXT          PIC 99 COMP-0 VALUE 6.
77  B-GET-FIRST         PIC 99 COMP-0 VALUE 12.
77  ZIP-KEY             PIC 99 COMP-0 VALUE 1.
77  B-STATUS            PIC 99 COMP-0.
    88   B-EOF          VALUE 9.
01  DSP-STAT            PIC 99999.
01  ADDRESS-REC.
    02   NAME           PIC X(20).
    02   STREET         PIC X(20).
    02   CITY           PIC X(10).
    02   STATE          PIC X(2).
    02   ZIP            PIC X(5).
01  DATA-LEN            PIC 99 COMP-0 VALUE 57.
01  KEY-BUFFER          PIC X(5).
01  POSITION-BLOCK      PIC X(128).

PROCEDURE DIVISION.
BEGIN.
    CALL 'BTRV' USING B-GET-FIRST, B-STATUS, POSITION-BLOCK,
         ADDRESS-REC, DATA-LEN, KEY-BUFFER, ZIP-KEY.
    IF B-STATUS NOT = 0
         MOVE B-STATUS TO DSP-STAT
         DISPLAY "Error reading file. Status = " DSP-STAT
         STOP RUN.

GET-NEXT.
    IF B-EOF GO TO XIT.
    DISPLAY NAME.
    DISPLAY STREET.
    DISPLAY CITY ", " STATE " " ZIP.
    CALL 'BTRV' USING B-GET-NEXT, B-STATUS, POSITION-BLOCK,
         ADDRESS-REC,DATA-LEN, KEY-BUFFER, ZIP-KEY.
    IF B-STATUS NOT = 0 AND NOT B-EOF
         MOVE B-STATUS TO DSP-STAT
         DISPLAY "Error reading file. Status = " DSP-STAT
         STOP RUN.
    GO TO GET-NEXT.
XIT.
```

COBOL GET POSITION

The following example illustrates how Get Position can be used to construct an external index for an existing Btrieve file. Once an external index is created, the application can read the external index file from lowest to highest and use Get Direct to sort the records in a Btrieve file by some field that was not originally defined as a key field.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  B-INSERT            PIC 99 COMP-0 VALUE 2.
77  B-GET-FIRST         PIC 99 COMP-0 VALUE 12.
77  B-GET-NEXT          PIC 99 COMP-0 VALUE 6.
77  B-GET-POS           PIC 99 COMP-0 VALUE 22.
77  KEY-NUM             PIC 99 COMP-0 VALUE 0.
77  B-STATUS            PIC 99 COMP-0.
    98    B-EOF         VALUE 9.
01  DSP-STAT            PIC 99999.
01  ADDR-REC.
    02    NAME          PIC X(20).
    02    STREET        PIC X(20).
    02    CITY          PIC X(10).
    02    STATE         PIC X(2).
    02    ZIP           PIC X(5).
01  POS-BUF REDEFINES ADDR-REC.
    02    REC-POS       PIC X(4).
01  FILE-LEN            PIC 99 COMP-0 VALUE 57.
01  INDX-REC.
    02    INX-POS       PIC X(4).
    02    INX-STATE     PIC X(2).
01  INDX-LEN            PIC 99 COMP-0 VALUE 6.
01  NAME-KEY            PIC X(20).
01  STATE-KEY           PIC X(2).
01  FILE-POS-BLK        PIC X(128).
01  INDX-POS-BLK        PIC X(128).

PROCEDURE DIVISION.
BEGIN.
    CALL 'BTRV' USING B-GET-FIRST, B-STATUS, FILE-POS-BLK,
          ADDR-REC, FILE-LEN, NAME-KEY, KEY-NUM.
```

```
GET–NEXT.
     IF B–EOF GO TO XIT.
     IF B–STATUS NOT = 0
          MOVE B–STATUS TO DSP–STAT
          DISPLAY "Error reading file. Status = " DSP–STAT
          STOP RUN.
     MOVE STATE TO INX–STATE.
     CALL 'BTRV' USING B–GET–POS, B–STATUS, FILE–POS–BLK,
          ADDR–REC, FILE–LEN, NAME–KEY, KEY–NUM.
     MOVE REC–POS TO INX–POS.
     CALL 'BTRV' USING B–INSERT, B–STATUS, INDX–POS–BLK, INDX–REC,
                              INDX–LEN,STATE–KEY, KEY–NUM.
     IF B–STATUS NOT = 0
          MOVE B–STATUS TO DSP–STAT
          DISPLAY "Error inserting record. Status = "DSP–STAT
          STOP RUN.
     CALL 'BTRV' USING B–GET–NEXT, B–STATUS, FILE–POS–BLK,
          ADDR–REC, FILE–LEN, NAME–KEY, KEY–NUM.
     GO TO GET–NEXT.
XIT.
```

COBOL GET PREVIOUS

The following example illustrates how an application can use Get Previous to list corporations and their total sales dollars for the year, beginning with the corporation having the highest sales and continuing in descending order of sales dollars. Total sales is key number 1 in the company file.

```
DATA DIVISION.
WORKING–STORAGE SECTION.
77   B–GET–PREV              PIC 99 COMP–0 VALUE 7.
77   B–GET–LAST              PIC 99 COMP–0 VALUE 13.
77   SALES–KEY               PIC 99 COMP–0 VALUE 1.
77   B–STATUS                PIC 99 COMP–0.
     88    B–EOF             VALUE 9.
01   DSP–STAT                PIC 99999.
01   COMPANY–REC.
     02    NAME              PIC X(30).
     02    TOTAL–SALES       PIC X(10).
01   DATA–LEN                PIC 99 COMP–0 VALUE 40.
01    KEY–BUFFER             PIC X(10).
01   POSITION–BLOCK          PIC X(128).
```

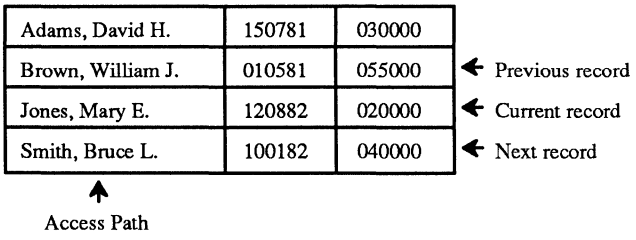
```
PROCEDURE DIVISION.
BEGIN.
     CALL 'BTRV' USING B–GET–LAST, B–STATUS, POSITION–BLOCK,
           COMPANY–REC, DATA–LEN, KEY–BUFFER, SALES–KEY.
     IF B–STATUS NOT = 0
           MOVE B–STATUS TO DSP–STAT
           DISPLAY "Error reading file. Status = " DSP–STAT
           STOP RUN.
GET–NEXT.
     IF B–EOF GO TO XIT.
     DISPLAY NAME "  " TOTAL–SALES.
     CALL 'BTRV' USING B–GET–PREV, B–STATUS, POSITION–BLOCK,
           COMPANY–REC, DATA–LEN, KEY–BUFFER, SALES–KEY.
     IF B–STATUS NOT = 0 AND NOT B–EOF
           MOVE B–STATUS TO DSP–STAT
           DISPLAY "Error reading file. Status = " DSP–STAT
           STOP RUN.
     GO TO GET–NEXT.
XIT.
```

COBOL INSERT

The following example shows how an application might use the Insert operation to add a new employee to the employee file.

```
DATA DIVISION.
WORKING–STORAGE SECTION.
77  B–INS              PIC 99 COMP–0 VALUE 2.
77  KEY–NUM            PIC 99 COMP–0 VALUE 0.
77  B–STATUS           PIC 99 COMP–0.
01  DSP–STAT           PIC 99999.
01  EMP–REC.
    02   NAME          PIC X(20).
    02   HIRE–DATE     PIC X(6).
    02   ANNUAL–SAL    PIC X(6).
01  DATA–LEN           PIC 99 COMP–0 VALUE 32.
01  KEY–BUFFER         PIC X(20).
01  POSITION–BLOCK     PIC X(128).

PROCEDURE DIVISION.
BEGIN.
    MOVE "Jones, Mary E." TO NAME.
    MOVE "120882" TO HIRE–DATE.
    MOVE "020000" TO ANNUAL–SAL.
    CALL 'BTRV' USING B–INS, B–STATUS, POSITION–BLOCK, EMP–REC,
                      DATA–LEN,KEY–BUFFER, KEY–NUM.
    IF B–STATUS NOT = 0
         MOVE B–STATUS TO DSP–STAT
         DISPLAY "Error inserting record. Status = "DSP–STAT
         STOP RUN.
```

After an Insert operation, Btrieve's current position in the file is as follows:



Adams, David H.	150781	030000
Brown, William J.	010581	055000
Jones, Mary E.	120882	020000
Smith, Bruce L.	100182	040000

COBOL OPEN

The following example illustrates the code required to open a Btrieve file from COBOL.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  B-OPEN              PIC 99 COMP-0 VALUE 0.
77  KEY-NUM             PIC 99 COMP-0 VALUE 0.
77  B-STATUS            PIC 99 COMP-0.
01  DSP-STAT            PIC 99999.
01  EMP-REC.
    02   EMP-NUM        PIC X(6).
    02   EMP-NAME       PIC X(20).
    02   EMP-DEPT       PIC X(2).
    02   EMP-TOT-COM PIC X(6).
    02   EMP-CUR-COM PIC X(6).
01  BUF-LEN             PIC 99 COMP-0 VALUE 40.
01  FILE-NAME           PIC X(21) VALUE"C:\DATA\EMPLOYEE.BTR ".
01  POSITION-BLOCK      PIC X(128).

PROCEDURE DIVISION.
BEGIN.
    CALL 'BTRV' USING B-OPEN, B-STATUS, POSITION-BLOCK, EMP-REC,
         BUF-LEN, FILE-NAME, KEY-NUM.
    IF B-STATUS NOT = 0
         MOVE B-STATUS TO DSP-STAT
         DISPLAY "Error on open. Status = " DSP-STAT
         STOP RUN.
```

COBOL SET DIRECTORY

In the following example, an application sets the current directory before performing an Open operation.

```
DATA DIVISION.
WORKING–STORAGE SECTION.
77  B–SET–DIR          PIC 99 COMP–0 VALUE 17.
77  B–OPEN             PIC 99 COMP–0 VALUE 0.
77  B–STATUS           PIC 99 COMP–0.
01  DSP–STATUS         PIC 99999.
01  ADDR–REC.
    02    NAME         PIC X(30).
    02    STREET       PIC X(30).
    02    CITY         PIC X(30).
    02    STATE        PIC X(2).
    02    ZIP          PIC X(5).
01  BUF–LEN            PIC 99 COMP–0 VALUE 97.
01  DIR–PATH           PIC X(5) VALUE "\DATA".
01  DIR–TERM           PIC 9 COMP–0 VALUE 0.
01  FILE–NAME          PIC X(14) VALUE "ADDRESS.BTR   ".
01  KEY–BUF            PIC X(30).
01  POS–BLK            PIC X(128).
01  KEY–NUM            PIC 99 COMP–0 VALUE 0.

PROCEDURE DIVISION.
BEGIN.
    CALL 'BTRV' USING B–SET–DIR, B–STATUS, POS–BLK, ADDR–REC,
                      BUF–LEN,DIR–PATH, KEY–NUM.
    IF B–STATUS NOT = 0
        MOVE B–STATUS TO DSP–STATUS
        DISPLAY "Error setting dir. Status = " DSP–STATUS
        STOP RUN.
    CALL 'BTRV' USING B–OPEN, B–STATUS, POS–BLK, ADDR–REC,
                      BUF–LEN, FILE–NAME, KEY–NUM.
    IF B–STATUS NOT = 0
        MOVE B–STATUS TO DSP–STATUS
        DISPLAY "Error opening file. Status = " DSP–STATUS
        STOP RUN.
```

COBOL STAT

In the following example, an application uses the Stat and Create operations to empty a Btrieve file.

```
DATA DIVISION.
WORKING–STORAGE SECTION.
77  B–CREATE                  PIC 99 COMP–0 VALUE 14.
77  B–OPEN                    PIC 99 COMP–0 VALUE 0.
77  B–CLOSE                   PIC 99 COMP–0 VALUE 1.
77  B–STAT                    PIC 99 COMP–0 VALUE 15.
77  KEY–NUMBER                PIC 99 COMP–0 VALUE 0.
77  B–STATUS                  PIC 99 COMP–0 VALUE 0.
01  DSP–STATUS                PIC 99999.
01  DATA–BUFFER.
    02   RECORD–LENGTH        PIC 99 COMP–0 VALUE 80.
    02   PAGE–SIZE            PIC 9(4) COMP–0 VALUE 1024.
    02   NUMBER–OF–INDEXES    PIC 99 COMP–0 VALUE 2.
    02   REC–TOT              PIC X(4).
    02   VAR–REC–LEN          PIC 99 COMP–0 VALUE 0.
    02   NOT–USED–1A          PIC X(2).
    02   PRE–ALLOC            PIC 99 COMP–0 VALUE 0.
    02   KEY SPECS OCCURS 3 TIMES.
         05 KEY–POSITION      PIC 99 COMP–0.
         05 KEY–LENGTH        PIC 99 COMP–0.
         05 KEY–FLAG          PIC 99 COMP–0.
         05 NOT–USED–2        PIC X(4).
         05 KEY–TYPE          PIC 9 COMP–0 VALUE 0.
         05 FILLER            PIC X(5).
01  DATA–BUF–LEN              PIC 99 COMP–0 VALUE 64.
01  FILE–NAME                 PIC X(17) VALUE"\data\create.tst ".
01  POSITION–BLOCK            PIC X(128) VALUE SPACES.
01  KEY–BUF                   PIC X(64).

PROCEDURE DIVISION.
BEGIN.
    CALL 'BTRV' USING B–OPEN, B–STATUS, POSITION–BLOCK,
         DATA–BUFFER, DATA–BUF–LEN, FILE–NAME, KEY–NUMBER.
    IF B–STATUS NOT = 0
         MOVE B–STATUS TO DSP–STATUS
         DISPLAY "Error opening file. Status = " DSP–STATUS
         STOP RUN.
    MOVE 64 TO DATA–BUF–LEN.
    CALL 'BTRV' USING B–STAT, B–STATUS, POSITION–BLOCK,
         DATA–BUFFER, DATA–BUF–LEN, KEY–BUF, KEY–NUMBER.
```

```
IF B–STATUS NOT = 0
     MOVE B–STATUS TO DSP–STATUS
     DISPLAY "Unable to retrieve stats. Status = " DSP–STATUS
     STOP RUN.
CALL 'BTRV' USING B–CLOSE, B–STATUS, POSITION–BLOCK,
     DATA–BUFFER, DATA–BUF–LEN, KEY–BUF, KEY–NUMBER.
IF B–STATUS NOT = 0
     MOVE B–STATUS TO DSP–STATUS
     DISPLAY "Unable to close file. Status = " DSP–STATUS
     STOP RUN.
MOVE 64 TO DATA–BUF–LEN.
CALL 'BTRV' USING B–CREATE, B–STATUS, POSITION–BLOCK,
     DATA–BUFFER, DATA–BUF–LEN, KEY–BUF, KEY–NUMBER.
IF B–STATUS NOT = 0
     MOVE B–STATUS TO DSP–STATUS
     DISPLAY "Unable to recreate file. Status = " DSP–STATUS
     STOP RUN.
```

COBOL STEP FIRST

See Step Next.

COBOL STEP LAST

See Step Previous.

COBOL STEP NEXT

The following example illustrates how an application might use the Step First and Step Next operations to recover a file whose indexes have been damaged by a system failure.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  B-INSERT            PIC 99 COMP-0 VALUE 2.
77  B-STEP-NEXT         PIC 99 COMP-0 VALUE 24.
77  B-STEP-FST          PIC 99 COMP-0 VALUE 33.
77  KEY-NUM             PIC 99 COMP-0 VALUE 0.
77  B-STATUS            PIC 99 COMP-0.
88  B-EOF               VALUE 9.
01  DSP-STAT            PIC 99999.
01  OLD-REC.
    02    OLD-NUM       PIC X(6).
    02    OLD-NAME      PIC X(30).
    02    OLD-ADDR      PIC X(50).
01  NEW-REC.
    02    NEW-NUM       PIC X(6).
    02    NEW-NAME      PIC X(30).
    02    NEW-ADDR      PIC X(50).
01  BUF-LEN             PIC 99 COMP-0 VALUE 86.
01  OLD-KEY             PIC X(6).
01  NEW-KEY             PIC X(6).
01  OLD-POS-BLK         PIC X(128).
01  NEW-POS-BLK         PIC X(128).

PROCEDURE DIVISION.
BEGIN.
    CALL 'BTRV' USING B-STEP-FST, B-STATUS, OLD-POS-BLK,
                        OLD-REC,BUF-LEN, OLD-KEY, KEY-NUM.
GET-NEXT.
    IF B-EOF GO TO XIT.
    IF B-STATUS NOT = 0
        MOVE B-STATUS TO DSP-STAT
        DISPLAY "Error reading file. Status = " DSP-STAT
        STOP RUN.
    MOVE OLD-NUM TO NEW-NUM.
    MOVE OLD-NAME TO NEW-NAME.
    MOVE OLD-ADDR TO NEW-ADDR.
    CALL 'BTRV' USING B-INSERT, B-STATUS, NEW-POS-BLK, NEW-REC,
                        BUF-LEN,NEW-KEY, KEY-NUM.
        IF B–STATUS NOT = 0
            MOVE B–STATUS TO DSP–STAT
            DISPLAY "Error reading record. Status = " DSP–STAT
            STOP RUN.
        CALL 'BTRV' USING B–STEP–NEXT, B–STATUS, OLD–POS–BLK,
                               OLD–REC,BUF–LEN, OLD–KEY, KEY–NUM.
        GO TO GET–NEXT.
    XIT.
```

COBOL STEP PREVIOUS

The following example illustrates how an application might use the Step Last and Step Previous operations to recover a file whose indexes have been damaged by a system failure.

```
    DATA DIVISION.
    WORKING–STORAGE SECTION.
    77  B–INSERT            PIC 99 COMP–0 VALUE 2.
    77  B–STEP–PREV         PIC 99 COMP–0 VALUE 35.
    77  B–STEP–LST          PIC 99 COMP–0 VALUE 34.
    77  KEY–NUM             PIC 99 COMP–0 VALUE 0.
    77  B–STATUS            PIC 99 COMP–0.
    88  B–EOF               VALUE 9.
    01  DSP–STAT            PIC 99999.
    01  OLD–REC.
        02  OLD–NUM         PIC X(6).
        02  OLD–NAME        PIC X(30).
        02  OLD–ADDR        PIC X(50).
    01  NEW–REC.
        02  NEW–NUM         PIC X(6).
        02  NEW–NAME        PIC X(30).
        02  NEW–ADDR        PIC X(50).
    01  BUF–LEN             PIC 99 COMP–0 VALUE 86.
    01  OLD–KEY             PIC X(6).
    01  NEW–KEY             PIC X(6).
    01  OLD–POS–BLK         PIC X(128).
    01  NEW–POS–BLK         PIC X(128).
```

```
PROCEDURE DIVISION.
BEGIN.
     CALL 'BTRV' USING B–STEP–LST, B–STATUS, OLD–POS–BLK,
                                OLD–REC,BUF–LEN, OLD–KEY, KEY–NUM.
GET–NEXT.
     IF B–EOF GO TO XIT.
     IF B–STATUS NOT = 0
           MOVE B–STATUS TO DSP–STAT
           DISPLAY "Error reading file. Status = " DSP–STAT
           STOP RUN.
     MOVE OLD–NUM TO NEW–NUM.
     MOVE OLD–NAME TO NEW–NAME.
     MOVE OLD–ADDR TO NEW–ADDR.
     CALL 'BTRV' USING B–INSERT, B–STATUS, NEW–POS–BLK, NEW–REC,
                                BUF–LEN, NEW–KEY, KEY–NUM.

     IF B–STATUS NOT = 0
           MOVE B–STATUS TO DSP–STAT
           DISPLAY "Error writing record. Status = " DSP–STAT
           STOP RUN.
     CALL 'BTRV' USING B–STEP–PREV, B–STATUS, OLD–POS–BLK,
                                OLD–REC,BUF–LEN, OLD–KEY, KEY–NUM.
     GO TO GET–NEXT.
XIT.
```

COBOL UPDATE

The following example shows how an application might use the Update operation to reflect the fact that an employee just received a raise.

```
DATA DIVISION.
WORKING–STORAGE SECTION.
77  B–GET–EQ              PIC 99 COMP–0 VALUE 5.
77  B–UPDATE              PIC 99 COMP–0 VALUE 3.
77  NAME–KEY              PIC 99 COMP–0 VALUE 0.
77  B–STATUS              PIC 99 COMP–0.
01  DSP–STAT              PIC 99999.
01  EMP–REC.
    02    NAME            PIC X(20).
    02    HIRE–DATE       PIC X(6).
    02    ANNUAL–SAL      PIC X(6).
01  DATA–LEN              PIC 99 COMP–0 VALUE 32.
01  KEY–BUFFER            PIC X(20).
01  POSITION–BLOCK        PIC X(128).

PROCEDURE DIVISION.
BEGIN.
    MOVE "Jones, Mary E." TO KEY–BUFFER.
    CALL 'BTRV' USING B–GET–EQ, B–STATUS, POSITION–BLOCK,
          EMP–REC, DATA–LEN, KEY–BUFFER, NAME–KEY.
    IF B–STATUS NOT = 0
          MOVE B–STATUS TO DSP–STAT
          DISPLAY "Error reading file. Status = " DSP–STAT
          STOP RUN.
    MOVE "025000" TO ANNUAL–SAL.
    CALL 'BTRV' USING B–UPDATE, B–STATUS, POSITION–BLOCK,
          EMP–REC, DATA–LEN, KEY–BUFFER, NAME–KEY.
    IF B–STATUS NOT = 0
          MOVE B–STATUS TO DSP–STAT
          DISPLAY "Error updating file. Status = " DSP–STAT
          STOP RUN.
```

After the Update operation, Btrieve's position in the file is as follows:

Adams, David H.	150781	030000	
Brown, William J.	010581	055000	← Previous record
Jones, Mary E.	120882	025000	← Current record
Smith, Bruce L.	100182	040000	← Next record

↑
Access Path

APPENDIX E: C EXAMPLES

The following examples illustrate how to call Btrieve from a C application. A different example is presented for each Btrieve operation.

C ABORT TRANSACTION

In the following example, an application must insert a record into an order header file and insert two records into an order detail file to record the sale of a box of diskettes and a ribbon to one customer. If an error occurs on any of the operations, the transaction is aborted so that the database remains consistent.

```
#define B_ABORT  21
#define B_BEGIN  19
#define B_END    20
#define B_INSERT  2
#define B_OPEN    0
struct  ORDER_HDR
    {
        int     HDR_ORD_NUM;
        int     HDR_CUST;
        char    HDR_DATE[6];
    };
struct ORDER_DET
    {
        int  DET_ORD_NUM;
        int  DET_PART;
        int  DET_QUAN;
    };
struct ORDER_DET DET_BUF;
int     DET_KEY;
int     DET_LEN;
char    DET_POS[128];
char    DUMY_BUF[128];
int     DUMY_LEN;
struct  ORDER_HDR HDR_BUF;
int     HDR_KEY;
int     HDR_LEN;
```

```
char    HDR_POS[128];
int     STAT;

main ()
{
/*   Start the transaction.                                                   */
STAT = BTRV (B_BEGIN, DUMY_BUF, DUMY_BUF, & DUMY_LEN, DUMY_BUF, 0);
if (STAT != 0)
    {
        printf ("Error beginning transaction. STAT = %d",STAT);
        exit (0);
    };
/*   Insert the header record.                                                */
HDR_BUF.HDR_ORD_NUM = 236;                                      /*Order #236*/
HDR_BUF.HDR_CUST = 201;                                      /*Customer #201*/
HDR_BUF.HDR_DATE[0] = '0'                                     /*Date of sale*/
HDR_BUF.HDR_DATE[1] = '6';
HDR_BUF.HDR_DATE[2] = '1';
HDR_BUF.HDR_DATE[3] = '5';
HDR_BUF.HDR_DATE[4] = '8';
HDR_BUF.HDR_DATE[5] = '3';
HDR_LEN = sizeof (HDR_BUF);
STAT = BTRV (B_INSERT, HDR_POS, &HDR_BUF, &HDR_LEN, &HDR_KEY, 0);
if (STAT != 0)
    {
        printf ("Error inserting header record. STAT = %d",STAT);
        STAT = BTRV (B_ABORT, DUMY_BUF,  DUMY_BUF, &DUMY_LEN, DUMY_BUF, 0);
        exit (0);
    };
/*   Insert two detail records.                                               */
DET_BUF.DET_ORD_NUM = 236;                                      /*Order #236*/
DET_BUF.DET_PART = 1002;                           /*Diskettes are part #1002*/
DET_BUF.DET_QUAN = 1;                                       /*Purchased 1 box*/
DET_LEN = sizeof (DET_BUF);
STAT = BTRV (B_INSERT, DET_POS, &DET_BUF, &DET_LEN, &DET_KEY, 0);
if (STAT != 0)
    {
        printf ("Error inserting detail record. STAT = %d",STAT);
        STAT = BTRV (B_ABORT, DUMY_BUF, DUMY_BUF, &DUMY_LEN, DUMY_BUF, 0);
        exit (0);
    };
```

```
DET_BUF.DET_PART = 1024;                                    /*Ribbon is part #1024*/
DET_BUF.DET_QUAN = 1;                                       /*Purchased 1 ribbon*/
STAT = BTRV (B_INSERT, DET_POS, &DET_BUF, &DET_LEN, &DET_KEY, 0);
if  (STAT != 0)
    {
        printf ("Error inserting detail record. STAT = %d",STAT);
        STAT = BTRV (B_ABORT, DUMY_BUF, DUMY_BUF, &DUMY_LEN, DUMY_BUF, 0);
        exit (0);
    };
/*  End the transaction.                                                        */
STAT = BTRV (B_END, DUMY_BUF, DUMY_BUF, &DUMY_LEN,DUMY_BUF, 0);
if  (STAT != 0)
    {
        printf ("Error ending transaction. STAT = %d",STAT);
        exit (0);
    };
```

C BEGIN TRANSACTION

See Abort Transaction and End Transaction.

C CLOSE

The following example illustrates the code required to close a Btrieve file from C.

```
#define B_CLOSE 1

main ()
{
    int      BUF_LEN;
    char     DATA_BUF[80];
    char     KEY_BUF[30];
    char     POS_BLK[128];
    int      STATUS;

    STATUS = BTRV (B_CLOSE, POS_BLK, DATA_BUF, &BUF_LEN, KEY_BUF, 0);
    if (STATUS != 0) printf ("Error closing file. Status = %d", STATUS);
}
```

C CREATE

In the following example, an application creates a Btrieve file with two keys. Key 0 is an integer key, 2 bytes long. Key 1 allows duplicates, is modifiable, and consists of two segments. The first segment is a 2-byte string, to be sorted in descending order, and the second is a zero terminated 30-byte string. The records have a fixed length of 80.

```
#define B_CREATE 14
#define DUP        1
#define MOD        2
#define BIN        4
#define SEG        16
#define DESC       64
#define EXT_TYPE  256
#define B_INT_TYPE  1
#define B_STR_TYPE  0
#define B_ZSTR_TYPE 11

struct  KEY_SPEC
   {
        int     KEY_POS;
        int     KEY_LEN;
        int     KEY_FLAG;
        char    NOT_USE1[4];
        char    KEY_TYPE;
        char    RESERVE1[5];
   };

struct  FIL_SPEC
   {
        int     REC_LEN;
        int     PAGE_SIZ;
        int     NDX_CNT;
        char    NOT_USE2[4];
        int     FILE_FLAG;
        char    RESERVE2[2];
        int     PRE_ALLOC;
        struct  KEY_SPEC KEY_BUF[3];
   };
```

```
int     BUF_LEN;
struct  FIL_SPEC FILE_BUF;
char    FIL_NAME[17] = "\\DATA\\CREATE.TST ";
char    POS_BLK[128];
int     STAT;

main ()
{
FILE_BUF.REC_LEN = 80;
FILE_BUF.PAGE_SIZ = 1024;
FILE_BUF.FILE_FLAG=0;
FILE_BUF.NDX_CNT = 2;                                   /*# of indexes in the file*/
FILE_BUF.KEY_BUF[0].KEY_POS = 1;                                /*Key 0 position*/
FILE_BUF.KEY_BUF[0].KEY_LEN = 2;                                  /*Key 0 length*/
FILE_BUF.KEY_BUF[0].KEY_FLAG = EXT_TYPE;                /*Key 0 is extended type*/
FILE_BUF.KEY_BUF[0].KEY_TYPE = B_INT_TYPE;            /* signed binary (integer) */
FILE_BUF.KEY_BUF[1].KEY_POS = 3;                    /*Key 1, segment 1 position*/
FILE_BUF.KEY_BUF[1].KEY_LEN = 2;                      /*Key 1, segment 1 length*/
FILE_BUF.KEY_BUF[1].KEY_FLAG = DUP | MOD | SEG | EXT_TYPE | DESC;
FILE_BUF.KEY_BUF[1].KEY_TYPE = B_STR_TYPE;
FILE_BUF.KEY_BUF[2].KEY_POS = 5;                    /*Key 1, segment 2 position*/
FILE_BUF.KEY_BUF[2].KEY_LEN = 30;                     /*Key 1, segment 2 length*/
FILE_BUF.KEY_BUF[2].KEY_FLAG = DUP | MOD | EXT_TYPE;
FILE_BUF.KEY_BUF[1].KEY_TYPE = B_ZSTR_TYPE;
BUF_LEN = sizeof (FILE_BUF);
STAT = BTRV (B_CREATE, POS_BLK, &FILE_BUF, &BUF_LEN, FIL_NAME, 0);
if  (STAT != 0)
    printf ("Error creating file.  Status = %d",STAT);
else
    printf ("File created successfully.");
}
```

C CREATE SUPPLEMENTAL INDEX

In the following example, an application adds an index to a Btrieve file. The first segment of the key is a 10-byte string. It allows duplicates and is modifiable. The second segment is a 2-byte integer.

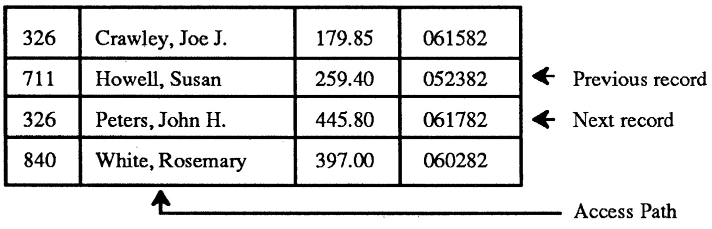
```
#define B_OPEN    0
#define B_STAT    15
#define B_CLOSE  1
#define B_INDEX  31
#define DUP   1
#define MOD   2
#define SEG   16
#define EXT_TYPE  256
#define B_STR_TYPE        0
#define B_INT_TYPE        1

struct  KEY_SPEC
    {
        int     KEY_POS;
        int     KEY_LEN;
        int     KEY_FLAG;
        char    NOT_USE1[4];
        char    KEY_TYPE;
        char    RESERVE1[5];
    };

int     KEY_BUF_LEN;
struct  KEY_SPEC KEY_BUF[2];
char    FIL_NAME[17] = "\\DATA\\CREATE.TST ";
char    POS_BLK[128];
char    DMY[80];
int     STAT;

main ()
{
KEY_BUF_LEN = sizeof (KEY_BUF);
STAT = BTRV (B_OPEN, POS_BLK, KEY_BUF, &KEY_BUF_LEN, FIL_NAME, 0);
if (STAT != 0)
    {
        printf ("Error opening file. STAT = %d", STAT);
        exit (0);
    };
KEY_BUF[0].KEY_POS = 40;                                    /* position */
KEY_BUF[0].KEY_LEN = 10;                                      /* length */
KEY_BUF[0].KEY_FLAG = DUP | MOD | EXT_TYPE | SEG;      /* extended type */
KEY_BUF[0].KEY_TYPE = B_STR_TYPE;                                   /* type string */
KEY_BUF[1].KEY_POS = 50;                                               /* position */
KEY_BUF[1].KEY_LEN = 2;                                                  /* length */
KEY_BUF[1].KEY_FLAG = DUP | MOD | EXT_TYPE;                       /* extended type */
KEY_BUF[1].KEY_TYPE = B_INT_TYPE;                                  /* type integer */
KEY_BUF_LEN = sizeof (KEY_BUF);
STAT = BTRV (B_INDEX, POS_BLK, KEY_BUF, &KEY_BUF_LEN, DMY, 0);
if (STAT != 0)
    {
        printf ("Error creating supplemental index in file. STAT = %d", STAT);
        exit (0);
    };
STAT = BTRV (B_CLOSE, POS_BLK, KEY_BUF, &KEY_BUF_LEN, DMY, 0);
if  (STAT != 0)
    {
        printf ("Error closing file. STAT = %d", STAT);
        exit (0);
    };
printf ("Supplemental Index added successfully");
}
```

C DELETE

In the following example, a C application for an airline company uses the Delete operation to reflect the fact that a passenger has cancelled a flight reservation.

```
#define B_DEL 4
#define B_GETEQ  5

struct   RESERV
    {
        char     FLIGHT[3];
        char     PASSENGR[15];
        char     AMT_PAID[6];
        char     ISS_DATE[6];
    };
int                    BUF_LEN;
struct   RESERV    RESV_BUF;
char                   KEY_BUF[] = "Martin, Dave H.";
char                   POS_BLK[128];
int                    STATUS;

main ()
{
BUF_LEN = sizeof (RESV_BUF);
STATUS = BTRV (B_GETEQ, POS_BLK, &RESV_BUF, &BUF_LEN, KEY_BUF, 0);
if (STATUS != 0)
    {
        printf ("Error reading file. Status = %d", STATUS);
        exit (0);
    }
STATUS = BTRV (B_DEL, POS_BLK, &RESV_BUF, &BUF_LEN, KEY_BUF, 0);
if (STATUS != 0)
    {
        printf ("Error deleting record. Status = %d", STATUS);
        exit (0);
    };
```

After the Delete operation, Btrieve's current position in the file is as follows:



326	Crawley, Joe J.	179.85	061582
711	Howell, Susan	259.40	052382
326	Peters, John H.	445.80	061782
840	White, Rosemary	397.00	060282

C DROP SUPPLEMENTAL INDEX

In the following example, an application drops a supplemental index in a Btrieve file because the file no longer needs to be accessed by that index. The key to be dropped is a segmented key. The first segment is a 10-byte string at position 40; the second a 2-byte integer at position 50. The code fragment checks the key specs to make sure it is deleting the proper index.

```
#define B_OPEN    0
#define B_STAT    15
#define B_DROP    32
#define SEG    16
#define SUPP   128
#define EXT_TYPE 256

struct    KEY_SPEC
    {
          int       KEY_POS;
          int       KEY_LEN;
          int       KEY_FLAG;
          long      KEY_TOT;
          char      KEY_TYPE;
          char      RESERVE1[5];
    };

struct    FIL_SPEC
    {
          int         REC_LEN;
          int         PAGE_SIZ;
          int         NDX_CNT;
          long        REC_TOT;
          int         FILE_FLAG;
          char        RESERVE2[2];
          int         UNUSED_P;
          struct    KEY_SPEC KEY_BUF[10];
    };

int       FIL_BUF_LEN;
struct    FIL_SPEC FIL_BUF;
char      KEY_BUF[64];
char      POS_BLK[128];
int       STAT;
int       KEY_NUM;
int       KEY_ENTRY;
char      DMY;
```

```
main ()
{
FIL_BUF_LEN = sizeof (FIL_BUF);
STAT = BTRV (B_STAT, POS_BLK, &FIL_BUF, &FIL_BUF_LEN, KEY_BUF, 0);
if (STAT != 0)
    {
        printf ("Error retrieving stats. STAT = %d", STAT);
        exit(0);
    };

/* find the key number that matches the specs for the index we want to delete          */
for (KEY_ENTRY = 0, KEY_NUM = 0;
    KEY_NUM = < FIL_BUF.NDX_CNT; KEY_NUM++, KEY_ENTRY++)
    {
        if ((FIL_BUF.KEY_BUF[KEY_ENTRY].KEY_FLAG & SUPP)
            && (FIL_BUF.KEY_BUF[KEY_ENTRY].KEY_POS == 40)
            && (FIL_BUF.KEY_BUF[KEY_ENTRY].KEY_LEN == 10)
            && (FIL_BUF.KEY_BUF[KEY_ENTRY+1]KEY_POS == 50)
            && (FIL_BUF.KEY_BUF[KEY_ENTRY+1]KEY_LEN == 2))
                break;
        while (FIL_BUF.KEY_FUB[KEY_ENTRY].KEY_FLAG & SEG) KEY_ENTRY++;
    }
if (KEY_NUM >= FIL_BUF.NDX_CNT)
    {
        printf ("No supplemental index has matching specifications. STAT = %d", STAT);
        exit (0);
    }
FIL_BUF_LEN = 1;
STAT = BTRV (B_DROP, POS_BLK, &DMY, &FIL_BUF_LEN, KEY_BUF, KEY_NUM);
if (STAT != 0)
    {
        printf ("Error dropping index. STAT = %d", STAT);
        exit (0);
    };
printf ("Supplemental Index dropped successfully");
};
```

C END TRANSACTION

The following application uses transaction control to ensure that one account in a ledger file is not debited unless another is also credited.

```
#define B_ABORT  21
#define B_BEGIN  19
#define B_END    20
#define B_GET_EQ 5
#define B_OPEN   0
#define B_UPDATE 3
struct  LEDGER_REC
    {
        int     ACCT_ID;
        char    DESC[40];
        long    BALANCE;
    };
int     BUF_LEN;
struct  LEDGER_REC DATA_BUF;
char    DUMY_BUF[128];
int     DUMY_LEN;
int     KEY_BUF;
char    POS_BLK[128];
int     STAT;
main ()
{
/*  Begin the transaction.                                              */
STAT = BTRV (B_BEGIN, DUMY_BUF, DUMY_BUF, &DUMY_LEN, DUMY_BUF, 0);
if  (STAT != 0)
    {
        printf ("Error beginning transaction. STAT = %d",STAT);
        exit (0);
    };
/*  Retrieve and update the cash account record.                        */
KEY_BUF = 101;                                  /*Cash is account #101*/
BUF_LEN = sizeof (DATA_BUF);
STAT = BTRV (B_GET_EQ, POS_BLK, &DATA_BUF, &BUF_LEN, &KEY_BUF, 0);
if (STAT != 0)
    {
        printf ("Error retrieving record. STAT = ", STAT);
        STAT = BTRV (B_ABORT, DUMY_BUF, DUMY_BUF, &DUMY_LEN, DUMY_BUF, 0);
        exit (0);
    };
DATA_BUF.BALANCE = DATA_BUF.BALANCE - 250;
STAT = BTRV (B_UPDATE, POS_BLK, &DATA_BUF, &BUF_LEN, &KEY_BUF, 0);
```

```
if (STAT != 0)
    {
        printf ("Error updating record. STAT = %d", STAT);
        STAT = BTRV (B_ABORT, DUMY_BUF, DUMY_BUF, &DUMY_LEN, DUMY_BUF, 0);
        exit (0);
    };

/*  Retrieve and update the office expense account record.                    */
/*  Office expense is account #511                                            */
KEY_BUF = 511;
STAT = BTRV (B_GET_EQ, POS_BLK, &DATA_BUF, &BUF_LEN, &KEY_BUF, 0);
if (STAT != 0)
    {
        printf ("Error retrieving record. STAT = %d", STAT);
        STAT = BTRV (B_ABORT, DUMY_BUF, DUMY_BUF, &DUMY_LEN, DUMY_BUF, 0);
        exit (0);
    };
DATA_BUF.BALANCE = DATA_BUF.BALANCE + 250;
STAT = BTRV(B_UPDATE, POS_BLK, &DATA_BUF, &BUF_LEN, &KEY_BUF, 0);
if  (STAT != 0)
    {
        printf ("Error updating record. STAT = ", STAT);
        STAT = BTRV( B_ABORT,  DUMY_BUF,  DUMY_BUF,  &DUMY_LEN, DUMY_BUF,  0);
        exit (0);
    };

/*  End the transaction.                                                      */
/*                                                                            */
STAT = BTRV (B_END, DUMY_BUF, DUMY_BUF, &DUMY_LEN, DUMY_BUF, 0);
if  (STAT != 0)
    {
        printf ("Error ending transaction. STAT = %d",STAT);
        exit (0);
    };
```

C EXTEND

The following example illustrates how an application might use the Extend operation to expand a Btrieve file in order to gain more disk space.

```
#define B_CLOSE   1
#define B_EXTEND 16
#define B_OPEN    0

struct ADDR_REC                                          /*Structure of address file entry*/
    {
        char    NAME[30];
        char    STREET[30];
        char    CITY[30];
        char    STATE[2];
        char    ZIP[5];
    };
int     BUF_LEN;
struct  ADDR_REC DATA_BUF;
char    EXT_NAME[15] = "B:ADDRESS.EXT ";
char    FIL_NAME[15] = "ADDRESS.BTR   ";
char    KEY_BUF[30];
char    POS_BLK[128];
int     STAT;

main ()
BUF_LEN = sizeof (DATA_BUF);
STAT = BTRV (B_OPEN, POS_BLK, &DATA_BUF, &BUF_LEN, FIL_NAME, 0);
if (STAT != 0)
    {
        printf ("Error opening file. STAT = %d", STAT);
        exit (0);
    };
STAT = BTRV (B_EXTEND, POS_BLK, &DATA_BUF, &BUF_LEN, EXT_NAME, 0);
if (STAT != 0)
    {
        printf ("Error extending file. STAT = %d", STAT);
        exit (0);
    };
STAT = BTRV (B_CLOSE, POS_BLK, &DATA_BUF, &BUF_LEN, KEY_BUF, 0);
STAT = BTRV (B_OPEN, POS_BLK, &DATA_BUF, &BUF_LEN, FIL_NAME, 0);
if (STAT != 0) printf ("Error reopening the file. STAT = %d", STAT);
else printf ("File reopened successfully.");
```

C GET DIRECT

The following example illustrates how an application can use the Get Direct operation to sort the records in a Btrieve file by an external index. (See the description of Get Position for an example of how to build the external index file.)

```
#define B_GETDRC        23
#define B_GETLW         12
#define B_GETNX         6
#define B_GETPOS        22
#define EOF_ERR         9
struct  ADDRESS
    {
        char    NAME[20];
        char    STREET[20];
        char    CITY[10];
        char    STATE[2];
        char    ZIP[5];
    };
union   ADDR_REC
    {
        struct  ADDRESS ADDR;
        long    REC_POS;
    }   FILE_BUF;
struct  INDEX
    {
        long    INX_POS;
        char    INX_ST[2];
    };
int     FILE_LEN;
char    FIL_POS[128];
struct  INDEX INDX_BUF;
int     INDX_LEN;
char    INX_POS[128];
char    NAME_KEY[20];
char    ST_KEY[2];
int     STATUS;

main ()
{
FILE_LEN = sizeof (FILE_BUF);
INDX_LEN = sizeof (INDX_BUF);
STATUS = BTRV (B_GETLW, INX_POS, &INDX_BUF, &INDX_LEN, ST_KEY, 0);
```

```
while (STATUS != EOF_ERR)
    {
        if (STATUS != 0)
            {
                printf ("Error reading file. Status = %d", STATUS);
                exit (0);
            }
        FILE_BUF.REC_POS = INDX_BUF.INX_POS;
        STATUS = BTRV (B_GETDRC, FIL_POS, &FILE_BUF, &FILE_LEN, NAME_KEY, 0);
        if (STATUS != 0)
            {
                printf ("Error on get direct. Status = %d",  STATUS);
                exit (0);
            }
        printf ("%.57s\n", FILE_BUF.ADDR.NAME);
        STATUS = BTRV (B_GETNX, INX_POS, &INDX_BUF, &INDX_LEN, ST_KEY, 0);
    }
```

C GET DIRECTORY

The following example illustrates how an application can use the Get and Set Directory operations to retrieve the current directory at the beginning of the program, and to restore it before terminating.

```
#define B_GET_DIR 18
#define B_SET_DIR 17

char        DIR_PATH[64];
char        DUMY_BUF[128];
int         DUMY_LEN;
int         STAT;

main ()
{
STAT = BTRV (B_GET_DIR, DUMY_BUF, DUMY_BUF, &DUMY_LEN, DIR_PATH, 0);
if (STAT != 0)
    {
        printf ("Error getting current dir. STAT = %d",STAT);
        exit (0);
    };
STAT = BTRV (B_SET_DIR, DUMY_BUF, DUMY_BUF, &DUMY_LEN, DIR_PATH, 0);
if (STAT != 0)
    {
        printf ("Error restoring current dir. STAT = %d",STAT);
        exit(0);
    };
```

C GET EQUAL

In the following example, part number is a key in an inventory file. The application program retrieves the record containing inventory information for a particular part number with a single Get Equal operation, in order to determine whether or not to reorder that part number.

```
#define B_GETEQ  5

struct  INVENT
    {
        char    PART_NUM[5];
        char    PART_DSC[10];
        char    QUAN_OH[3];
        char    REORD_PT[3];
        char    REORD_QU[3];
    };

int     DATA_LEN;
int     I;
struct  INVENT     INV_BUF;
char    KEY_BUF[] = "03426";
char    POS_BLK[128];
char    STR1[4];
char    STR2[4];
int     STATUS;

main ()
{
DATA_LEN = sizeof (INV_BUF);
STATUS = BTRV (B_GETEQ, POS_BLK, &INV_BUF, &DATA_LEN, KEY_BUF, 0);
if (STATUS != 0)
    {
        printf ("Error reading file. Status = %d", STATUS);
        exit (0);
    };
for (I = 0; I < 3; I++)                                  /* convert data to C strings */
    {
        STR1[I] = INV_BUF.QUAN_OH[I];
        STR2[I] = INV_BUF.REORD_PT[I];
    }
STR1[3] = STR2[3] = '\0';
if (strcmp(STR1, STR2) < 0) printf ("Time to order %.3s units of %.10s",
          INV_BUF.REORD_QU, INV_BUF.PART_DSC);
```

The table below shows Btrieve's current position in the file after the Get Equal operation.

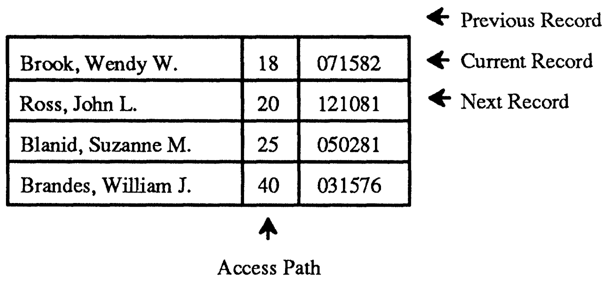
03419	Pliers	003	010	015	← Previous record
03426	Hammer	010	003	005	← Current record
03430	Saw	005	002	003	← Next record
03560	Wrench	008	005	005	
↑					
Access Path					

C GET FIRST

The following example illustrates how an application might use the Get First operation to find the youngest employee in the company. Age is key number 2 in the employee file.

```
#define B_GETFIRST  12

struct  EMP_REC
    {
        char    NAME[20];
        char    AGE[2];
        char    HIRE_DAT[6];
    };
int         DATA_LEN;
struct  EMP_REC  EMP_BUF;
char        KEY_BUF[2];
char        POS_BLK[128];
int         STATUS;

main ()
{
DATA_LEN = sizeof (EMP_BUF);
STATUS = BTRV (B_GETFIRST, POS_BLK, &EMP_BUF, &DATA_LEN, KEY_BUF, 2);
if (STATUS != 0)
    printf ("Error reading file. Status = %d", STATUS);
else
    printf ("Youngest employee is %.20s", EMP_BUF.NAME);
}
```

After the Get First operation, Btrieve's current position in the file is as follows:



Brook, Wendy W.	18	071582
Ross, John L.	20	121081
Blanid, Suzanne M.	25	050281
Brandes, William J.	40	031576

C GET GREATER

The following example indicates how a C application for an insurance company might use the Get Greater operation to determine which policyholders have more than three traffic violations. The number of traffic violations is key 2 in the policy file.

```
#define B_GETGT  8
#define B_GETNX  6
#define EOF_ERR  9

struct POLICY
    {
        char    POL_NUM[10];
        char    NAME[20];
        char    EFF_DATE[6];
        char    VIOLAT[2];
    };

int         DATA_LEN;
struct  POLICY    POL_BUF;
char        KEY_BUF[] = "03";
char        POS_BLK[128];
int         STATUS;

main ()
{
DATA_LEN = sizeof (POL_BUF);
STATUS = BTRV (B_GETGT, POS_BLK, &POL_BUF, &DATA_LEN, KEY_BUF, 2);
while (STATUS != EOF_ERR)
    {
        if (STATUS != 0)
            {
                printf ("Error reading file. Status = %d", STATUS);
                exit (0);
            }
        printf ("%.20s has %.2s traffic violations\n", POL_BUF.NAME, POL_BUF.VIOLAT);
        STATUS = BTRV (B_GETNX, POS_BLK, &POL_BUF, &DATA_LEN, KEY_BUF, 2);
    }
```

C GET GREATER OR EQUAL

If the date of a sale is a key in an invoice file, an application might use the Get Greater Or Equal operation to retrieve the invoice record for the first sale made in May, 1982.

```
#define B_OPEN    0
#define B_GETGE  9

struct   INVOICE
    {
        char     INV_NUM[5];
        char     SALE_DT[6];
        char     CUST_NUM[5];
        char     TOT_PRC[8];
    };
int            DATA_LEN;
struct INVOICE  INV_BUF;
char           KEY_BUF[] = "050182";
char           POS_BLK[128];
int            STATUS;

main ()
{
DATA_LEN = sizeof (INV_BUF);
STATUS = BTRV (B_GETGE, POS_BLK, &INV_BUF, &DATA_LEN, KEY_BUF, 1);
if (STATUS != 0)
    printf ("Error reading file. Status = %d",  STATUS);
else
    printf ("First sale in May was to %.5s for %.8s", INV_BUF.CUST_NUM, INV_BUF.TOT_PRC);
}
```

After the Get Greater Or Equal Operation, Btrieve's current position in the file is as follows:

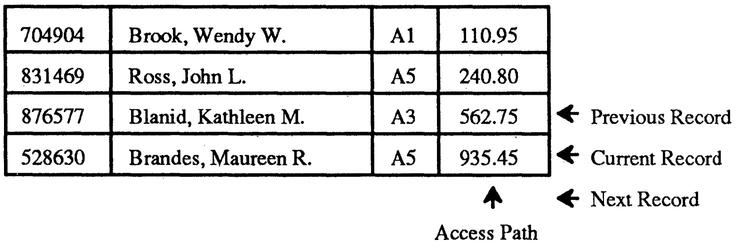
03110	041582	11315	00184.00	
03111	042882	34800	00096.00	
03112	042882	51428	00124.56	← Previous record
03113	050282	62541	00036.45	← Current record
03114	050282	14367	00098.72	← Next record
03115	051682	15699	00575.99	
	↑ Access Path			

C GET LAST

The following example illustrates how an application might use the Get Last operation to determine which employee had the highest commissions last month.

```
#define B_GETLAST          13

struct   EMP_REC
    {
         char     EMP_NUM[6];
         char     EMP_NAME[20];
         char     EMP_DEPT[2];
         char     EMP_TOT[6];
         char     EMP_CUR[6];
    };

int          BUF_LEN;
struct   EMP_REC  EMP_BUF;
char         KEY_BUF[6];
char         POS_BLK[128];
int          STATUS;

main ()
{
BUF_LEN = sizeof (EMP_BUF);
STATUS = BTRV (B_GETLAST, POS_BLK, &EMP_BUF, &BUF_LEN, KEY_BUF, 1);
if (STATUS != 0)
    printf ("Error reading file. Status = %d", STATUS);
else
    printf ("Employee with highest commissions last month was %.20s", EMP_BUF.EMP_NAME);
}
```

After the Get Last operation, Btrieve's current positioning in the file is as follows:



704904	Brook, Wendy W.	A1	110.95
831469	Ross, John L.	A5	240.80
876577	Blanid, Kathleen M.	A3	562.75
528630	Brandes, Maureen R.	A5	935.45

C GET LESS THAN

The following C example indicates how an application may use Get Less Than, followed by Get Previous, to find the names of all customers whose magazine subscriptions have less than three issues left before they run out. The number of issues remaining is key 2 in the subscription file.

```
#define B_GETLT   10
#define B_GETPR   7
#define EOF_ERR   9

struct SUBSCRP
    {
        char    CUST[20];
        char    SUB_DATE[6];
        char    PD_DATE[6];
        char    ISS_PUR[3];
        char    ISS_REM[3];
 };

int         BUF_LEN;
struct   SUBSCRP   SUB_BUF;
char        KEY_BUF[] = "003";
char        POS_BLK[128];
int         STATUS;

main ()
{
BUF_LEN = sizeof (SUB_BUF);
STATUS = BTRV (B_GETLT, POS_BLK, &SUB_BUF, &BUF_LEN, KEY_BUF, 2);
while (STATUS != EOF_ERR)
    {
        if (STATUS != 0)
            {
                printf ("Error reading file. Status = %d", STATUS);
                exit (0);
            }
        printf ("Send reorder form to %.20s\n", SUB_BUF.CUST);
        STATUS = BTRV (B_GETPR, POS_BLK, &SUB_BUF, &BUF_LEN, KEY_BUF, 2);
    }
```

C GET LESS THAN OR EQUAL

In the following example, an application uses the Get Less Than Or Equal operation to retrieve the first house that falls within a prospective customer's price limit of $110,000.

```
#define B_GETLE 11

struct  HOME_REC
    {
        char    PRICE[7];
        char    ADDRESS[20];
        char    SQ_FEET[6];
        char    YR_BUILT[4];
    };

int             BUF_LEN;
struct HOME_REC  HOME_BUF;
char            KEY_BUF[] = "0110000";
char            POS_BLK[128];
int             STATUS;

main ()
{
BUF_LEN = sizeof (HOME_BUF);
STATUS = BTRV (B_GETLE, POS_BLK, &HOME_BUF, &BUF_LEN, KEY_BUF, 0);
if (STATUS != 0)
    printf ("Error reading file. Status = %d", STATUS);
else
    printf ("The home at %.20s sells for %.7s", HOME_BUF.ADDRESS, HOME_BUF.PRICE);
}
```

After the Get Less Than or Equal operation, Btrieve's current position is as follows:

0050000	330 N. 31st	002200	1960	
0055000	11132 Maple Ave.	002000	1965	
0070000	624 Church Street	002300	1968	← Previous Record
0105000	3517 N. Lakes Avenue	002500	1975	← Current Record
0220000	4500 Oceanfront Ave.	003000	1980	← Next Record

↑ Access Path

C GET NEXT

In the following example, an application uses the Get Next operation to generate a set of mailing labels sorted according to zip code. The zip code (ZIP) is key 1 in the file.

```
#define B_GETFST 12
#define B_GETNX  6
#define EOF_ERR  9

struct   ADDRESS
    {
        char    NAME[20];
        char    STREET[20];
        char    CITY[10];
        char    STATE[2];
        char    ZIP[5];
    };

int            DATA_LEN;
struct ADDRESS    ADDR_BUF;
char           KEY_BUF[5];
char           POS_BLK[128];
int            STATUS;

main ()
{
DATA_LEN = sizeof (ADDR_BUF);
STATUS = BTRV (B_GETFST, POS_BLK, &ADDR_BUF, &DATA_LEN, KEY_BUF, 1);
while (STATUS != EOF_ERR)
    {
        if (STATUS != 0)
            {
                printf ("Error reading file. Status = %d", STATUS);
                exit (0);
            }
        printf ("%.20s\n", ADDR_BUF.NAME);
        printf ("%.20s\n", ADDR_BUF.STREET);
        printf ("%.10s, %.2s %.5s\n\n", ADDR_BUF.CITY, ADDR_BUF.STATE, ADDR_BUF.ZIP);
        STATUS = BTRV (B_GETNX, POS_BLK, &ADDR_BUF, &DATA_LEN, KEY_BUF, 1);
    }
```

C GET POSITION

The following example illustrates how Get Position can be used to construct an external index for an existing Btrieve file. Once an external index is created, the application can read the external index file from lowest to highest and use Get Direct to sort the records in a Btrieve file by some field that was not originally defined as a key field.

```
#define B_INSERT  2
#define B_GETFST 12
#define B_GETNX  6
#define B_GETPOS 22
#define EOF_ERR  9

struct ADDRESS
    {
        char    NAME[20];
        char    STREET[20];
        char    CITY[10];
        char    STATE[2];
        char    ZIP[5];
    };

union   ADDR_REC
    {
        struct  ADDRESS ADDR;
        long    REC_POS;
    }   FILE_BUF;

struct INDEX
    {
        long    INX_POS;
        char    INX_ST[2];
    };

int         FILE_LEN;
char        FIL_POS[128];
int         I;
struct  INDEX  INDX_BUF;
int         INDX_LEN;
char        INX_POS[128];
char        NAME_KEY[20];
char        ST_KEY[2];
int         STATUS;
```

```
main ()
{
FILE_LEN = sizeof (FILE_BUF);
INDX_LEN = sizeof (INDX_BUF);
STATUS = BTRV (B_GETFST, FIL_POS, &FILE_BUF, &FILE_LEN, NAME_KEY, 0);
while (STATUS != EOF_ERR)
   {
      if (STATUS != 0)
         {
            printf ("Error reading file. Status = %d", STATUS);
            exit (0);
         }
      for (I = 0; I < 2; I++) INDX_BUF.INX_ST[I] = FILE_BUF.ADDR.STATE[I];
      STATUS = BTRV (B_GETPOS, FIL_POS, &FILE_BUF, &FILE_LEN, NAME_KEY, 0);
      INDX_BUF.INX_POS = FILE_BUF.REC_POS;
      STATUS = BTRV (B_INSERT, INX_POS, &INDX_BUF, &INDX_LEN, ST_KEY, 0);
      if (STATUS != 0)
         {
            printf ("Error inserting index record.  Status = %d", STATUS);
            exit (0);
         }
      STATUS = BTRV (B_GETNX, FIL_POS, &FILE_BUF, &FILE_LEN, NAME_KEY, 0);
   }
```

C GET PREVIOUS

The following example illustrates how an application can use Get Previous to list corporations and their total sales dollars for the year, beginning with the corporation having the highest sales and continuing in descending order of sales dollars. Total sales is key number 1 in the company file.

```
#define B_GETLST 13
#define B_GETPR  7
#define EOF_ERR  9

struct   COMPANY
    {
        char     NAME[30];
        char     TOT_SALE[10];
    };

struct COMPANY    COMP_BUF;
int          DATA_LEN;
char         KEY_BUF[10];
char         POS_BLK[128];
int          STATUS;

main ()
{
DATA_LEN = sizeof (COMP_BUF);
STATUS = BTRV (B_GETLST, POS_BLK, &COMP_BUF, &DATA_LEN, KEY_BUF, 1);
while (STATUS != EOF_ERR)
    {
        if (STATUS != 0)
            {
                printf ("Error reading file. Status = %d", STATUS);
                exit (0);
            }
        printf ("%.30s  %.10s\n", COMP_BUF.NAME, COMP_BUF.TOT_SALE);
        STATUS = BTRV (B_GETPR, POS_BLK, &COMP_BUF, &DATA_LEN, KEY_BUF, 1);
    }
```

C INSERT

The following example shows how an application might use the Insert operation to add a new employee to the employee file.

```
#define B_INS  2
struct EMP_REC
    {
        char    NAME[20];
        char    HIRE_DAT[6];
        char    ANNL_SAL[6];
    };

• int       DATA_LEN;
struct EMP_REC    EMP_BUF;
int       I;
char      KEY_BUF[20];
char      NEW_HIRE[] = "120882";
char      NEW_NAME[] = "Jones, Mary E.      ";
char      NEW_SAL[] = "020000";
char      POS_BLK[128];
int       STATUS;
main ()
{
DATA_LEN = sizeof (EMP_BUF);
for (I = 0; I < sizeof(EMP_BUF.HIRE_DAT); I++) EMP_BUF.HIRE_DAT[I] = NEW_HIRE[I];
for (I = 0; I < sizeof(EMP_BUF.NAME); I++) EMP_BUF.NAME[I] = NEW_NAME[I];
for (I = 0; I < sizeof(EMP_BUF.ANNL_SAL); I++) EMP_BUF.ANNL_SAL[I] = NEW_SAL[I];
STATUS = BTRV (B_INS, POS_BLK, &EMP_BUF, &DATA_LEN, KEY_BUF, 0);
if (STATUS != 0) printf ("Error inserting record. Status = %d", STATUS);
}
```

After an Insert operation, Btrieve's current position in the file is as follows:

Adams, David H.	150781	030000	
Brown, William J.	010581	055000	← Previous record
Jones, Mary E.	120882	020000	← Current record
Smith, Bruce L.	100182	040000	← Next record

↑

Access Path

C OPEN

The following example illustrates the code required to open a Btrieve file from C.

```
#define B_OPEN 0

struct   EMP_REC
    {
         char     EMP_NUM[6];
         char     EMP_NAME[20];
         char     EMP_DEPT[2];
         char     EMP_TOT[6];
         char     EMP_CUR[6];
    };

int      BUF_LEN;
struct   EMP_REC  EMP_BUF;
char     POS_BLK[128];
int      STATUS;

main ()
{
BUF_LEN = sizeof (EMP_BUF);
STATUS = BTRV (B_OPEN, POS_BLK, &EMP_BUF, &BUF_LEN,
                         "C:\\DATA\\EMPLOYEE.BTR ", 1);
if (STATUS != 0)
    {
         printf ("Error opening file. Status = %d", STATUS);
         exit (0);
    }
```

C SET DIRECTORY

In the following example, an application sets the current directory before performing an Open operation.

```
#define B_OPEN      0
#define B_SET_DIR 17

struct ADDR_REC                                          /*Structure of address file entry*/
    {
        char    NAME[30];
        char    STREET[30];
        char    CITY[30];
        char    STATE[2];
        char    ZIP[5];
    };

int         BUF_LEN;
struct      ADDR_REC DATA_BUF;
char        DIR_PATH[6] = "\\DATA";
char        FIL_NAME[15] = "ADDRESS.BTR   ";
char        KEY_BUF[30];
char        POS_BLK[128];
int         STAT;

main ()
{
BUF_LEN = sizeof (DATA_BUF);
STAT = BTRV (B_SET_DIR, POS_BLK, &DATA_BUF, &BUF_LEN, DIR_PATH, 0);
if (STAT != 0)
    {
        printf ("Unable to set current directory.  STAT =%d", STAT);
        exit (0);
    };
STAT = BTRV (B_OPEN, POS_BLK, &DATA_BUF, &BUF_LEN, FIL_NAME, 0);
if (STAT != 0)
    {
        printf ("Error opening file. STAT = %d", STAT);
        exit (0);
    };
```

C STAT

In the following example, an application uses the Stat and Create operations to empty a Btrieve file.

```
#define B_CREATE 14
#define B_OPEN    0
#define B_CLOSE   1
#define B_STAT    15

struct KEY_SPEC
   {
      int     KEY_POS;
      int     KEY_LEN;
      int     KEY_FLAG;
      long    KEY_TOT;
      char    KEY_TYPE;
      char    RESERVE1[5];
   };

struct FIL_SPEC
   {
      int     REC_LEN;
      int     PAGE_SIZ;
      int     NDX_CNT;
      long    REC_TOT;
      char    RESERVED[6];
      int     FILE_FLAG;
      char    RESERVE2[2];
      int     UNUSED_P;
      struct  KEY_SPEC KEY_BUF[2];
   };

int         FIL_BUF_LEN;
struct      FIL_SPEC FIL_BUF;
char        FIL_NAME[15] = "LEDGER.BTR";
char        KEY_BUF[64];
char        POS_BLK[128];
int         STAT;
```

```
main ()
{
FIL_BUF_LEN = sizeof (FIL_BUF);
STAT = BTRV (B_OPEN, POS_BLK, &FIL_BUF, &FIL_BUF_LEN, FIL_NAME, 0);
if  (STAT != 0)
    {
        printf ("Error opening file. STAT = %d", STAT);
        exit (0);
    };
FIL_BUF_LEN = sizeof (FIL_BUF);
STAT = BTRV (B_STAT, POS_BLK, &FIL_BUF, &FIL_BUF_LEN, KEY_BUF, 0);
if (STAT != 0)
    {
        printf ("Error retrieving stats. STAT = %d", STAT);
        exit (0);
    };
STAT = BTRV (B_CLOSE, POS_BLK, &FIL_BUF, &FIL_BUF_LEN, FIL_NAME, 0);
if  (STAT != 0)
    {
        printf ("Error closing file. STAT = %d", STAT);
        exit (0);
    };
FIL_BUF_LEN = sizeof (FIL_BUF);
STAT = BTRV (B_CREATE, POS_BLK, &FIL_BUF, &FIL_BUF_LEN, FIL_NAME, 0);
if (STAT != 0)
    printf ("Error recreating file. STAT = %d", STAT);
else
    printf ("File emptied successfully");
}
```

C STEP FIRST

See Step Next.

C STEP LAST

See Step Previous.

C STEP NEXT

The following example shows how an application uses the Step First and Step Next operations to recover a file with damaged indexes.

```
#define B_INSERT   2
#define B_STEPNXT 24
#define B_STEPFST 33
#define EOF_ERR    9
struct EMP_REC
     {
          char  NUM[6];
          char  NAME[30];
          char  ADDR[50];
     };
int       BUF_LEN;
struct    EMP_REC NEW_BUF;
char      NEW_KEY[6];
char      NEW_POS[128];
struct    EMP_REC OLD_BUF;
char      OLD_KEY[6];
char      OLD_POS[128];
int       I;
int       STATUS;
main ()
{
BUF_LEN = sizeof (NEW_BUF);
STATUS = BTRV (B_STEPFST, OLD_POS, &OLD_BUF, &BUF_LEN, OLD_KEY, 0);
while (STATUS != EOF_ERR)
     {
          if (STATUS != 0)
               {
                    printf ("Error reading file. Status = %d", STATUS);
                    exit (0);
               }
          for (I = 0; I < 6; I++) NEW_BUF.NUM[I] = OLD_BUF.NUM[I];
          for (I = 0; I < 30; I++) NEW_BUF.NAME[I] = OLD_BUF.NAME[I];
          for (I = 0; I < 50; I++) NEW_BUF.ADDR[I] = OLD_BUF.ADDR[I];
          STATUS = BTRV ( B_INSERT, NEW_POS, &NEW_BUF, &BUF_LEN, NEW_KEY, 0);
          if (STATUS != 0)
               {
                    printf ("Error inserting record. Status = %d", STATUS);
                    exit (0);
               }
          STATUS = BTRV (B_STEPNXT, OLD_POS, &OLD_BUF, &BUF_LEN, OLD_KEY, 0);
     }
```

C STEP PREVIOUS

This example shows how an application uses the Step Last and Step Previous operations to recover a file with damaged indexes.

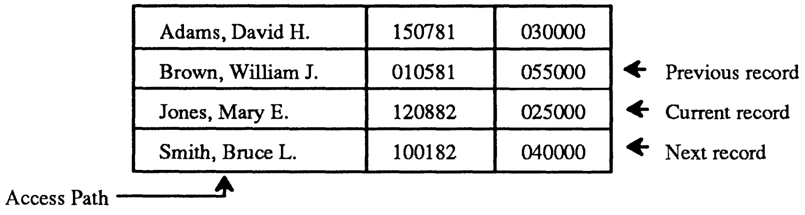
```
#define  B_INSERT   2
#define  B_STEPLST 34
#define  B_STEPPREV 35
#define  EOF_ERR     9
struct EMP_REC
      {
          char     NUM[6];
          char     NAME[30];
          char     ADDR[50];
      };
int         BUF_LEN;
struct      EMP_REC  NEW_BUF;
char        NEW_KEY[6];
char        NEW_POS[128];
struct      EMP_REC  OLD_BUF;
char        OLD_KEY[6];
char        OLD_POS[128];
int         I;
int         STATUS;

main ()
{
BUF_LEN = sizeof (NEW_BUF);
STATUS = BTRV (B_STEPLST, OLD_POS, &OLD_BUF, &BUF_LEN, OLD_KEY, 0);
while (STATUS != EOF_ERR)
      {
          if (STATUS != 0)
              {
                  printf ("Error reading file. Status = %d", STATUS);
                  exit (0);
              }
          for (I = 0; I < 6; I++) NEW_BUF.NUM[I] = OLD_BUF.NUM[I];
          for (I = 0; I < 30; I++) NEW_BUF.NAME[I] = OLD_BUF.NAME[I];
          for (I = 0; I < 50; I++) NEW_BUF.ADDR[I] = OLD_BUF.ADDR[I];
          STATUS = BTRV (B_INSERT, NEW_POS, &NEW_BUF, &BUF_LEN, NEW_KEY, 0);
          if (STATUS != 0)
              {
                  printf ("Error inserting record. Status = %d", STATUS);
                  exit (0);
              }
          STATUS = BTRV (B_STEPPREV, OLD_POS, &OLD_BUF, &BUF_LEN, OLD_KEY,0);
      }
```

C UPDATE

The following example shows how an application might use the Update operation to reflect the fact that an employee just received a raise.

```
#define B_GETEQ  5
#define B_UPD    3

struct   EMP_REC
    {
    char     NAME[20];
    char     HIRE_DAT[6];
    char     ANNL_SAL[6];
    };
int      DATA_LEN;
struct   EMP_REC  EMP_BUF;
int      I;
char     KEY_BUF[] = “Jones, Mary E.        ”;
char     NEW_SAL[] = “025000”;
char     POS_BLK[128];
int      STATUS;
main ()
{
DATA_LEN = sizeof (EMP_BUF);
STATUS = BTRV (B_GETEQ, POS_BLK, &EMP_BUF, &DATA_LEN, KEY_BUF, 0);
if (STATUS != 0)
    {
        printf (“Error reading file. Status = %d”, STATUS);
        exit (0);
    }
for (I = 0; I < sizeof(EMP_BUF.ANNL_SAL); I++) EMP_BUF.ANNL_SAL[I] = NEW_SAL[I];
STATUS = BTRV (B_UPD, POS_BLK, &EMP_BUF, &DATA_LEN, KEY_BUF, 0);
if (STATUS != 0) printf (“Error updating record. Status = %d”, STATUS);
}
```

After the Update operation, Btrieve's positioning in the file is as follows:



Adams, David H.	150781	030000	
Brown, William J.	010581	055000	← Previous record
Jones, Mary E.	120882	025000	← Current record
Smith, Bruce L.	100182	040000	← Next record

APPENDIX F: BASIC EXAMPLES

The following examples illustrate how to call Btrieve from a BASIC application. A different example is presented for each Btrieve operation.

BASIC ABORT TRANSACTION

The following example shows an application which inserts a record into an order header file and two records into an order detail file to record the sale of a box of diskettes and a ribbon to one customer. If an error occurs on any of the operations, Btrieve aborts the transaction so that the database remains consistent.

```
 60  FIELD #2, 2 AS ORDER.NUM$, 2 AS CUST.NUM$, 6 AS SALE.DATE$
 70  FIELD #3, 2 AS DET.ORD.NUM$, 2 AS PART.NUM$, 2 AS QUAN$
190  'Begin the transaction and insert the header record.
210  OPERATION% = 19                                  'Set Begin Transaction code
230  CALL BTRV (OPERATION%, STATUS%, DUMMY.FCB%, DUMMY.LEN%,
                     DUMMY.KEY$, KEY.NUMBER%)
240  IF STATUS% <> 0 THEN
     PRINT "Error beginning transaction =", STATUS% : END
250  LSET ORDER.NUM$ = MKI$(236)                                      'Order #236
260  LSET CUST.NUM$ = MKI$(201) : LSET SALE.DATE$ = "061583"
280  OPERATION% = 2 : BUF.LEN% = 10
290  CALL BTRV (OPERATION%, STATUS%, HDR.FCB%, BUF.LEN%,HDR.KEY$,
                     KEY.NUMBER%)
300  IF STATUS% <> 0 THEN
     PRINT "Status = ", STATUS%, "inserting header record" :
     OP%=21 : CALL BTRV(OP%, STATUS%, DUMMY.FCB%, DUMMY.LEN%,
                     DUMMY.KEY$, KEY.NUMBER%) : END
320  'Header record has been inserted, now insert 2 detail records
340  LSET DET.ORD.NUM$ = MKI$(236)                                    'Order #236
350  LSET PART.NUM$ = MKI$(1002)                 'Diskettes are part number 1002
360  LSET QUAN$ = MKI$(1)                                     'Purchased 1 box
365  BUF.LEN% = 6                                       'Set data buffer length
370  CALL BTRV (OPERATION%, STATUS%, DET.FCB%, BUF.LEN%,DET.KEY$,
                     KEY.NUMBER%)
```

```
380 IF STATUS% <> 0 THEN
    PRINT "Status = ",STATUS%,"inserting detail record":
    OP%=21 : CALL BTRV(OP%, STATUS%, DUMMY.FCB%, DUMMY.LEN%,
                    DUMMY.KEY$, KEY.NUMBER%) : END
390 LSET PART.NUM$ = MKI$(1024)                    'Ribbon is part number 1024
400 LSET QUAN$ = MKI$(1)                                   'Purchased 1 ribbon

410 CALL BTRV (OPERATION%, STATUS%, DET.FCB%, BUF.LEN%, DET.KEY$,
                    KEY.NUMBER%)
420 IF STATUS% <> 0 THEN
    PRINT "Status = ",STATUS%, "inserting detail record": OP%=21 :
    CALL BTRV(OP%, STATUS%, DUMMY.FCB%, DUMMY.LEN%, DUMMY.KEY$,
                    KEY.NUMBER%) : END
430 '
440 'All records have been inserted. End the transaction.
450 '
460 OPERATION% = 20                                'Set end transaction code
470 CALL BTRV (OPERATION%, STATUS%, DUMMY.FCB%, DUMMY.LEN%,
                    DUMMY.KEY$,  KEY.NUMBER%)
480 IF STATUS% <> 0 THEN
            PRINT "Error ending transaction. Status= ", STATUS%
    ELSE PRINT "Transaction completed successfully."
```

BASIC BEGIN TRANSACTION

See the examples for Abort Transaction and End Transaction.

BASIC CLEAR OWNER

The following example illustrates the code required to perform a Clear Owner operation.

```
100 OP% = 30
110 CALL BTRV (OP%, STATUS%, FILE.PTR%, BUF.LEN%, KEY.VAL$,  KEY.NUM%)
120 IF STATUS% <> 0 THEN PRINT "Btrieve status = ", STATUS% : END
```

BASIC CLOSE

The following example illustrates the code required to perform a Close operation:

```
110 OP% = 1
170 CALL BTRV (OP%, STATUS%, FILE.PTR%, BUF.LEN%, KEY.VAL$,
              KEY.SELECT%)
175 IF STATUS% <> 0 THEN PRINT "Btrieve status = ", STATUS% :END
```

BASIC CREATE

In the following example, an application creates a Btrieve file with two keys. Key 0 is an integer key, 2 bytes long. Key 1 allows duplicates, is modifiable, and consists of two segments. The first segment is a 2-byte string, to be sorted in descending order, and the second is a zero terminated 30-byte string. The records have a fixed length of 80.

```
20  DUPLICATES% = 1
30  MODIFIABLE% = 2
40  SEGMENTED% = 16
50  DESCEND% = 64
60  EXTTYPE% = 256
70  BINTEGER% = 1
80  BSTRING%  = 0
90  BZSTRING% = 11
100 OPEN "NUL" AS #1
110 FIELD 1, 2 AS RECL$, 2 AS PGSZ$, 2 AS NKEY$,
    4  AS NREC$, 2 AS VARL$, 2 AS RES$,
    2  AS PREALLOC$, 2 AS POS1$, 2 AS LEN1$,
    2  AS FLG1$, 4 AS CNT1$, 1 AS KEYT1$,
    5  AS RES1$, 2 AS POS2$, 2 AS LEN2$,
    2  AS FLG2$, 4 AS CNT2$, 1 AS KEYT2$,
    5  AS RES2$, 2 AS POS3$, 2 AS LEN3$,
    2  AS FLG3$, 4 AS CNT3$, 1 AS KEYT3$,
    6  AS RES3$
120 LSET RECL$ = MKI$(80)
130 LSET PGSZ$ = MKI$(1024)
140 LSET NKEY$ = MKI$(2)
150 LSET VARL$ = MKI$(0)
160 LSET POS1$ = MKI$(1)
170 LSET LEN1$ = MKI$(2)
180 LSET FLG1$ = MKI$ (EXTTYPE%)
190 LSET KEYT1$ = MKI$ (BINTEGER%)
```

```
200 LSET POS2$ = MKI$(3)
210 LSET LEN2$ = MKI$(2)
220 LSET FLG2$ = MKI$ (DUPLICATES% + MODIFIABLE% +  SEGMENTED% +
                          EXTTYPE% + DESCEND%)
230 LSET KEYT2$ = MKI$(BSTRING%)
240 LSET POS3$ = MKI$(5)
250 LSET LEN3$ = MKI$(30)
260 LSET FLG3$ = MKI$ (DUPLICATES% + MODIFIABLE% + EXTTYPE%)
270 LSET KEYT3$ = MKI$(BZSTRING%)
280 OP% = 14
290 BUF.LEN% = 64
300 FILE.NAME$ = "\DATA\CREATE.TST "
310 FCB.ADDR% = VARPTR(#1)
320 CALL BTRV (OP%, STATUS%, FCB.ADDR%, BUF.LEN%, FILE.NAME$, KEY.NUMBER%)
330 IF STATUS% <> 0 THEN
    PRINT "INVALID STATUS", STATUS%
```

BASIC CREATE SUPPLEMENTAL INDEX

In the following example, an application adds an index to a Btrieve file. The first segment of the key is a 10-byte string. It allows duplicates and is modifiable. The second segment is a 2-byte integer.

```
20   DUPLICATES% = 1
30   MODIFIABLE% = 2
40   SEGMENTED% = 16
50   EXTTYPE% = 256
60   BINTEGER% = 1
70   BSTRING%  = 0
80   OPEN "NUL" AS #1
90   FIELD 1, 2 AS POS1$, 2 AS LEN1$, 2 AS FLG1$,
     4 AS CNT1$, 1 AS KEYT1$, 5 AS RES1$,
     2 AS POS2$, 2 AS LEN2$, 2 AS FLG2$,
     4 AS CNT2$, 1 AS KEYT2$, 5 AS RES2$
100  OP% = 0 '                                                  Set open code
110  BUF.LEN% = 32
120  FILE.NAME$ = "\DATA\CREATE.TST "
130  FCB.ADDR% = VARPTR(#1)
140  CALL BTRV (OP%, STATUS%, FCB.ADDR%, BUF.LEN%, FILE.NAME$, KEY.NUMBER%)
150  IF STATUS%  0 THEN
    PRINT "INVALID STATUS ON OPEN = " STATUS%

160  LSET POS1$ = MKI$(40) : LSET LEN1$ = MKI$(10)
180  LSET FLG1$ = MKI$ (DUPLICATES% + MODIFIABLE% + SEGMENTED% + EXTTYPE%)
190  LSET KEYT1$ = MKI$ (BSTRING%)
```

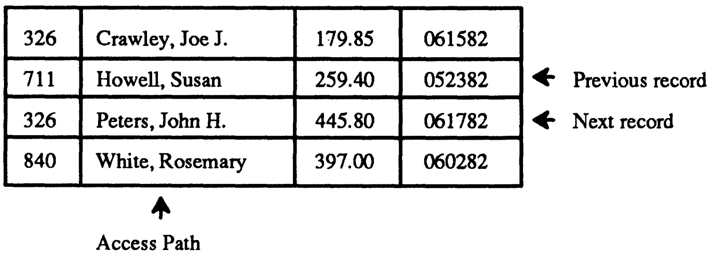
```
200 LSET POS2$ = MKI$(50) : LSET LEN2$ = MKI$(2)
220 LSET FLG2$ = MKI$ (DUPLICATES% + MODIFIABLE% + EXTTYPE%)
230 LSET KEYT2$ = MKI$ (BINTEGER%)
240 DUMYK$ = SPACE$(10)                                   ' Initialize key buffer
250 BUF.LEN% = 32                                         'Set data buffer length
260 OP% = 31                                   'Set create supplemental index code
270 CALL BTRV (OP%, STATUS%, FCB.ADDR%, BUF.LEN%, DUMYK$, KEY.NUMBER%)
280 IF STATUS% 0 THEN
    PRINT "INVALID STATUS ON CREATE SUPPLEMENTAL INDEX = ",STATUS%
```

BASIC DELETE

The sample below shows how an application for an airline service deletes a passenger reservation from the file.

```
110 OP% = 5
130 FILE.PTR% = VARPTR(#2)               'Point to position block of second open file
140 KEY.SELECT% = 0                                              ' Set key number
145 KEY.VAL$ = "Martin, Dave H."                                 'Set key value
150 FIELD #2, 3 AS FLT.NO$,
    15 AS PASNGR$,
    6 AS AMNT.PD$,
    6 AS ISSU.DATE$
160 BUF.LEN% = 30                                         'Set data buffer length
170 CALL BTRV (OP%, STATUS%, FILE.PTR%, BUF.LEN%, KEY.VAL$, KEY.SELECT%)
175 IF STATUS% <> 0 THEN
    PRINT "Btrieve status = ", STATUS% : END
190 OP% = 4
350 CALL BTRV (OP%, STATUS%, FILE.PTR%, BUF.LEN%, KEY.VAL$, KEY.SELECT%)
360 IF STATUS% <> 0 THEN
    PRINT "Btrieve status = ", STATUS% : END
```

After the Delete operation, Btrieve's position in the file is as follows:



326	Crawley, Joe J.	179.85	061582	
711	Howell, Susan	259.40	052382	← Previous record
326	Peters, John H.	445.80	061782	← Next record
840	White, Rosemary	397.00	060282	

Access Path

BASIC DROP SUPPLEMENTAL INDEX

In the following example, an application drops a supplemental index in a Btrieve file because the file no longer needs to be accessed by that index.

```
 40  BUF.LEN% = 1
 50  FCB.ADDR% = VARPTR(#1)
 60  OP% = 32
 70  KEY.NUMBER% = 3
 80  DUMYK$ = SPACE$(10)
 90  CALL BTRV (OP%, STATUS%, FCB.ADDR%, BUF.LEN%, DUMYK$, KEY.NUMBER%)
100  IF STATUS% = 6 THEN
                PRINT "KEY NUMBER TO DROP IS NOT A SUPPLEMENTAL INDEX"
     ELSE IF STATUS% <> 0 THEN
                PRINT "INVALID STATUS ON DROP SUPPLEMENTAL INDEX = " STATUS%
```

BASIC END TRANSACTION

The following application uses transaction control to ensure that the application does not debit one account in a ledger file unless it credits another account.

```
 50  FIELD #2, 2 AS ACCT.ID$,40 AS DESC$, 4 AS BALANCE$
130  'Begin the transaction
150  OPERATION% = 19                                  'Set Begin Transaction code
170  CALL BTRV (OPERATION%, STATUS%, DUMMY.FCB%, DUMMY.LEN%,
                    DUMMY.KEY$, KEY.NUMBER%)
180  IF STATUS% <> 0 THEN
     PRINT "Error beginning transaction =", STATUS%: END
190                                      'Retrieve and update the cash account record.
210  OPERATION% = 5 : KEY.BUF$ = MKI$(101)                    'Cash account is 101
230  BUF.LEN% = 46
240  CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%, KEY.BUF$,
                    KEY.NUMBER%)
250  IF STATUS% <> 0 THEN
     PRINT "Status = ", STATUS%,"retrieving record":
     OP%=21 : CALL BTRV(OP%, STATUS%, DUMMY.FCB%, DUMMY.LEN%,
                    DUMMY.KEY$,  KEY.NUMBER%) : END
260  LSET BALANCE$ = MKS$(CVS(BALANCE$) – 250)        'Remove $250 from account
270  OPERATION% = 3                                       'Set Update operation
280  CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%,  KEY.BUF$,
                    KEY.NUMBER%)
290  IF STATUS% <> 0 THEN PRINT "Status = ", STATUS%,"updating cash account":
   OP%=21 : CALL BTRV(OP%, STATUS%, DUMMY.FCB%, DUMMY.LEN%,
                    DUMMY.KEY$, KEY.NUMBER%) : END
```

```
310 'Retrieve and update the office expense record. The office expense account is 511
335 OPERATION% = 5 : KEY.BUF$ = MKI$ (511) : BUF.LEN% = 46
360 CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%, KEY.BUF$,
                        KEY.NUMBER%)
370 IF STATUS% <> 0 THEN
    PRINT "Status = ", STATUS%, "retrieving record" :
  OP%=21 : CALL BTRV(OP%, STATUS%, DUMMY.FCB%, DUMMY.LEN%,
                        DUMMY.KEY$, KEY.NUMBER%) : END
380 LSET BALANCE$ = MKS$(CVS(BALANCE$) + 250)          'Add $250 to office expense
390 OPERATION% = 3                                          'Set Update operation
400 CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%, KEY.BUF$,
                        KEY.NUMBER%)
410 IF STATUS% <> 0 THEN
    PRINT "Status = ",STATUS%,"updating office expense" :
    OP%=21 : CALL BTRV(OP%, STATUS%, DUMMY.FCB%, DUMMY.LEN%,
                        DUMMY.KEY$, KEY.NUMBER%) : END
415 OPERATION% = 20                                    'Set End Transaction code
420 CALL BTRV (OPERATION%, STATUS%, DUMMY.FCB%, DUMMY.LEN%,
                        DUMMY.KEY$,  KEY.NUMBER%)
430 IF STATUS% <> 0 THEN
            PRINT "Error ending transaction. Status= ", STATUS%
    ELSE PRINT "Transaction completed successfully."
```

BASIC EXTEND

The following example illustrates how an application might use the Extend operation to expand a Btrieve file to gain more disk space.

```
40  OPEN "NUL" AS #2                                     'Open file from BASIC
50  FIELD #2, 30 AS FULL.NAME$,
    30 AS STREET$,
    30 AS CITY$,
    2 AS STATE$,
    6 AS ZIP$
60  OPERATION% = 0 : STATUS% = 0                  'Set open op code and status
70  FCB.ADDR% = VARPTR(#2)                                'Get address of FCB
90  KEY.BUFFER$ = "ADDRESS.BTR " : KEY.NUM% = 0    'Set key buffer and key number
100 CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%, KEY.BUFFER$,
                        KEY.NUM%)
120 IF STATUS% <> 0 THEN
    PRINT "Error opening file. Status = ", STATUS% : END
130 OPERATION% = 16 : KEY.BUFFER$ = "B:\ADDRESS.EXT "
150 CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%, KEY.BUFFER$,
                        KEY.NUM%)
```

```
160 IF STATUS% <> 0 THEN
    PRINT "Error extending the file. Status = ", STATUS% : END
170 OPERATION% = 1                                        'Set close operation code
180 CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%, KEY.BUFFER$,
                  KEY.NUM%)
190 OPERATION% = 0 : KEY.BUFFER$ = "ADDRESS.BTR "
200 CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%, KEY.BUFFER$,
                  KEY.NUM%)
210 IF STATUS% <> 0 THEN
    PRINT "Error reopening the file. Status = ", STATUS% : END
```

BASIC GET DIRECT

The following example illustrates how an application can use the Get Direct operation to sort the records in a Btrieve file by an external index. (See the description of Get Position for an example of how to build the external index file.)

```
 40  FIELD #1, 20 AS NAM$, 20 AS STREET$, 10 AS CITY$,
     2 AS STATE$, 5 AS ZIP$
 50  FIELD #1, 4 AS REC.POSITION$
 60  FIELD #2, 4 AS POS.INX$, 2 AS STATE.INX$
 70  KEY.NUM% = 0                                       'Use key 0 for both files
 80  FILE.PTR% = VARPTR(#1)                                  'Original file's FCB
 85  FILE.LEN% = 57                            'Set data buffer length for File #1
 90  INDX.PTR% = VARPTR(#2)                                     'Index file's FCB
 95  INDX.LEN% = 6                             'Set data buffer length for File #2
100  NAME.KEY$ = SPACE$(20) : STAT.KEY$ = SPACE$(2)
120  READ.LOW% = 12 : READ.NEXT% = 6 : GET.DIRECT% = 23
130  CALL BTRV (READ.LOW%, STATUS%, INDX.PTR%, INDX.LEN%, STAT.KEY$,
                   KEY.NUM%)
140  WHILE STATUS% <> 9                                    'Read until end of file
150  IF STATUS% <> 0 THEN
     PRINT "Error reading file. Status = ", STATUS% : END
160  LSET REC.POSITION$ = POS.INX$
170  CALL BTRV (GET.DIRECT%, STATUS%, FILE.PTR%, FILE.LEN%, NAME.KEY$,
                   KEY.NUM%)
180  IF STATUS% <> 0 THEN
     PRINT "Error retrieving record. Status = ", STATUS% : END
190  PRINT NAM$, STREET$, CITY$, STATE$, ZIP$
210  CALL BTRV (READ.NEXT%, STATUS%, INDX.PTR%, INDX.LEN%, STAT.KEY$,
                   KEY.NUM%)
220  WEND
```

BASIC GET DIRECTORY

The following example illustrates how an application can use the Get and Set Directory operations to retrieve the current directory at the beginning of the program, and to restore it before terminating.

```
40  OPERATION% = 18                                              'Set get directory operation
50  STATUS% = 0                                                             'Initialize status
60  DIR.PATH$ = SPACE$(65)                                               'Initialize key buffer
70  CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%, DIR.PATH$,
                        KEY.NUMBER%)
80  IF STATUS% <> 0 THEN
    PRINT "Error getting directory. Status = ", STATUS% : END
90  OPERATION% = 17                                          'Set current directory operation
100 CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%, DIR.PATH$,
                        KEY.NUMBER%)
110 IF STATUS% <> 0 THEN
    PRINT "Error restoring current dir. Status = ", STATUS% : END
```

BASIC GET EQUAL

The following example shows how an application retrieves a part number from an inventory file using the Get Equal operation.

```
110 FIELD #2, 5 AS PART.NUM$,
    10 AS PART.DESC$,
    3 AS QUAN.ON.HAND$,
    3 AS REORDER.POINT$,
    3 AS REORDER.QUAN$
140 FCB.ADDR% = VARPTR(#2)                                      'Get address of FCB
150 OPERATION% = 5                                             'Set get equal op code
160 PART.NUMBER$ = "03426"                                       'Part number to find
170 KEY.NUMBER% = 0                                             'Part number is key 0
180 BUF.LEN% = 24
350 CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%,
                  PART.NUMBER$, KEY.NUMBER%)                        'Retrieve record
360 IF STATUS% <> 0 THEN
    PRINT "Error reading file. Status = ", STATUS% : END
370 IF QUAN.ON.HAND$ < REORDER.POINT$ THEN
    PRINT "Time to order ", REORDER.QUAN$, "units of ", PART.DESC$
```

The table below shows Btrieve's current position in the file after the Get Equal operation.

03419	Pliers	003	010	015	← Previous record
03426	Hammer	010	003	005	← Current record
03430	Saw	005	002	003	← Next record
03560	Wrench	008	005	005	

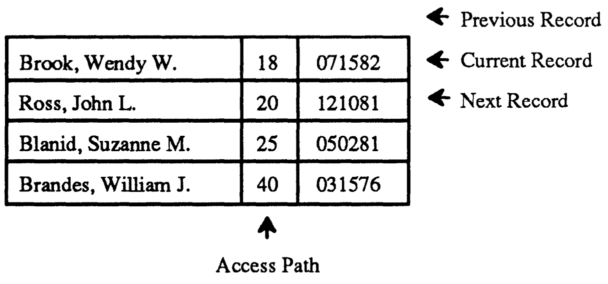
↑
Access Path

BASIC GET FIRST

The following example illustrates how an application uses the Get First operation to find the youngest employee in the company. Age is key number 2 in the employee file.

```
300 FIELD #1, 20 AS EMP.NAME$,
    2 AS AGE$,
    6 AS DATE.OF.HIRE$
310 FCB.ADDR% = VARPTR(#1)                                   'Get address of FCB
315 BUF.LEN% = 28
320 OPERATION% = 12                                          'Set Get First op code
330 KEY.NUM% = 2                                                     'Age is key 2
340 KEY.BUF$ = SPACE$(2)                                       'Initialize key buffer
350 CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%, KEY.BUF$,
                        KEY.NUM%)
360 IF STATUS% <> 0 THEN
    PRINT "Error reading file. Status = ", STATUS% : END
370 PRINT "Youngest employee is " EMP.NAME$
```

After the Get First operation, Btrieve's current position in the file is as follows:



Brook, Wendy W.	18	071582
Ross, John L.	20	121081
Blanid, Suzanne M.	25	050281
Brandes, William J.	40	031576

BASIC GET GREATER

The following example of an insurance company application uses the Get Greater operation to determine which policy holders have more than three traffic violations. The number of traffic violations is the second key in the policy file.

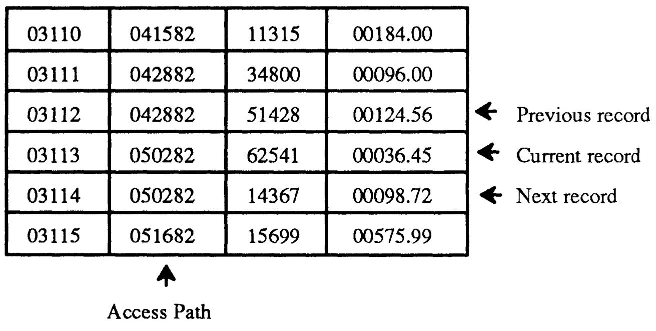
```
110 FIELD #2, 10 AS POLICY.NUM$, 20 AS NAM$, 6 AS EFFECT.DATE$,
                        2 AS VIOLATIONS$
140 FCB.ADDR% = VARPTR(#2)                                   'Get address of FCB
150 OPERATION% = 8 : BUF.LEN%=38               'Set op code and data buffer length
170 KEY.NUM% = 2                                            'Violations is key 2
180 KEY.BUF$ = "03"                                         'Start search above 3
190 CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%, KEY.BUF$,
                        KEY.NUM%)
200 IF STATUS% <> 0 THEN
    PRINT "Error reading file. Status = ", STATUS% : END
210 OPERATION% = 6                                          'Set get next op code
220 WHILE STATUS% <> 9                                      'Read until end of file
230 PRINT NAM$; "has "; VIOLATIONS$; " traffic violations"
240 CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%, KEY.BUF$,
                        KEY.NUM%)
250 IF (STATUS% <> 0) AND (STATUS% <> 9) THEN
    PRINT "Error reading file. Status = ", STATUS% : END
260 WEND
```

BASIC GET GREATER OR EQUAL

If the date of a sale is a key in an invoice file, an application might use the Get Greater Or Equal operation to retrieve the invoice record for the first sale made in May, 1982.

```
110 FIELD #1, 5 AS INVOICE.NUM$,
    6 AS DATE.OF.SALE$,
    5 AS CUST.NUM$,
    8 AS TOTAL.PRICE$
140 FCB.ADDR% = VARPTR(#1)                                    'Get address of FCB
150 OPERATION% = 9                                 'Set Get Greater Or Equal op code
155 BUF.LEN% = 24                                            'Set data buffer length
160 FIRST.DATE$ = "050182"                               'Start search at May 1, 1982
170 KEY.NUMBER% = 1                                           'Date of sale is key 1
350 CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%,
                   FIRST.DATE$,KEY.NUMBER%)                         'retrieve record
360 IF STATUS% <> 0 THEN
    PRINT "Error reading file. Status = ", STATUS% : END
370 PRINT "First sale in May was to", CUST.NUM$, " for ",TOTAL.PRICE$
```

After the Get Greater Or Equal Operation, Btrieve's current position in the file is as follows:



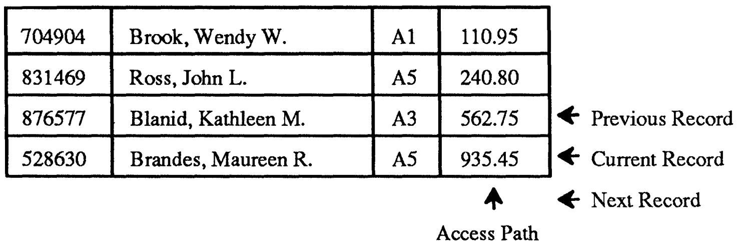
03110	041582	11315	00184.00
03111	042882	34800	00096.00
03112	042882	51428	00124.56
03113	050282	62541	00036.45
03114	050282	14367	00098.72
03115	051682	15699	00575.99

BASIC GET LAST

The following example illustrates how to determine which employee had the highest commissions last month.

```
110 FIELD #2, 6 AS EMP.NUM$,
    20 AS EMP.NAME$,
    2 AS DEPT$,
    6 AS TOT.COM$,
    6 AS CUR.COM$
140 FCB.ADDR% = VARPTR(#2)                                'Get address of FCB
150 OPERATION% = 13 : BUF.LEN% = 40         'Set Get Last op code and data buffer length
160 KEY.NUM% = 1 : KEY.BUF$ = SPACE$(6)          'Set key number and key buffer length
180 CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%, KEY.BUF$,
                     KEY.NUM%)
190 IF STATUS% <> 0 THEN
    PRINT "Error reading file. Status = ", STATUS% : END
200 PRINT "Employee with highest commissions last month was", EMP.NAME$
```

After the Get Last operation, Btrieve's current positioning in the file is as follows:



704904	Brook, Wendy W.	A1	110.95	
831469	Ross, John L.	A5	240.80	
876577	Blanid, Kathleen M.	A3	562.75	← Previous Record
528630	Brandes, Maureen R.	A5	935.45	← Current Record
			↑ Access Path	← Next Record

BASIC GET LESS THAN

The following example uses the Get Less Than operation followed by a Get Previous operation to find the names of all customers whose magazine subscriptions have less than three issues left before they expire. The number of issues remaining is key 2 in the subscription file.

```
110 FIELD #2, 20 AS CUST.NAME$,
    6 AS DATE.SUBSCRIBED$,
    6 AS DATE.PAID$,
    3 AS ISSUES.PURCHASED$,
    3 AS ISSUES.REMAINING$
140 FCB.ADDR% = VARPTR(#2)                             'Get address of FCB
150 OPERATION% = 10 : BUF.LEN% = 38         'Set op code and data buffer length
160 KEY.NUM% = 2                                    'Issues remaining is key 2
180 KEY.BUF$ = "003"                                   'Start search below 3
190 CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%, KEY.BUF$,
                        KEY.NUM%)
200 IF STATUS% <> 0 THEN
    PRINT "Error reading file. Status = ", STATUS% : END
210 OPERATION% = 7                                  'Set Get Previous op code
220 WHILE STATUS% <> 9                                'Read until start of file
230 PRINT "Send reorder form to", CUST.NAME$
240 CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%, KEY.BUF$,
                        KEY.NUM%)
250 IF (STATUS% <> 0) AND (STATUS% <> 9) THEN
    PRINT "Error reading file. Status = ", STATUS% : END
260 WEND
```

BASIC GET LESS THAN OR EQUAL

In the following example, the application uses the Get Less Than Or Equal operation to retrieve the first house from a file of homes for sale, that falls within the prospective customer's price limit of $110,000.

```
110 FIELD #1, 7 AS PRICE$,
    20 AS ADDRESS$,
    6 AS SQUARE.FEET$,
    4 AS YEAR.BUILT$
140 FCB.ADDR% = VARPTR(#1)                               'Get address of FCB
150 OPERATION% = 11                          'Set Get Less Than Or Equal code
160 BUF.LEN% = 37                                      'Set data buffer length
170 KEY.NUM% = 0                                               'Price is key 0
180 KEY.BUF$ = "0110000"                                'Start search at 110,000
190 CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%, KEY.BUF$,
                  KEY.NUM%)
200 IF STATUS% <> 0 THEN
    PRINT "Error reading file. Status = ", STATUS% : END
210 PRINT "The home at ", ADDRESS$, " sells for ", PRICE$
```

After the Get Less Than or Equal operation, Btrieve's current position is as follows:

0050000	330 N. 31st	002200	1960	
0055000	11132 Maple Ave.	002000	1965	
0070000	624 Church Street	002300	1968	← Previous Record
0105000	3517 N. Lakes Avenue	002500	1975	← Current Record
0220000	4500 Oceanfront Ave.	003000	1980	← Next Record

↑

Access Path

BASIC GET NEXT

In the following example the zip code is the first key. The application uses Get Next to generate a set of mailing labels sorted according to zip code.

```
110 FIELD #3, 20 AS NAM$,
    20 AS STREET$,
    10 AS CITY$,
    2 AS STATE$,
    5 AS ZIP$
140 FCB.ADDR% = VARPTR(#3)                                    'Get address of FCB
150 OPERATION% = 12 : BUF.LEN% = 57         'Get First to establish current position
160 KEY.NUM% = 1                                                 'Zip code is key 1
170 KEY.VALUE$ = SPACE$(5)                                     'Initialize key buffer
350 CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%,
                      KEY.VALUE$, KEY.NUM%)
360 IF STATUS% <> 0 THEN
    PRINT "Error reading address file. Status = ",STATUS% : END
370 OPERATION% = 6                                     'Set get next operation code
380 WHILE STATUS% <> 9                                    'Read until end of file
390 LPRINT FORM.FEED$                                          'Start new label
400 LPRINT NAM$                                                    'Print name
410 LPRINT STREET$                                                 'Print street
420 LPRINT CITY$, ",", STATE$, ZIP$                         'Print city and state
430 CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%,
                      KEY.VALUE$, KEY.NUM%)
440 IF (STATUS% <> 0) AND (STATUS% <> 9) THEN
    PRINT "Error reading address file. Status = ", STATUS% : END
450 WEND
```

BASIC GET POSITION

The following example illustrates how an application uses Get Position to construct an external index for an existing Btrieve file. Once an external index exists, the application can read the external index file from lowest to highest and use Get Direct to sort the records in a Btrieve file by some field that was not originally defined as a key field.

```
40  FIELD #1, 20 AS NAM$,
    20 AS STREET$,
    10 AS CITY$,
    2 AS STATE$,
    5 AS ZIP$
50  FIELD #1, 4 AS REC.POSITION$
60  FIELD #2, 4 AS POS.INX$,
    2 AS STATE.INX$
70  KEY.NUM% = 0                                          'Use key 0 for both files
80  FILE.PTR% = VARPTR(#1) : LEN1%=57                        'Original file's FCB
90  INDX.PTR% = VARPTR(#2) : LEN2%=6                            'Index file's FCB
100 NAME.KEY$ = SPACE$(20)                              'Key buffer for original file
110 STAT.KEY$ = SPACE$(2)                                'Key buffer for index file
120 READ.LOW% = 12 : READ.NEXT% = 6 : INSERT% = 2
125 GET.POS% = 22
130 CALL BTRV (READ.LOW%, STATUS%, FILE.PTR%, LEN1%, NAME.KEY$,
                     KEY.NUM%)
140 WHILE STATUS% <> 9                                    'Read until end of  file
150 IF STATUS% <> 0 THEN
    PRINT "Error reading file. Status = ", STATUS% : END
160 LSET STATE.INX$ = STATE$
170 CALL BTRV (GET.POS%, STATUS%, FILE.PTR%, LEN1%,  NAME.KEY$,
                     KEY.NUM%)
180 LSET POS.INX$ = REC.POSITION$
190 CALL BTRV (INSERT%, STATUS%, INDX.PTR%, LEN2%,  STAT.KEY$,
                     KEY.NUM%)
200 IF STATUS% <> 0 THEN
    PRINT "Error inserting record. Status = ", STATUS% : END
210 CALL BTRV (READ.NEXT%, STATUS%, FILE.PTR%, LEN1%, NAME.KEY$,
                     KEY.NUM%)
220 WEND
```

BASIC GET PREVIOUS

The following example lists corporations and their total sales dollars for the year, beginning with the corporation having the highest sales and continuing in descending order of sales dollars. Total sales is key number 1 in the company file.

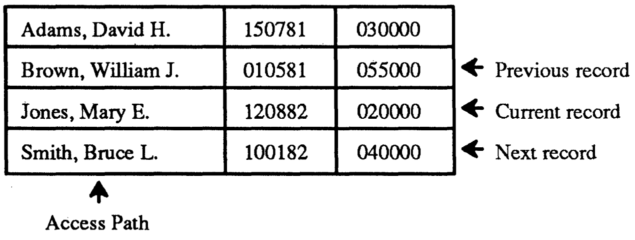
```
110 FIELD #1, 30 AS COMPANY$,
    10 AS TOTAL.SALES$
140 FCB.ADDR% = VARPTR(#1)                                  'Get address of FCB
145 BUF.LEN% = 40                                          'Set data buffer length
150 OPERATION% = 13                        'Get last to establish current position
160 KEY.NUM% = 1                                            'Total sales is key 1
170 KEY.VALUE$ = SPACE$(10)                           'Initialize key buffer length
350 CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%,
                   KEY.VALUE$, KEY.NUM%)
360 IF STATUS% <> 0 THEN
    PRINT "Error reading file. Status = ", STATUS% : END
365 'Set Get Previous op code and data buf length
370 OPERATION% = 7 : BUF.LEN% = 38
380 WHILE STATUS% <> 9                                     'Read until end of file
420 PRINT COMPANY$, TOTAL.SALES$
430 CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%,
                   KEY.VALUE$, KEY.NUM%)
440 IF (STATUS% <> 0) AND (STATUS% <> 9) THEN
    PRINT "Error reading file. Status = ", STATUS% : END
450 WEND
```

BASIC INSERT

The following example shows how to add a new employee to the employee file.

```
110 OP% = 2
130 FILE.PTR% = VARPTR(#2)
140 KEY.SELECT% = 0
150 FIELD #2, 20 AS NAM$,
    6 AS DATE.OF.HIRE$,
    6 AS ANNUAL.SAL$
160 LSET NAM$ = "Jones, Mary E."                       'LSET values into fields
180 LSET DATE.OF.HIRE$ = "120882"
185 LSET ANNUAL.SAL$ = "020000"
190 KEY.VAL$ = SPACE$(20)                               'Initialize key buffer
200 BUF.LEN% = 32                                      'Set data buffer length
350 CALL BTRV (OP%, STATUS%, FILE.PTR%, BUF.LEN%, KEY.VAL$,
                        KEY.SELECT%)
360 IF STATUS% <> 0 THEN
    PRINT "Btrieve status = ", STATUS% : END
```

After an Insert operation, Btrieve's current position in the file is as follows:



Adams, David H.	150781	030000	
Brown, William J.	010581	055000	← Previous record
Jones, Mary E.	120882	020000	← Current record
Smith, Bruce L.	100182	040000	← Next record

BASIC OPEN

The following example illustrates the code required to open a Btrieve file in an interpretative BASIC program:

```
.5  OPEN "SEGMENT.ADR" FOR INPUT AS #1          'lines 5, 10, and 20 are not required
10  INPUT #1, SEG.ADDR%                          'for a Compiled BASIC program
20  DEF SEG=SEG.ADDR%
30  OPEN "NUL" AS #2 LEN=92
40  FIELD #2, 6 AS EMP.NUM$,
    30 AS EMP.NAM$,
    6 AS HIRE.DATE$,
    50 AS ADDR$
45  BUF.LEN% = 92
50  OP% = 0
60  FILE.PTR% = VARPTR(#2)
65  KEY.VAL$ = "C:\DATA\EMPLOYE.BTR "
70  CALL BTRV (OP%, STATUS%, FILE.PTR%, BUF.LEN%, KEY.VAL$, KEY.SELECT%)
80  IF STATUS% <> 0 THEN
    PRINT "Btrieve status = ", STATUS% : END
```

BASIC RESET

The following example illustrates the code required to issue a Reset operation in a BASIC program:

```
100 OP% = 28
110 CALL BTRV (OPERATION%, STATUS%, FILE.PTR%, BUF.LEN%,
                    KEY.BUFFER$, KEY.NUM%)
120 IF STATUS% <> 0 THEN
    PRINT "Btrieve status = ", STATUS% : END
```

BASIC SET DIRECTORY

In the following example, an application sets the current directory before performing an Open operation.

```
 60  OPERATION% = 17                                              'Set Directory op code
 70  DIR.PATH$ = "\DATA"+CHR$(0)
 80  CALL BTRV (OPERATION%, STATUS%, DUMMY.FCB%, DUMMY.LEN%,
                        DIR.PATH$,  KEY.NUMBER%)
 90  IF STATUS% <> 0 THEN
     PRINT "Error setting current directory. Status = ", STATUS% : END
 92  OPEN "NUL" AS #2                                             'Open file from BASIC
 94  FIELD #2, 30 AS FULL.NAME$,
     30 AS STREET$,
     30 AS CITY$,
     2 AS STATE$,
     6 AS ZIP$
100  OPERATION% = 0                                             'Set Open operation code
110  STATUS% = 0                                                      'Initialize status
120  FCB.ADDR% = VARPTR(#2)                                       'Get address of FCB
125  BUF.LEN% = 98
130  KEY.BUFFER$ = "ADDRESS.BTR "                                'Initialize key buffer
140  KEY.NUMBER% = 0                                          'Use key 0 access path
150  CALL BTRV (OPERATION%, STATUS%, FCB.ADDR%, BUF.LEN%,
                        KEY.BUFFER$, KEY.NUMBER%)
160  IF STATUS% <> 0 THEN
   PRINT "Error opening file. Status = ", STATUS%  : END
```

BASIC SET OWNER

The following example shows how to set the owner name to "Payroll."

```
100  OP% = 29
110  FIELD #2, 8 AS OWNER$
120  LSET OWNER$ = "Payroll" + CHR$(0)
125  BUF.LEN% = 8
130  KEY.VAL$ = "Payroll" + CHR$(0)
135  KEY.NUM% = 0
140  CALL BTRV (OP%, STATUS%, FILE.PTR%, BUF.LEN%, KEY.VAL$, KEY.NUM%)
150  IF STATUS% <> 0 THEN
     PRINT "Btrieve status = ", STATUS% : END
```

BASIC STAT

In the following example, an application uses the Stat and Create operations to empty a Btrieve file.

```
 40  FIELD 2, 2 AS RECL$, 2 AS PGSZ$, 2 AS NKEY$,
     4 AS NREC$, 2 AS VARL$,
     2 AS RES$, 2 AS PREALLOC$,
     2 AS POS1$, 2 AS LEN1$,
     2 AS FLG1$, 4 AS CNT1$,
     1 AS KEYT1$, 5 AS RES1$,
     2 AS POS2$, 2 AS LEN2$,
     2 AS FLG2$, 4 AS CNT2$,
     1 AS KEYT2$, 5 AS RES2$
 50  FCB.ADDR% = VARPTR(#2)                                    'Set pointer to File #2
 60  BUF.LEN% = 48                                            'Set data buffer length
 70  OP% = 0                                                        'Set open code
 80  STATUS% = 0
 90  FILE.NAME$ = "LEDGER.BTR "
100  CALL BTRV (OP%, STATUS%, FCB.ADDR%, BUF.LEN%,  FILE.NAME$, KEY.NUMBER%)
110  OP% = 15                                              'Set Stat operation code
120  KEY.BUF$ = SPACE$(64)
130  BUF.LEN% = 48                         'Set key buffer and data buffer length
140  CALL BTRV (OP%, STATUS%, FCB.ADDR%, BUF.LEN%, KEY.BUF$, KEY.NUMBER%)
150  IF STATUS% <> 0 THEN
    PRINT "Error retrieving file stats.  Status = ", STATUS%
160  OP% = 1
170  CALL BTRV (OP%, STATUS%, FCB.ADDR%, BUF.LEN%, FILE.NAME$, KEY.NUMBER%)
180  IF STATUS% <> 0 THEN
    PRINT "Unable to close the file.  Status = ", STATUS%
190  OP% = 14 : BUF.LEN% = 48
210  CALL BTRV (OP%, STATUS%, FCB.ADDR%, BUF.LEN%, FILE.NAME$, KEY.NUMBER%)
220  IF STATUS% <> 0 THEN
     PRINT "Unable to create the file. Status = ", STATUS%
```

BASIC STEP FIRST

See Step Next.

BASIC STEP LAST

See Step Previous.

BASIC STEP NEXT

In the following example, the indexes for a file have been damaged by a power failure. The application uses the Step First and Step Next operations to recover the file.

```
30  OPEN "NUL" AS #1 LEN=86
40  FIELD #1, 6 AS EMP1.NUM$,
    30 AS EMP1.NAM$,
    50 AS ADDR1$
50  OP1% = 0                                        'Set open operation code
60  KEY.NUM1% = -2                                  'Open in recovery mode
60  FILE.PTR1%=VARPTR(#1) : BUF.LEN1% = 86   'Point to File#1, set data buf. length
70  KEY.VAL1$ = "C:\DATA\EMPLOYE.BTR "
80  CALL BTRV (OP1%, STATUS%, FILE.PTR1%, BUF.LEN1%, KEY.VAL1$, KEY.NUM1%)
90  IF STATUS% <> 0 THEN
    PRINT "Btrieve status = ", STATUS% : END
100 KEY.VAL1$ = SPACE$(6) : KEY.NUM1% = 0
110 OPEN "NUL" AS #2 LEN=86
120 FIELD #2, 6 AS EMP2.NUM$,
    30 AS EMP2.NAM$,
    50 AS ADDR2$
125 BUF.LEN2% = 86 : OP1% = 0                'Set data buffer length and op code
140 KEY.NUM2% = 0 : FILE.PTR2% = VARPTR(#2)  'Set open mode and point to File #2
150 KEY.VAL2$ = "C:\DATA\EMPLOYE2.BTR "
170 CALL BTRV (OP1%, STATUS%, FILE.PTR2%, BUF.LEN2%,  KEY.VAL2$, KEY.NUM2%)
180 IF STATUS% <> 0 THEN
    PRINT "Btrieve status = ",STATUS% : END
190 KEY.VAL2$ = SPACE$(6)                                'Initialize key buffer
200 OP2% = 33 : OP3% = 2 : OP4% = 24                          'Set op codes
220 CALL BTRV (OP2%, STATUS%, FILE.PTR1%, BUF.LEN1%,  KEY.VAL1$, KEY.NUM1%)
230 WHILE STATUS% <> 9                          'Read until end of damaged file
240 LSET EMP2.NUM$ = EMP1.NUM$               'Format new record with old data
250 LSET EMP2.NAM$ = EMP1.NAM$ : LSET ADDR2$ = ADDR1$
265 'Insert record
270 CALL BTRV (OP3%, STATUS%, FILE.PTR2%, BUF.LEN2%, KEY.VAL2$, KEY.NUM2%)
280 IF (STATUS% <> 0) THEN
    PRINT "Error writing to file. Status = ", STATUS% : END
285 'Step Next to retrieve next record
290 CALL BTRV (OP4%, STATUS%, FILE.PTR1%, BUF.LEN1%, KEY.VAL1$, KEY.NUM1%)
300 WEND
```

BASIC STEP PREVIOUS

In the following example, the indexes for a file have been damaged by a power failure. The application uses the Step Last and Step Previous operations to recover the file.

```
 30  OPEN "NUL" AS #1 LEN=86
 40  FIELD #1, 6 AS EMP1.NUM$,
     30 AS EMP1.NAM$,
     50 AS ADDR1$
 50  OP1% = 0                                           'Set open operation code
 60  KEY.NUM1% = -2                                     'Open in recovery mode
 60  FILE.PTR1%=VARPTR(#1) : BUF.LEN1% = 86    'Point to File#1, set data buf. length
 70  KEY.VAL1$ = "C:\DATA\EMPLOYE.BTR "
 80  CALL BTRV (OP1%, STATUS%, FILE.PTR1%, BUF.LEN1%, KEY.VAL1$, KEY.NUM1%)
 90  IF STATUS% <> 0 THEN
     PRINT "Btrieve status = ", STATUS%: END
100  KEY.VAL1$ = SPACE$(6) : KEY.NUM1% = 0
110  OPEN "NUL" AS #2 LEN=86
120  FIELD #2, 6 AS EMP2.NUM$,
     30 AS EMP2.NAM$,
     50 AS ADDR2$
125  BUF.LEN2% = 86                                     'Set data buffer length
140  KEY.NUM2% = 0 : FILE.PTR2% = VARPTR(#2)   'Set open mode and point to File #2
150  KEY.VAL2$ = "C:\DATA\EMPLOYE2.BTR "
170  CALL BTRV (OP1%, STATUS%, FILE.PTR2%, BUF.LEN2%,  KEY.VAL2$, KEY.NUM2%)
180  IF STATUS% <> 0 THEN
     PRINT "Btrieve status = ",STATUS%: END
190  KEY.VAL2$ = SPACE$(6)                              'Initialize key buffer
200  OP2% = 34 : OP3% = 2 : OP4% = 35                   'Set op codes
220  CALL BTRV (OP2%, STATUS%, FILE.PTR1%, BUF.LEN1%,  KEY.VAL1$, KEY.NUM1%)
230  WHILE STATUS% <> 9                          'Read until end of damaged file
240  LSET EMP2.NUM$ = EMP1.NUM$                  'Format new record with old data
250  LSET EMP2.NAM$ = EMP1.NAM$ : LSET ADDR2$ = ADDR1$
265  'Insert record
270  CALL BTRV (OP3%, STATUS%, FILE.PTR2%, BUF.LEN2%, KEY.VAL2$,
                          KEY.NUM2%)
280  IF (STATUS% <> 0) THEN
     PRINT "Error writing to file. Status = ", STATUS% : END
285  'Step Previous to retrieve previous record
290  CALL BTRV (OP4%, STATUS%, FILE.PTR1%, BUF.LEN1%, KEY.VAL1$, KEY.NUM1%)
300  WEND
```

BASIC STOP

The following example shows how to remove Btrieve from memory.

```
110 OP% = 25
170 CALL BTRV (OP%, STATUS%, FILE.PTR%, BUF.LEN%, KEY.VAL$,
                        KEY.SELECT%)
175 IF STATUS% <> 0 THEN
    PRINT "Btrieve status = ", STATUS% :END
```

BASIC UNLOCK

The following example shows how to unlock a record.

```
100 OP% = 27 : KEY.NUM% = 0
140 CALL BTRV (OP%, STATUS%, FILE.PTR%, BUF.LEN%, KEY.VAL%,
                        KEY.NUM%)
150 IF STATUS% <> 0 THEN
    PRINT "Btrieve status = ", STATUS% : END
```

BASIC UPDATE

The example below shows how to update a personnel file for an employee who just received a raise.

```
110 OP% = 5
130 FILE.PTR% = VARPTR(#2)
140 KEY.SELECT% = 0
145 KEY.VAL$ = "Jones, Mary E."
150 FIELD #2, 20 AS NAM$,
    6 AS DATE.OF.HIRE$,
    6 AS ANNUAL.SAL$
160 BUF.LEN% = 32
170 CALL BTRV (OP%, STATUS%, FILE.PTR%, BUF.LEN%, KEY.VAL$,
                        KEY.SELECT%)
175 IF STATUS% <> 0 THEN
    PRINT "Btrieve status = ", STATUS% : END
180 LSET ANNUAL.SAL$ = "025000"
190 OP% = 3
350 CALL BTRV (OP%, STATUS%, FILE.PTR%, BUF.LEN%, KEY.VAL$,
                        KEY.SELECT%)
360 IF STATUS% <> 0 THEN
    PRINT "Btrieve status = ", STATUS% : END
```

After the Update operation, Btrieve's current positioning in the file is as follows:

Adams, David H.	150781	030000	
Brown, William J.	010581	055000	← Previous record
Jones, Mary E.	120882	025000	← Current record
Smith, Bruce L.	100182	040000	← Next record

↑ Access Path

BASIC VERSION

The following example shows how an application might use the Version operation.

```
 90  FILE.PTR% = VARPTR (#2)
100  FIELD #2, 2 AS VER$,
     2 AS REV$,
     1 AS NET$
110  OP% = 26 :  BUF.LEN% = 5
170  CALL BTRV (OP%, STATUS%, FILE.PTR%, BUF.LEN%, KEY.VAL$, KEY.NUM%)
180  IF STATUS% <> 0 THEN
     PRINT "Btrieve status =", STATUS%
190  END
```

APPENDIX G: EXTENDED KEY TYPES

This appendix describes the extended key type codes and the internal storage formats for the extended key types supported by Btrieve.

EXTENDED KEY TYPE CODES

Specify the extended key type using the codes listed in the table in Figure G.1.

Type	Code
string	0
integer	1
float	2
date	3
time	4
decimal	5
money	6
logical	7
numeric	8
bfloat	9
lstring	10
zstring	11
unsigned binary	14
autoincrement	15

Figure G.1
Extended Key Type Codes

EXTENDED KEY TYPES

AUTOINCREMENT

An autoincrement key is an integer that can be either 2 or 4 bytes long. Btrieve sorts autoincrement keys by their absolute value, comparing the values stored in different records a word at a time from right to left.

Autoincrement keys allow you to specify a key for which Btrieve will increment the key value each time you insert a record into the file.

The following restrictions apply to autoincrement keys:

- An autoincrement key cannot allow duplicate key values.
- An autoincrement key cannot be segmented, or be included as a segment of another key.
- An autoincrement key cannot overlap another key.

The following paragraphs describe how Btrieve treats autoincrement key values when you insert records into a file.

If you specify a value of binary 0 for the autoincrement key, Btrieve will assign a value to the key based on the following criteria:

- If the record you are inserting is the first record to be inserted into the file, Btrieve will assign the autoincrement key a value of 1 and will insert the record into the file.
- If records already exist in the file, Btrieve will assign the key a value that is one number higher than the highest existing absolute value in the file, and will insert the record into the file.

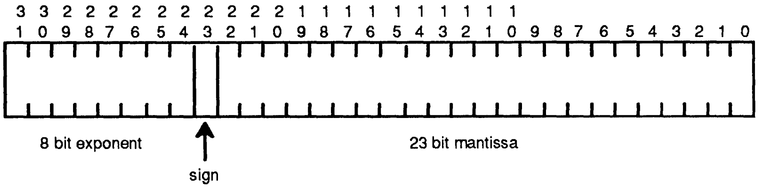
If you specify a nonzero value for the autoincrement key, Btrieve will insert the record into the file, and use the specified value as the key value. If a record containing that value already exists in the file, Btrieve will return a nonzero status and will not insert the record.

When you delete a record containing an autoincrement key, Btrieve will completely remove the record from the file. Btrieve will not re-use the deleted key value unless you specify that value when you insert another record into the file.

As mentioned previously, Btrieve always sorts autoincrement keys by their absolute value. If you specify a negative value for an autoincrement key when you insert a record, or if you update a record and negate the value for the autoincrement key, Btrieve will sort the value according to its absolute value. This allows you to use negation to flag records without altering the record's position in the index. In addition, when you perform a Get operation and specify a negative value in the key buffer, Btrieve will treat the negative value as the absolute value of the key.

BFLOAT

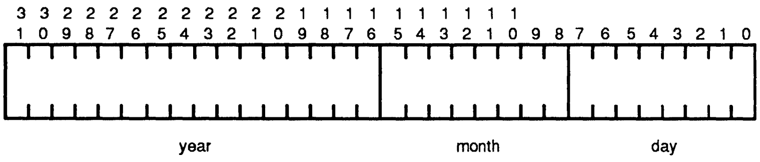
A field with a bfloat type is a single or double precision real number stored in a format that is compatible with Microsoft BASIC. A single precision real number is stored with a 23-bit mantissa, an 8-bit exponent biased by 128, and a sign bit. The internal layout for a 4-byte float is as follows:



The representation of a double precision real number is the same as that for a single precision real number, except that the mantissa is 55 bits instead of 23 bits. The least significant 32 bits are stored in bytes 0 through 3.

DATE

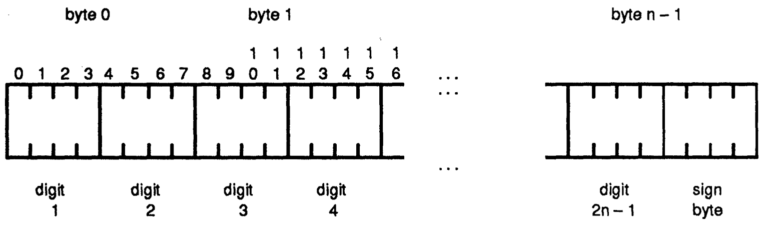
Date-type fields are 4-byte values stored internally as follows:



Day and month are each stored in 1-byte binary format. Year is a 2-byte binary number that represents the entire year value, not an offset from some defined year.

DECIMAL

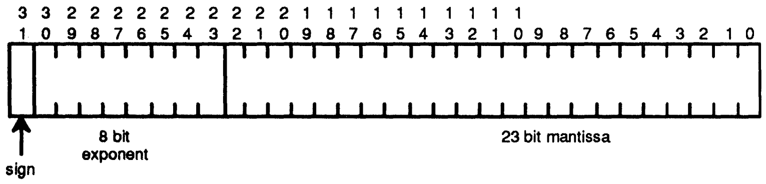
Decimal-type fields are stored internally as packed decimal numbers, with two decimal digits per byte. This format is consistent with the COMP-3 data type in ANSI-74 standard COBOL. The internal representation for an n-byte decimal field is as follows:



The sign byte is either F or C for positive numbers and D for negative numbers. Notice that the decimal point is implied. All of the values for a decimal key type must have the same number of decimal places in order for Btrieve to collate the key correctly.

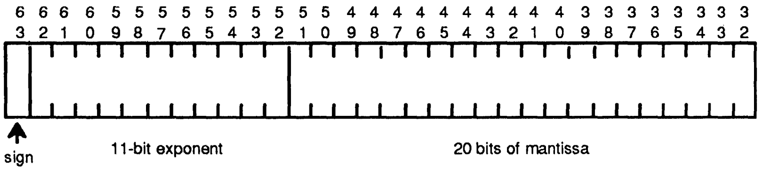
FLOAT

A float type is consistent with the IEEE standard for single and double precision real numbers. The internal format for a 4-byte float consists of a 23-bit mantissa, an 8-bit exponent biased by 127, and a sign bit. An illustration is shown below:

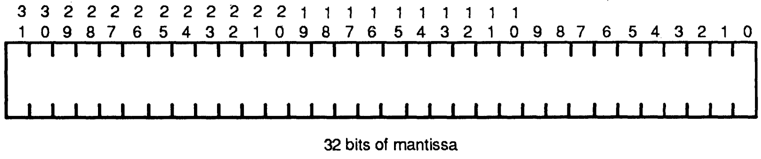


A float-type field with 8 bytes has a 52-bit mantissa, an 11-bit exponent biased by 1023, and a sign bit. The internal format is:

bytes 7-4:



bytes 3-0:



INTEGER

An integer type is a signed whole number and must contain an even number of bytes. Internally, integers are stored in Intel binary integer format, with the high-order and low-order bytes reversed within a word. Btrieve evaluates the key from right to left, a word at a time. The sign must be stored in the first nibble of the low order byte. This is compatible with the integer storage format of most languages.

LOGICAL

The logical extended key type is stored as a 1- or 2-byte value. Btrieve collates logical key types as a string. This allows the application to determine the stored values which represent true or false.

LSTRING

An lstring type in Btrieve corresponds to a Pascal string. It has the same characteristics as a regular string type except that the first byte of the string contains the binary representation of the string's length. The length stored in byte 0 of the lstring determines the number of significant bytes. Btrieve ignores any values beyond the specified length of the string.

MONEY

The internal representation for money types is exactly the same as that for decimal.

NUMERIC

Numeric values are stored as ASCII strings, right justified with leading zeros. Each digit occupies 1 byte internally. The right-most byte of the number includes an embedded sign. The following table indicates how the right-most digit is represented when it contains an embedded sign for positive and negative numbers.

Number	Positive	Negative
2	B	K
3	C	L
4	D	M
5	E	N
6	F	O
7	G	P
8	H	Q
9	I	R
0	{	}

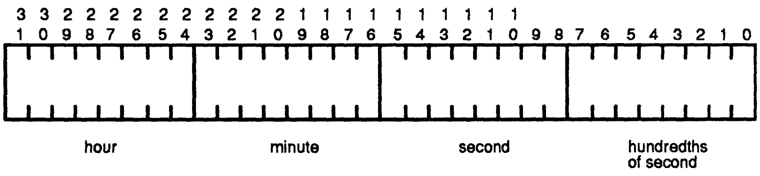
For positive numbers, the rightmost digit can be represented by 1 – 0 instead of A – {. Btrieve processes positive numbers represented either way equally.

STRING

A string type in Btrieve is a sequence of characters ordered from left to right. Each character is represented in ASCII format in a single byte.

TIME

Time-type fields are 4-byte values stored internally as follows:



UNSIGNED BINARY

Btrieve sorts unsigned binary keys as unsigned integers. An unsigned binary key must contain an even number of bytes. Btrieve compares unsigned binary keys a word at a time from right to left.

ZSTRING

A zstring type in Btrieve corresponds to a C string. It has the same characteristics as a regular string type except that a zstring type is terminated by a byte containing a binary 0. Btrieve ignores any values beyond the first binary 0 it encounters in the zstring.

GLOSSARY

access path: An index based on key fields that Btrieve uses to retrieve records. The key number determines the current access path. A file may have up to 24 separate access paths.

alternate collating sequence: A sorting sequence other than the standard ASCII sequence that specifies the order in which Btrieve will sort a key.

application interface: A program that allows access to Btrieve file structures from an application program.

ASCII: An acronym for American Standard Code for Information Interchange. ASCII is a standard 7-bit information code that defines 128 standard characters, including control characters, letters, numbers, and symbols.

cache: The area of main memory where physical disk pages are buffered to reduce physical disk requests.

concurrency controls: The methods provided by Btrieve to resolve possible conflicts when two stations attempt to update or delete the same records at the same time. Concurrency controls include passive control, transaction control, and record locking.

database: A set of one or more consecutive records or files on a related subject.

description files: Sequential files containing information necessary for Btrieve's CREATE, INDEX, and SINDEX utilities.

directory: A disk structure that contains files. A directory may also contain other subdirectories.

DOS: Either the MS DOS or the PC DOS operating system.

duplicate key: An attribute associated with a key field that allows a single key value to identify a subset of records within the file that contains that value.

FCR: *See* file control record.

field: A storage area within a record for a group of characters that constitute an item of information.

file: A collection of related records treated as a unit.

file control record: The first page in a file that contains the file's size and other characteristics.

filename: A combination of the device, directory path, name, and extension which uniquely identify a file.

index: An ordered set of keys associated with records and locations of the records in a file. A file can have more than one index.

integrity control: The method used to ensure the completeness of files. Specifically, Btrieve uses pre-imaging to guarantee file and database level integrity.

key: A record identifier kept in the file index identifying the position of the record. The keys in an index are ordered according to a definite collating sequence. A key takes its value from the area in a record corresponding to its predefined offset and length.

key field: A field Btrieve uses to identify specific records.

key number: An identifier associated with a specific key field in a file.

memory resident: An attribute of a computer program that allows the image of the program to remain in memory, even after execution is terminated. The operating system does not load other programs in the same memory area occupied by a memory resident program. Under DOS, the Btrieve Record Manager is a memory resident program that resides in memory until the system is restarted or a Stop operation is performed.

modifiable key: An attribute associated with a key field that allows the value of the field to change during updates. Otherwise, key field values are not allowed to change.

null key: A key field that allows the value of the field to be a user-defined null character. For null keys, Btrieve does not index a record if the record's key value matches the null value.

operation: A specific action (Delete, Create, Get Equal, etc.) that manipulates a record in a Btrieve file.

page: A unit of disk storage containing a multiple of 512 bytes, no greater than 4096 bytes. A page is the smallest unit of storage that Btrieve moves between main memory and disk.

partitioned file: A logical Btrieve file composed of two separate physical disk files. This allows Btrieve logical files to be larger than the physical disk.

positioning: The establishment of the record manager's current location within a current access path following the successful completion of an operation.

pre-imaging: The process of storing the image of a file page before a record on the page is updated. Btrieve uses pre–imaging to provide recovery capabilities in case a file is damaged, or in the event of a system failure during an operation.

primary file: The original part of a partitioned file before an EXTEND operation was performed.

record: A set of one or more consecutive fields about a related subject, such as an employee's payroll record.

Record Manager: The part of Btrieve that performs the transfer of logical records between an application program and their physical disk location.

segmented key: A key field that allows the key to have non-contiguous parts. That is, non-contiguous sets of characters in a record can constitute a single key.

transaction: A set of logically related Btrieve operations on up to 12 different files.

utilities: Standard routines that perform "housekeeping" functions, such as BUTIL –COPY, –STOP, etc.

TRADEMARKS

INDEX

Novell Manuals and Novell Products USER COMMENTS

We at Novell would like to hear from you if you have comments about our manuals and products, or have suggestions for improving them. Please address responses to:

Novell Development Products Division
User Comments
6034 West Courtyard Dr., Suite 220
Austin, Texas 78730

Please indicate the relevant chapters and page numbers and other pertinent information as requested below.

Product Name: ______________________ Version Number: ________

Manual Name and Revision (if applicable): ______________________

Your Name: ______________________

Company Name: ______________________

Address: ______________________

City: ______________ State: __________ Zip: _________

Phone Number: (____) ______________________

COMMENTS OR SUGGESTIONS

____

____

____

____

____

____

____

____

____

183-000095-001

NOVELL, INC.

122 EAST
1700 SOUTH

PROVO, UTAH
84606

(801) 379-5900
(800) 453-1267

MADE IN U.S.A.

101-000530-001

NOVELL®