

TopSpeed Modula-2



TopSpeed® Modula-2

For IBM® Personal Computers and Compatibles

B-tree Toolkit

Jensen & Partners International

LICENSE STATEMENT

Jensen & Partners International hereby grants you a non-exclusive license to use the TopSpeed B-tree Toolkit. You may use the software on any computer, **provided that the software cannot possibly be used on more than one computer at the same time.** You may make copies of the software for the sole purpose of loading it into the computer on which you wish to use it, or for keeping a maximum of two backup copies.

You are furthermore granted the right to sell programs developed by you using the program modules provided in this package. You are not allowed to re-sell or give away any part of the source code provided with this product.

WARRANTY

Jensen & Partners International warrants the enclosed diskette to be an accurate copy of the master disk, and will replace any defective copy free of charge for a period of 60 days after purchase.

Jensen & Partners International hereby explicitly disclaims all other warranties whether express or implied, including without limitation, the implied warranties of merchantability and fitness for any particular purpose. Jensen & Partners International shall have no liability for consequential, incidental, exemplary or special damages, including lost profits.

Jensen & Partners International reserves the right to change the product at any time, without prior notice.

TopSpeed[®] is a registered trademark of Jensen & Partners International.

IBM[®] is a registered trademark of the International Business Machines Corporation.

Microsoft[®] is a registered trademark of Microsoft Corporation.

Copyright © 1989 by Jensen & Partners International. All rights reserved.

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

Chapter 1

Contents

Introduction

1	Introduction	1
2	Architecture	3
3	A Simple Address Book	5
	Data Structures	10
	The Btree Interface	10
	Supporting Functions	13
4	The Club — An Advanced Application	17
	Background	17
	Data Structures and Files	17
	The Club Program	19
	Clients, Items, and Sales	19
	GLOBALS Support	22
5	Using Btree in Multi-User Environments	25
	The FIOx Module	26
	The Btree Module	30
	An Example	31
6	Btree and B-Tree Indexing	35
	Balanced Trees	37
	Page Size	38
	Btree Implementation	39
7	B-Tree Toolkit Module Descriptions	41
	Btree Module	41
	FIOx Module	60
	Misc Module	63
	Index	65

Chapter 1

Introduction

Welcome to the world of database manipulation with the B-tree Toolkit from Jensen & Partners International. This toolkit contains all the tools necessary to build fast and efficient database programs with TopSpeed Modula-2.

The B-tree Toolkit is a set of routines and structures which perform B-tree indexing, data and index file manipulation, and shared file handling for local area network, multi-user and multi-tasking systems.

The B-tree Toolkit definition modules are identical for both MS-DOS and OS/2, so that applications may be written for either environment without changes to your source code. The Toolkit contains all source code for the file handling and shared file access modules, along with several complete example programs.

This manual is organized as follows:

- | | |
|---------------------------|--|
| Chapter 2 | briefly describes the architecture and philosophy of the B-tree Toolkit. |
| Chapters 3 & 4 | present case studies using Btree . Chapter 3 is a simple address book, while chapter 4 presents a small point-of-sale system. (While reading these chapters, you may find it helpful to refer to Chapter 7 — for more information about procedures in the Btree module.) |

Chapter 5

outlines the changes needed to use Btree-based programs in multi-tasking and multi-user environments.

Chapter 6

gives a short tutorial on how **Btree** works, and the principles of B-tree indexing.

Chapter 7

presents the complete definition modules for the modules that form the B-tree Toolkit, namely modules **Btree.DEF** and **FIOx.DEF**.

Chapter 2

Architecture

The B-tree Toolkit provides routines for maintaining multiple data and index files. It differs from conventional database systems, in that it does not attempt to provide *data abstraction*. Rather, the data and index items are stored in the same binary format as they exist in while in program memory.

The application must open one or more physical files. These are the actual disk files which contain data and indexes. It then opens logical data and index files, specifying which physical files the data and index files are actually stored in. This allows related files to be stored in the same physical file on the user's disk drive. It also avoids the infamous "15 handle limit" imposed by MS-DOS — the number of physical files concurrently open is still limited by the 15 handle limit, but any number of data and index files (program memory permitting) may be open within those physical files. Physical files are limited to 4 gigabytes in size (disk drive space permitting).

When a data file is opened, it may be specified as containing either fixed- or variable-length records. Each type of record may be up to 65535 bytes in length. If the records are to be of fixed-length, then that length must be specified when the data file is opened. If the records are to be of variable length, then the records lengths are specified when records are added to the data file.

When an index file is opened, it may be associated with a data file. If an index file is opened in this manner, then adding and deleting records to/from the data file causes the index to be updated automatically. Index keys may be up to 128 bytes in length.

(This size is determined by a constant in the **BTREE.DEF** file — it may be modified upward, if attention is given to possible stack overflows, as well as the page size used for the B-tree nodes). Index comparisons are performed by procedures which the application must supply when the index is opened. If the index is associated with a data file, then the application must also supply a procedure to be used for deriving keys from data items. Since key derivation and comparison is accomplished by application-supplied procedures, there are almost no limitations on the sorting algorithm used by indexes. Indexes may also be specified as allowing or disallowing duplicate keys.

In addition to the high-level data file and index file manipulation routines, there are low-level routines for allocation/deallocation and reading/writing of blocks within physical files. Up to 4 gigabytes of space may be allocated within the physical file, though the reading/writing routines only allow 65535 bytes to be transferred in a single function call.

Allocations (such as adding data and index items) within the physical file first attempt to use any free space which may be available, and then possibly cause the physical file to grow. Deallocations within the physical file cause the space to be added to a *free list*, but never cause the physical file to contract. Because of the binary nature of the data within the data and index records, no automatic file compaction can be provided by the **Btree** module. This is because data and index records may refer to absolute offsets (such as those generated by allocating records larger than 65535 bytes) within the physical record. As a result, applications must compact physical files by copying the active data and index items to a new physical file.

Error handling is provided in two ways.

1. Each physical and logical file has an associated *last error* status, which may be queried by calling **LastError()** with the logical file handle, or by calling **LastFError()** with the physical file handle. (If no error has occurred, then the last error will be the value **OK** (which is one of the values of the **Errors** enumerated type in the **Btree** module).)
2. Certain “serious errors” are also reported by calling a procedure variable (exported by **Btree**), with the error type and an error message. The default behavior (as initialized by **Btree**) is for the error message to be printed, and the program terminated. Applications may replace the default procedure with something more amiable to recovery.

Information on the networking support provided by the B-tree Toolkit is contained in Chapter 5. In addition, specific information about the various **Btree** routines is contained in Chapter 7.

Chapter 3

A Simple Address Book

To better explain how the B-tree Toolkit is used, this chapter presents a simple application — an address book. This program allows the user to enter, delete, and list the contents of an electronic address book, using **Btree** to manage all necessary files and indexes.

(NOTE: This example may be found on the B-tree Toolkit disk, in the file **ADDRESS.MOD**.)

```
MODULE Address;
(*
  Copyright 1989 by Jensen & Partners International, All Rights Reserved
*)
IMPORT IO, FIO, Window, Btree, Str, Lib, Misc;

TYPE
  AddrKey      = ARRAY[0..25] OF CHAR;
  AddrRec      = RECORD
    LName      : ARRAY[0..14] OF CHAR;
    FName      : ARRAY[0..9] OF CHAR;
    Street      : ARRAY[0..25] OF CHAR;
    City        : ARRAY[0..15] OF CHAR;
    State       : ARRAY[0..1] OF CHAR;
    Zip         : ARRAY[0..4] OF CHAR;
    Phone       : ARRAY[0..12] OF CHAR;
  END; (* record *)
  CharSet      = SET OF CHAR;

VAR
  choice, ch2 : CHAR;
```



```

FOUND,
DONE      : BOOLEAN;
DataColor,
TemplateColor : Window.Color;
NameFile   : Btree.FHandle;  (* physical file handle *)
NameData,  (* logical data file handle *)
NameIndex  : Btree.IHandle;  (* logical index file handle *)
blankAddr,
tempAddr   : AddrRec;
Create     : BOOLEAN;

PROCEDURE MAKEKEY(keyA, myRecA: ADDRESS);
VAR
  key : POINTER TO AddrKey;
  myRec : POINTER TO AddrRec;
  i, j : CARDINAL;
BEGIN
  key := keyA;
  myRec := myRecA;
  Str.Copy(key^, myRec^.LName);
  i := Str.Length(key^);
  WHILE (i#0) AND (key^[i-1]=' ') DO
    key^[i-1] := 0C;
    DEC(i);
  END; (* while *)
  Str.Append(key^, " ");
  Str.Append(key^, myRec^.FName);
  Str.Caps(key^);
END MAKEKEY;

PROCEDURE COMPARE(key1, key2: ADDRESS): Btree.CmpRes;
(* compare two keys, and determine which is "smaller" *)
BEGIN
  RETURN Btree.CmpRes(Str.Compare(AddrKey(key1^), AddrKey(key2^))+1);
END COMPARE;

PROCEDURE Beep();
BEGIN
  (* generic *)      FIO.WrChar(FIO.StandardOutput, CHR(7));
  (* OS/2 *)          (* Lib.Speaker(1500,10); *)
  (* DOS *)           (* Lib.Sound(1500);
                      Lib.Delay(10);
                      Lib.NoSound(); *)
END Beep;

PROCEDURE WriteAt(row, col: CARDINAL; txt: ARRAY OF CHAR);
BEGIN
  Window.GotoXY(col, row);
  IO.WrStr(txt);
END WriteAt;

PROCEDURE Template();
BEGIN
  Window.TextColor(TemplateColor);
  WriteAt(3, 3, "Name:");
  WriteAt(3, 50, "(first last)");
  WriteAt(4, 3, "Address:");
  WriteAt(5, 3, "City:");
  WriteAt(5, 40, "State:");
  WriteAt(5, 55, "Zip:");

```



```

        WriteAt(6,3,"Phone:");
    END Template;

PROCEDURE Display(myRec: AddrRec);
BEGIN
    Window.TextColor(DataColor);
    WriteAt(3,12,myRec.FName); IO.WrStr(" "); IO.WrStr(myRec.LName);
    WriteAt(4,12,myRec.Street);
    WriteAt(5,12,myRec.City);
    WriteAt(5,47,myRec.State);
    WriteAt(5,60,myRec.Zip);
    WriteAt(6,12,myRec.Phone);
END Display;

PROCEDURE GetMenu(Prompt: ARRAY OF CHAR; OKChars: CharSet;
    VAR ch: CHAR);
VAR
    OK      : BOOLEAN;
    MenuWin : Window.WinType;
    i       : CARDINAL;
BEGIN
    MenuWin := Window.Open( Window.WinDef( 0, 22, 79, 24,
        Window.White,Window.Black,TRUE,FALSE,TRUE,TRUE,
        Window.SingleFrame,Window.White,Window.Black) );
    i := Str.Length(Prompt);
    Window.Change(MenuWin, ((80-i) DIV 2) -2,22, ((80-i) DIV 2) + i + 3, 24);
    Window.SetFrame(MenuWin,Window.SingleFrame,TemplateColor,Window.Black);
    Window.TextColor(DataColor);
    Window.Use(MenuWin);
    Window.PutOnTop(MenuWin);
    IO.WrStr(" ");
    IO.WrStr(Prompt);
    OK := FALSE;
    REPEAT
        ch := CAP(IO.RdKey());
        IF ch IN OKChars THEN
            OK := TRUE;
        ELSE
            Beep();
        END;
    UNTIL OK;
    Window.Close(MenuWin);
END GetMenu;

PROCEDURE ShowNames();
VAR
    myRec : AddrRec;
    ch     : CHAR;
    row    : CARDINAL;
    switchColor : BOOLEAN;
BEGIN
    Window.Clear();
    Btree.Reset(NameData);
    row := 1;
    switchColor := TRUE;
    WHILE Btree.Next(NameIndex,myRec) DO
        IF switchColor THEN
            Window.TextColor(TemplateColor);
        ELSE
            Window.TextColor(DataColor);
        END;
        switchColor := NOT switchColor;
    END;

```



```

IO.WrStrAdj(myRec.LName, -17);
IO.WrStrAdj(myRec.FName, -12);
IO.WrStrAdj(myRec.Phone, -15);
IO.WrLn();
INC(row);
IF row>23 THEN
  IO.WrLn();
  IO.WrStr("Press <ESC> to quit, any other key to continue: ");
  ch := IO.RdKey();
  IF ch = 33C THEN RETURN; END;
  Window.Clear();
  row := 1;
END; (* if *)
END; (* while *)
END ShowNames;

PROCEDURE FindName();
VAR
  fname : AddrKey;
  myRec : AddrRec;
  ch : CHAR;
BEGIN
  Window.Clear();
  Window.TextColor(TemplateColor);
  FOUND := FALSE;
  WriteAt(3,3,"Enter name as 'Last First' (e.g. 'Smith John').");
  WriteAt(4,3,"Be sure to include the space between the names.");
  WriteAt(5,3,"> ");
  Window.TextColor(DataColor);
  Misc.InputStr(fname,25);
  Str.Caps(fname);
  Window.TextColor(TemplateColor);
  WriteAt(7,3,"Searching... ");
  IF Btree.Find(NameIndex,fname,myRec) THEN
    Window.Clear();
    Template();
    Display(myRec);
    FOUND := TRUE;
  ELSE
    WriteAt(7,20,"done. Failed exact search. Trying closest match.");
    Window.TextColor(DataColor);
    WriteAt(9,3,"Searching... ");
    IF Btree.Search(NameIndex,fname,myRec) THEN
      Window.Clear();
      Template();
      Display(myRec);
      FOUND := TRUE;
    REPEAT
      GetMenu("N)ext, P)revious, or T)his one? ",
        CharSet{'N','P','T'},ch);
      WriteAt(10,3,"");
      CASE ch OF
        'N' : IF Btree.Next(NameIndex,myRec) THEN
          Display(blankAddr);
          Display(myRec);
        ELSE
          WriteAt(10,3,"No more records found");
          Beep();
        END;
        | 'P' : IF Btree.Prev(NameIndex,myRec) THEN
          Display(blankAddr);
          Display(myRec);
        ELSE
          WriteAt(10,3,"No more records found");

```



```

        Beep();
    END;
    ELSE
        Beep();
    END; (* case *)
    UNTIL ch = 'T';
    ELSE
        WriteAt(9,20,"done. Failed closest match search.");
    END; (* if *)
    END; (* if *)
END FindName;

PROCEDURE AddName();
VAR
    myRec : AddrRec;
    ch : CHAR;
BEGIN
    Window.Clear();
    Template();
    ch := 'N';
    myRec := blankAddr;
    REPEAT
        Window.TextColor(DataColor);
        Window.GotoXY(12,3);      Misc.InputStr(myRec.FName,10);
        Window.GotoXY(35,3);      Misc.InputStr(myRec.LName,15);
        Window.GotoXY(12,4);      Misc.InputStr(myRec.Street,25);
        Window.GotoXY(12,5);      Misc.InputStr(myRec.City,15);
        Window.GotoXY(47,5);      Misc.InputStr(myRec.State,2);
        Window.GotoXY(60,5);      Misc.InputStr(myRec.Zip,5);
        Window.GotoXY(12,6);      Misc.InputStr(myRec.Phone,12);
        GetMenu("Do these look ok? (Y/N) ",CharSet{'Y','N'},ch);
    UNTIL ch = 'Y';
    (* Now put it in the file *)
    Btree.Add(NameData,myRec,SIZE(myRec));
END AddName;

PROCEDURE DeleteName();
VAR
    ch : CHAR;
BEGIN
    FindName();
    IF FOUND THEN
        GetMenu("Delete this one? (Y/N) ",CharSet{'Y','N'},ch);
        IF ch = 'Y' THEN
            Btree.Delete(NameData);
        END; (* if *)
    END; (* if *)
    FOUND := FALSE;
END DeleteName;

BEGIN (* main program *)
    TemplateColor := Window.LightGreen;
    DataColor := Window.Yellow;
    Str.Copy(blankAddr.LName,"");
    Str.Copy(blankAddr.FName,"");
    Str.Copy(blankAddr.Street,"");
    Str.Copy(blankAddr.City,"");
    Str.Copy(blankAddr.State,"");
    Str.Copy(blankAddr.Zip,"");
    Str.Copy(blankAddr.Phone,"");
    Window.Clear();
    Create := NOT FIO.Exists("Address.DAT");

```



```

NameFile := Btree.Open("Address.DAT",2,Btree.FixSize,
                      FALSE,FALSE,Create);
NameData := Btree.OpenData(NameFile,1,SIZE(AddrRec),Create);
NameIndex := Btree.OpenIndex(NameFile,NameData,2,COMPARE,MAKEKEY,
                             SIZE(AddrKey),TRUE,Create);

DONE := FALSE;
choice := ' ';
REPEAT
  GetMenu("S)howNames F)indSomeone A)ddSomeone D)elteSomeone Q)uit :",
    CharSet{'S','F','A','D','Q'},choice);
  CASE choice OF
    'S' : ShowNames();
    'F' : FindName();
    'A' : AddName();
    'D' : DeleteName();
    'Q' : DONE := TRUE;
  ELSE
    Beep();
  END; (* case *)
UNTIL DONE;
Btree.Close(NameFile);
END Address.

```

Data Structures

In order to build an address book, we must first define the information which will be kept on each person in the book. For our application, we will use the following record definition:

```

TYPE
  AddrKey = ARRAY[0..25] OF CHAR;
  AddrRec = RECORD
    LName      : ARRAY[0..14] OF CHAR;
    FName      : ARRAY[0..9] OF CHAR;
    Street     : ARRAY[0..25] OF CHAR;
    City       : ARRAY[0..15] OF CHAR;
    State      : ARRAY[0..1] OF CHAR;
    Zip        : ARRAY[0..4] OF CHAR;
    Phone      : ARRAY[0..12] OF CHAR;
  END; (* record *)

```

The Btree Interface

For **Btree** to correctly manage these data, two routines must be written. The first routine creates a key from the data record, and the second routine compares two keys and returns the result.

In this case, we wish to index the file based on the last and first names of the person in question. Therefore, our key-making routine looks like this:

```

PROCEDURE MAKEKEY(keyA,myRecA: ADDRESS);
VAR
  key : POINTER TO AddrKey;
  myRec : POINTER TO AddrRec;

```



```

1,j : CARDINAL;

BEGIN
  key := keyA;
  myRec := myRecA;
  Str.Copy(key^,myRec^.LName);
  i := Str.Length(key^);
  WHILE (i#0) AND (key^[i-1]=' ') DO
    key^[i-1] := 0C;
    DEC(i);
  END; (* while *)
  Str.Append(key^," ");
  Str.Append(key^,myRec^.FName);
  Str.Caps(key^);
END MAKEKEY;

```

The **MAKEKEY** routine starts by copying the last name into the **key** field, using the **Copy** routine from the **Str** module. Then, the trailing blanks are deleted. Finally, a space and the first name are appended, and the key is capitalized. This creates our key, upon which the file will be indexed.

To compare the records, a **COMPARE** routine is created as follows:

```

PROCEDURE COMPARE(key1A,key2A: ADDRESS): Btree.CmpRes;
(* compare two keys, and determine which is "smaller" *)
BEGIN
  RETURN Btree.CmpRes(Str.Compare(AddrKey(key1A^),AddrKey(key2A^))+1);
END COMPARE;

```

Here, since we know that the keys are simply strings, we use the **Compare** procedure from the **Str** module to compare them. Based on the value returned by **Compare**, we return a value of **Eq**, **Greater**, or **Less** (all of which values are found in **Btree**). These values allow **Btree** to know which record key should be stored first in the index.

The necessary "physical" and "logical" files must now be created and/or opened by the application. The physical file is the actual **Btree** file which is stored on the disk. The logical files are files within the physical file, and may be either data or index files, or a combination of both. The calls are shown next, followed by an explanation.

```

Create := NOT FIO.Exists("Address.DAT");
NameFile := Btree.Open("Address.DAT",2,Btree.FixSize,FALSE,FALSE,
  Create);
NameData := Btree.OpenData(NameFile,1,SIZE(AddrRec),Create);
NameIndex := Btree.OpenIndex(NameFile,NameData,2,COMPARE,MAKEKEY,
  SIZE(AddrKey),TRUE,Create);

```

The first file that must be opened is the special **Btree** file, or "physical" file. This file will contain all the data and index information used by your program and by **Btree**. This file is the actual file on the disk, and is created through a call to **Btree.Open**. The **Open** call takes six parameters.

Filename

The name of the physical disk file.

NumberOfLogFiles	The number of logical files that will reside within this physical file.
RecordType	This is one of four values, which define the record type used. FixSize denotes a fixed sized record, Size16 denotes variable length records up to a length of 64K bytes. Size32 denotes variable sized records based only on the size of memory in the machine. NoDeal-loc denotes that no deallocation will ever take place in the file.
ReadOnlyFlag	This flag denotes whether this file is only to be used for reading (no output allowed). In most cases, this will be FALSE .
SharedFlag	This flag denotes whether the file should be available for sharing with other processes. This should be FALSE , unless you have another application that is specifically written to make use of this feature. (See Chapter 5 for more information on using Btree with networks and OS/2, using shared files.)
CreateFlag	If this parameter is TRUE , the file will be created if it does not exist. If the flag is FALSE , and the file does not exist, an error ensues.

We have created a physical file, and must now define the data and index files which will reside within it. First, we make a call to **OpenData** to define the data file itself. The call to **OpenData** takes four parameters.

Btree File Handle	The handle of the Btree physical file we are using, in this case NameFile .
LogFile	This parameter lets Btree know which of the logical files for the physical file will be used by this data file. In this example, we have created a physical file with two logical handles available — one for the data file and one for the index file. Here, we use "1" as the parameter to allocate the first handle to the data file.
RecSize	The third parameter passed is the size of the data record. If the size is given as "0," then Btree will assume that variable length records are being used.
Create	This last parameter notifies Btree whether this data file should be "created" within the given physical file. If the logical data file is not to be created, and if it has not already been created, an error is generated.

Once we have a data file defined, we can move on and set up the index file. This is performed through a call to **OpenIndex**, which is a bit more complicated than the other "Open" statements.

OpenIndex takes eight parameters.

Btree File Handle	This parameter is the handle of the Btree physical file being used.
Data File	This is the handle of the logical data file that we wish to index.
Log File	This is the number of the logical file handle we wish to use within this data file.
CompareFunc	This is the name of the function to be called by Btree for comparing two record keys. In this case, we point to a local COMPARE function, which returns the valid Btree responses.
KeyFunc	This is the name of the function which Btree can call to generate a key from a given data record.
KeySize	This parameter specifies the size of the keys in this index.
DupKeys Flag	The DupKeys flag lets Btree know whether duplicate keys should be allowed. In some cases, like when using a last name for a key, duplicates should be allowed (flag = TRUE), to allow multiple people to have the same last names. In other cases, for instance when tracking different pieces in inventory, each record should have a unique key, and this parameter would be FALSE .
Create Flag	The create flag signifies whether this index needs to be created for the first time, or whether it should already exist within the system.

All the files and indexes are now open. We may proceed with our application, in this case, a small address book.

Supporting Functions

The **Address** program makes use of several, non-**Btree** support modules to execute certain functions. These functions are noted briefly below.

GetMenu

```
TYPE CharSet = SET OF CHAR;
```

```
PROCEDURE GetMenu(Prompt: ARRAY OF CHAR; OKChars: CharSet;
  VAR ch: CHAR);
```

GetMenu is used to prompt the user for a choice. The **Prompt** variable holds the information which the user must see in order to make a decision. The **OKChars** set holds all the possible selections, so that incorrect selections will remain in the loop until the user learns how to read. The **ch** variable returns the value of the key which actually caused the routine to return.

For instance, in **Address**, the main loop of the program is controlled in this way:

```
REPEAT
  GetMenu("S)howNames F)indSomeone A)ddSomeone D)eteSomeone Q)uit :",
    CharSet{'S','F','A','D','Q'}, choice);
  CASE choice OF
    'S': ShowNames();
    'F': FindName();
    'A': AddName();
    'D': DeleteName();
    'Q': DONE := TRUE;
  ELSE
    Beep();
  END;
UNTIL DONE;
```

This allows the user to make a choice after each function. He can quickly recognize the available options and be able to make selections based on single keystroke entries.

Beep

```
PROCEDURE Beep();
```

This routine exists only to make a sound when an incorrect option is exercised.

WriteAt

```
PROCEDURE WriteAt(row,col: CARDINAL; text: ARRAY OF CHAR);
```

WriteAt uses the **Window.GotoXY** routine to move to the correct screen location (within the current window) and then writes the text message to the screen. This simplifies the placement of messages and prompts which need to appear at specified locations.

Address Specific Routines

```
PROCEDURE Template();  
  
PROCEDURE Display(myRec: AddrRec);  
  
PROCEDURE AddName();  
PROCEDURE FindName();  
PROCEDURE ShowNames();  
PROCEDURE DeleteName();
```

Template is used to show the prompts for each field in the **AddrRec**. Then, a call to **Display** will fill in the data fields with the information in the **myRec** record.

The last four routines (**AddName**, **ShowNames**, **DeleteName**, **FindName**) utilize the corresponding **Btree** routines to maintain our address book file.

AddName asks for the necessary information about the entry we will be saving, and then writes the information to the **Btree** file using a call to **Btree.Add**. The following parameters are used:

NameData	The name of the data file into which to add this information.
myRec	The actual data collected at this time.
SIZE (myRec)	The number of bytes to be saved.

FindName is used to find names, not only for display, but also for deletion, as the **Btree** deletion routine works on the most recently retrieved record. **FindName** asks you for the name to find, and then searches the database using first the **Btree.Find** call (for exact match), and then the **Btree.Search** call (for close matches). **Find** and **Search** each take three parameters — the index to use, the name to find, and a record into which the found data is to be placed.

Once a name has been found and displayed, **FindName** allows you to move forward and backward through the file, utilizing calls to the **Btree.Next** and **Btree.Prev** routines. These routines use two parameters — the name of the index to use and a record in which the found data can be stored.

DeleteName first makes a call to **FindName**, to locate the record to be deleted. Once the record is selected, a call to **Btree.Delete**, using the data file as the only parameter, deletes the record most recently found.

ShowNames uses a call to **Btree.Reset**, using the name of the data file which should have its indexes reset, to set the index to the first entry. Once this is done, each record is read, using repeated calls to **Btree.Next**. The record is then displayed on the screen, in one-line format.

Three major types of data need to be maintained in the Club scenario — information about clients, inventory, and sales. The client and inventory information will be

Data Structures and Files

(NOTE: This example may be found on the B-tree Toolkit disk, in the following files: CLUB.MOD, CLIENT.DEF, CLIENT.MOD, ITEMS.DEF, ITEMS.MOD, SALES.DEF, SALES.MOD, GLOBALS.DEF, and GLOBALS.MOD)

This example involves a point-of-sale system for a small club, where information is maintained on clients and inventory, and a sale can be made from the program, either charging the amount to the client's record, or accepting cash as payment.

Although the example presented in the previous chapter highlighted some of the features of Btree, a more extensive example will be given in this chapter as a more in-depth study.

Background

Chapter 4 The Club — An Advanced Application

maintained in **Btree** indexed files, for quick recall during a point-of-sale transaction, and the daily sales totals will be maintained in a single record binary file.

The client information record is described in **CLIENT.DEF**, and looks like this:

TYPE

```
ClientRec = RECORD
  CNum   : ARRAY[0..9] OF CHAR;
  LName  : ARRAY[0..14] OF CHAR;
  FName  : ARRAY[0..9] OF CHAR;
  Street : ARRAY[0..24] OF CHAR;
  City   : ARRAY[0..14] OF CHAR;
  State  : ARRAY[0..1] OF CHAR;
  Zip    : ARRAY[0..5] OF CHAR;
  Phone  : ARRAY[0..12] OF CHAR;
  Balance: LONGREAL;
END; (* record *)
```

A single index is kept on the client, based on the **CNum** (client number) field. This information will be kept in the **Btree** file named **CLIENT.DAT**.

The inventory information record is described in **ITEMS.DEF**, and looks like this:

TYPE

```
ItemRec = RECORD
  INum    : ARRAY[0..9] OF CHAR;
  Description : ARRAY[0..9] OF CHAR;
  PartNum  : ARRAY[0..9] OF CHAR;
  Mfgr     : ARRAY[0..9] OF CHAR;
  QOHand,
  QOOrder  : CARDINAL;
  Cost,
  Retail   : LONGREAL;
  Supplier : ARRAY[0..9] OF CHAR;
END; (* record *)
```

A single index is kept for the inventory database, based on the **INum** (item number) field. This information is kept in the **Btree** file named **ITEMS.DAT**.

The daily sales items are kept in a record defined in the **SALES.DEF** file, which looks like this:

TYPE

```
TotalRec = RECORD
  TotalDollars : LONGREAL;
  TotalCash,
  TotalCharge  : LONGREAL;
  NumTrans     : CARDINAL;
END; (* record *)
```

This information is kept in a file whose name changes each day. The format of the file name is "MMDDYY.TOT," where MM is the current month, DD is the day, and YY is the year. So, on August 1, 1989, the totals would be kept in a file named 080189.TOT. These files are available on an all day basis. Should you need to exit the program at any time during the day, the totals can be reread from the file.

The Club Program

The **Club** program has three main functions:

- Maintaining client information
- Maintaining inventory information, and
- Making sales

To perform these functions, three modules have been created. **CLIENT.DEF** and **CLIENT.MOD** implement the functions necessary to maintain client records. Modules **ITEMS.DEF** and **ITEMS.MOD** perform similar functions to maintain the inventory database. **SALES.DEF** and **SALES.MOD** use the client and inventory databases to make sales.

The main program is found in the **CLUB.MOD** file. The basic functions here are to display a main menu, accept a user choice, and call the necessary routines in the **ITEMS**, **CLIENT**, and **SALES** modules to execute the selected function.

The **CLUB.MOD** file also handles the creation, opening, and closing of **Btree** files. This enables you to see how simply **Btree** can be integrated into a program.

Two routines perform the main work. **OpenClientFiles** checks for the existence of **CLIENT.DAT**. If this file does not exist, it is created with **Btree** calls. **Open()** is called to create the file, **OpenData()** is used to open the logical data file, and **OpenIndex()** opens the logical index file.

OpenItemFiles then performs a similar function — opening the inventory files, creating them if they did not exist.

A simple **Help()** routine is also available, to notify the user what the options on the main menu mean.

As part of the main menu, a listing may be requested for **Client** and **Items** records. These listings are made using calls to the **Btree.Reset** and **Btree.Next** routines.

Clients, Items, and Sales

In order to make using and modifying the **Club** program simpler, a common interface was adopted across the **CLIENT**, **ITEMS**, and **SALES** definition and implementation files. The following routines are available in the modules, as described below.

Template

PROCEDURE Template();

Template() is used in each case to display, on the screen, the window that will be used for the current functions. For instance, the **Client.Template()** routine displays the window in which client data is entered and displayed.

Template() also stores the previous window, so that it can be restored when a call is made to **Close()** (see below).

The only difference in the three modules, other than the actual screen layout, is that the **Sales.Template()** routine also reads in the daily totals file, if it exists, or creates a new one. This is to maintain sales for an entire day when the program must be exited at some point.

Clear

PROCEDURE Clear();

The **Clear()** procedure works the same in all three modules. It is used to clear the previous data record from the screen. This gets the screen ready for another transaction.

Close

PROCEDURE Close();

The **Close()** routine works similarly in the three modules. Its major purpose is to hide the window that contains the current information, and to restore the window that was active previously. However, the data window is not destroyed, which saves time when the next call is made to **Template()**.

In the **Store.Close()** routine, the current sales totals are written to a newly created copy of the totals file.

Display

```
PROCEDURE Display(anItem: ItemRec);  
PROCEDURE Display(aClient: ClientRec);
```

The **Display()** procedure is only available in the **CLIENT** and **ITEMS** modules. It is used to present a data record in the open template window. This record may also be blank, in which case the current information is cleared.

Input

```
PROCEDURE Input(VAR anItem: ItemRec; Clear: BOOLEAN);  
PROCEDURE Input(VAR aClient: ClientRec; Clear: BOOLEAN);
```

Input() is used in the **CLIENT** and **ITEMS** modules to input a data record. The **Clear** parameter is used to determine whether a blank record (**Clear = TRUE**) should be edited, or whether the current contents of the data record should be edited (**Clear = FALSE**).

Search

```
PROCEDURE Search(VAR anItem: ItemRec): BOOLEAN;  
PROCEDURE Search(VAR aClient: ClientRec): BOOLEAN;
```

The **Search()** routine is used to find a data record in the respective **Btree** files. Once a record is found, it may be edited or deleted. This function is not available in the **SALES** module because the functions are performed within the actual **SALES** code.

MakeKey

```
PROCEDURE MakeKey(keyA, myItemA: ADDRESS);  
PROCEDURE MakeKey(keyA, myClientA: ADDRESS);
```

The **MakeKey** routine for **CLIENT** and **ITEMS** is the routine passed to **Btree** to create a key for the data record, for addition into the file index. In each case, an address of a record is passed in, as well as the address where the key is to be placed. The keys are of a common type — **FileKey**, from the **Globals** module. This was done so that a common comparison routine could be employed by **Btree**.

Work

```
PROCEDURE Work();
```

The **Work()** procedure is only available in the **SALES** module. Its purpose is to enact a single point-of-sale transaction. This involves the following steps:

- Find a client. This involves finding and displaying the client, based on the client number. If a close match is found, the user may search forward and backward through the index to find the correct client.
- Enter one or more inventory items. Again, a search must be made for each of the items in the sale. Once an item has been found, a quantity is entered, and running tax and total fields are calculated and displayed.
- Ask whether the sale is being made via cash or charge. If it is a charge transaction, the client's record is updated to reflect the new balance.
- Add the current transaction into the daily totals.

Summary

```
PROCEDURE Summary();
```

The **Summary()** routine is also available only with the **SALES** module. Its purpose is to present, in a pop-up window, the collected sales statistics for the day. This information should always be current, as the statistics are written each time the **Sales.Close** call is made, and reread each time the **Sales.Template** call is made.

GLOBALS Support

The **GLOBALS** module contains several of the items from the **Address** example, as well as several routines specific to the **Club** program. This has been done so that all other modules might have access to these routines.

```
VAR
```

```
  TaxRate : LONGREAL;
```

```
  DisplayFore, DisplayBack,  
  DataFore, DataBack, DisplayBorder,  
  PromptFore, PromptBack : Window.Color;
```

```
  ClientFile, ItemFile : Btree.FHandle;  
  ClientData, ItemData,  
  ClientIndex, ItemIndex : Btree.IHandle;
```


`StoreName : ARRAY[0..60] OF CHAR;`

TaxRate is the rate of tax that should be added to all sales. This number should be entered as a percentage. For instance, a five percent tax rate should be entered as 5.0, not 0.05.

The next group of variables is used to control the colors of the screens. **DisplayFore** and **DisplayBack** are the colors used for most textual output on the screen. **DataFore** and **DataBack** are used for the display of data on the template and sales screens. **PromptFore** and **PromptBack** are used at times when the user is asked for information. **DisplayBorder** is the color used to draw the border of the screen.

StoreName is the name of the store, which is centered at the top of the screen by the **UpdateTitle()** procedure.

Compare

`PROCEDURE Compare(key1, key2: ADDRESS): Btree.CmpRes;`

This procedure is used by **Btree** to compare keys for either **Client** or **Item** records.

UpdateTitle

`PROCEDURE UpdateTitle( All : BOOLEAN );`

UpdateTitle() draws the current time in a box at the top of the screen. Also, if **All** is **TRUE**, the store name and current date are written. If the current time is not different from the previous time, no display is made.

GetMenu

`PROCEDURE GetMenu(Prompt : ARRAY OF CHAR;
 OKChars : CharSet; VAR ch : CHAR);`

The **GetMenu** routine is nearly the same as the one presented in the **Address** example. The only major difference is that this version calls the **UpdateTitle** routine if no key has been pressed. Because of this, the screen will always show the correct time if a menu is being displayed.

Chapter 5

Using Btree in Multi-User Environments

The B-tree Toolkit has been designed to perform the functions necessary for seamless database access. This includes using files which are *shared* (available to more than one program, process, or user), under MS-DOS or OS/2.

Under MS/PC-DOS, a file may be shared only if DOS version 3.0 or higher is being used, and if the program **SHARE.EXE** is executed before the main application is run. This performs file and record locking overhead, which can then be utilized by user programs. (Some local area networks may provide file locking services without loading **SHARE.EXE**. These environments must provide the DOS 3.x file-locking functions, and they must emulate **SHARE**'s method of signaling that it is installed.) If **SHARE.EXE** is not loaded, all data files are opened for exclusive access.

OS/2 is always able to support file sharing, without requiring any additional software.

In both environments, files may also need to be opened as Read Only, meaning that although the file may be viewed by any process, no process is allowed to make any changes. (This allows the **Btree** module to perform more efficient buffering operations on the file, which must be avoided in read/write sharing mode.)

Since **Btree** is written to execute similarly under both operating systems, a common interface was designed to allow for multi-tasking access to files. This is implemented through the **Btree** and **FIOx** modules. For finer detail on all the calls available in these modules, see Chapter 7.

The FIOx Module

The **FIOx** module provides the low-level file-locking support used by the **Btree** module. Applications sharing files that use only the **Btree** module should never need to call **FIOx** routines directly — **Btree** performs all of the necessary actions. If you wish to use the **FIOx** module to build file-sharing capabilities other than those provided by **Btree**, the following description may prove helpful.

There are different implementations of **FIOx** for the different operating systems. The **FIOx.DOS** file provides file locking under DOS, while **FIOx.OS2** provides file locking under OS/2. You must rename the implementation for your operating system to **FIOx.MOD**, in order to link programs which use the **Btree** module.

The first task, when using files which may be multi-user, is to verify that the operating system can handle multiple accesses to a single file. The routine **FIOx.Multi()** returns a **BOOLEAN** value. This value is always **TRUE** under OS/2, as the operating system always handles multiple accesses to any file. Under MS-DOS, a **TRUE** value is returned only if the **SHARE** program was previously run. Otherwise, DOS is not set up to handle shared files, and further shared access should not be attempted.

Thus, a program may look like this:

```
PROCEDURE xxx();
VAR
    MultiOK : BOOLEAN;
BEGIN
    MultiOK := FIOx.Multi();
    IF MultiOK THEN
        SharedFileProc();
    ELSE
        IO.WrStr("I'm sorry.  ");
        IO.WrStr("Shared files are not allowed now.");
        IO.WrLn();
    END;
END xxx;
```

This enables your program to check dynamically whether shared access is allowed on files.

Of course, many programs only run under standard DOS, and do not require shared access to files. If this is the case, the routines covered in this chapter will be of little use or need.

The basis of the **FIOx** module, and of shared files in general, is the **LockRec**, or record to lock. In **FIOx**, the definition of a **LockRec** looks like this:

```
TYPE
    LockRec = RECORD
        pos,
        len : LONGCARD;
    END;
```


It is made up of two numbers. **pos** is the location within the physical file which is to be locked. **len** is the number of bytes at that offset to be locked. When this record is locked, no other tasks may have access to it, although they may look at other parts of the file.

FIOx has another **Open** call, which replicates the **FIO.Open**, but also allows the file to be opened for shared access. To open your shared file, you might do this:

```
VAR
    SharedFile : FIO.FILE;
    Shared,
    ReadOnly,
    CreateNewFile : BOOLEAN;
BEGIN
    Shared := TRUE;
    CreateNewFile := FALSE;
    ReadOnly := FALSE;
    SharedFile := FIOx.Open("SHARED.DAT", Shared, ReadOnly,
        CreateNewFile);
    ...
```

This has just opened shared-access file, called **SHARED.DAT**. Now you may begin performing locks and unlocks as your application requires. The file has been opened in read/write mode (the **ReadOnly** parameter was **FALSE**), and other processes may also open the file in read/write mode (the **Shared** parameter was **TRUE**, and **ReadOnly** was **FALSE**). If the **ReadOnly** parameter had been **FALSE**, then the file would have been opened for reading only, and other processes would have been denied permission to write.

To lock a single record, you need to create a **LockRec** structure and fill it in with the required data. For instance, the next piece of code demonstrates opening a shared access file, locking the first record, reading the first record and changing it, writing the record back out, and then unlocking the record.

```
MODULE Test4a;
IMPORT FIO, IO, FIOx;

TYPE
    DataRec = RECORD
        Count : CARDINAL;
        Name : CHAR;
    END;

VAR
    locker : FIOx.LockRec;
    ShrFile : FIO.FILE;
    datum : DataRec;
    DataCnt : CARDINAL;

BEGIN
    IF FIOx.Multi() THEN
        ShrFile := FIOx.Open("SHARED.DAT", TRUE, FALSE, FALSE);

        WITH locker DO
            pos := 0;
```



```

        len := SIZE(DataRec);
    END;
    IF NOT FIOx.Lock(ShrFile,locker) THEN
        (* error - another process has it locked *)
        HALT;
    END;
    FIO.Seek(ShrFile,0);
    DataCnt := FIO.RdBin(ShrFile,datum,SIZE(DataRec));

    WITH datum DO
        INC(Count);
        IF Count > 127 THEN
            Count := 1;
        END;
        Name := CHR(Count);
    END;
    FIO.Seek(ShrFile,0);
    DataCnt := FIO.WrBin(ShrFile,datum,SIZE(DataRec));
    FIOx.Unlock(ShrFile,locker);
    FIO.Close(ShrFile);

ELSE
    IO.WrStr("Shared Files Not Allowed.");
    IO.WrLn();

END;
END Test4a.

```

This small example, though not extremely useful, shows how simple shared file accessing may be. But before you run off to write the next great multi-tasking file manager, let's look at some even handier routines that are in **FIOx**.

Being able to lock a record in a data file is a very handy thing indeed. But what if the application requires three records. You could do three calls to **Lock()**. What if you needed 10? That would take 10 calls to **Lock()**.

What if your task was to give all 3500 employees a ten percent raise. You could make 3500 calls to **Lock()**. Or you could create a group of records to be locked at once, and make a call to **LockRange**. **LockRange** takes an **ARRAY OF LockRec**, and performs the locks on each separate record. If any lock attempt fails, then none of the locks is placed. **LockRange** will lock all of the records specified in the array, but will stop locking when it finds a record specifying a lock position of **MAX(LONGCARD)**.

Here is an example of multiple locks being effected simultaneously using **LockRange**, and then freed using **UnLockRange**.

```

MODULE Test4b;

IMPORT FIO, IO, FIOx;

TYPE
    DataRec = RECORD
        Count : CARDINAL;
        Name : CHAR;
    END;

VAR
    lockers : ARRAY[0..10] OF FIOx.LockRec;
    ShrFile : FIO.FILE;

```



```

datum      : DataRec;
DataCnt    : CARDINAL;

BEGIN
  IF FIOx.Multi() THEN

    ShrFile := FIOx.Open("SHARED.DAT",FALSE);

    FOR i := 0 TO 9 DO
      WITH lockers[i] DO
        (* here we are using a total record
           size of 50 bytes, but we are only
           interested in the first portion of
           each. *)
        pos := i*50;
        len := SIZE(DataRec);

        END;
        lockers[10].pos := MAX(LONGCARD);

        IF NOT FIOx.LockRange(ShrFile,lockers) THEN
          (* error - another process has one locked *)
          HALT;
        END;

        FOR i := 0 TO 9 DO
          FIO.Seek(ShrFile,i*50);
          DataCnt := FIO.RdBin(ShrFile,datum,
                               SIZE(DataRec));
          WITH datum DO
            INC(Count);
            IF Count > 127 THEN
              Count := 1;
            END;
            Name := CHR(Count);
          END;
          FIO.Seek(ShrFile,i*50);
          DataCnt := FIO.WrBin(ShrFile,datum,
                               SIZE(DataRec));

        END;

        FIOx.UnLockRange(ShrFile,lockers);
        FIO.Close(ShrFile);

      ELSE

        IO.WrStr("Shared Files Not Allowed.");
        IO.WrLn();

      END;
    END Test4b.
  
```

One thing that you may wish to keep in mind with **LockRange** is that locking a large group of records limits the access for other users (tasks). This may slow their processes extremely. The **LockRec** procedure, though handy, should only be used when access to multiple records is necessary for integrity. Otherwise, single **Lock** calls and data updates will keep the file available for other tasks in a much more open way.

When performing file locking, applications must assume that file locks may fail. Applications must also avoid locking records for longer than necessary. **FIOx** takes into

account both of these rules, by providing a retry and delay mechanism. When a locking routine is called, **FIOx** calls its **RetryCount** procedure variable to find out how many times the lock should be retried. Between each retry, **FIOx** calls its **DelayCount** procedure variable to determine how many milliseconds it should delay between each retry. Only after this wait/retry cycle has finished does **FIOx** return a **FALSE** value to the caller to signal its failure to achieve the lock. On initialization, **FIOx** assigns internal procedures which cause 5 retries, with 300 milliseconds waited between each retry. Applications can assign their own procedures to these variables, if desired.

Note that the **FIOx** module does not keep track of the records which have been locked. This has the side-effect that, if a single process attempts to lock the same record (or even overlapping records) twice, the second attempt will fail — neither **FIOx** nor the operating system allow nesting of file locks.

The Btree Module

As mentioned above, the B-tree Toolkit was designed to handle data files as transparently as possible. Because of this, the problems of locking and unlocking records, and determining whether the user is in a shared environment, are greatly simplified when using the **Btree** module.

When either the **Share** parameter or **FIOx.Multi()** is **FALSE**, **Btree** opens the file in deny-all mode. This prevents other processes from accessing the file in any way.

When the **Share** parameter and **FIOx.Multi()** are both **TRUE**, **Btree** opens the file in one of two ways:

1. If the **ReadOnly** parameter is **FALSE**, the file is opened in read/write and deny-none modes. This allows the process to both read and write to the file, and allows other processes to do the same. **Btree** also prevents certain internal buffering of data, since this might cause a mismatch between the information different processes have about the file size and contents.
2. If the **ReadOnly** parameter is **TRUE**, the file is opened in read-only and deny-write modes. This allows the process only to read from the file, and prevents other processes from writing to the file. Since no process can change the file, **Btree** performs the buffering it normally performs on non-shared files.

Note that it is illegal to open a file with both the **ReadOnly** and **Create** parameters set to **TRUE**. Creating an (empty) file that can never be written to is not very useful.

Also note that it is illegal to open a file with both the **Share** and **Create** parameters set to **TRUE**. This prevents other processes from interfering during the initialization of the physical file. Since (for similar reasons) creating data files and indexes is not permitted within shared files either, shared files must first be created in non-shared

mode, then closed and reopened in shared mode. (This doesn't mean that all of the data and index items must be added when the file is created — just that the actual data and index files within the physical file must be created in non-shared mode).

Btree maintains an internal lock table for each physical file. Any lock attempts within the file are checked first against the table. Thus, **Btree** locks may be nested up to 255 levels deep. There is an internal limit of 20 different lock positions at once within a physical file. (This limit may be expanded by altering a constant within the **Btree.DEF** file.) Each internal lock record contains four items of data:

1. The position being locked. (The lock length is always one byte.)
2. The implicit lock count.
3. The explicit lock count.
4. The index file with which the lock is associated.

The position is used when searching the lock table when another lock is attempted. The implicit and explicit lock counts record how deeply the lock is nested. The associated index file is used when releasing implicit locks (which are always caused by index file operations). Several explicit and implicit locks may be placed on a position at one time — only when both the implicit and explicit counts reach zero is the position actually freed for other processes to access.

As mentioned above, implicit locks are placed by index operations. When an implicit lock is being placed, the associated index field is handled as follows:

- If there is already an implicit lock, then the old associated index field is unchanged;
- If there are no locks, or only explicit locks, then the new associated index value is placed in the lock record.

On the other hand, explicit locks are placed only by direct calls from the application, and are used by applications for managing more sophisticated locking requirements.

Btree attempts to manage locks in a method convenient to the application. **Btree** automatically locks and unlocks indexes, so that simple applications can completely ignore index locking. (More complex applications involving transaction-like operations may need to call the **LockIHandle** and **UnLockIHandle** procedures themselves.) The module also automatically locks data items which are found by index searches.

An Example

The following file-sharing example is provided to illustrate how the **Btree** module makes locking operations easy to use. It first creates a data file containing 5000 random **CARDINAL** values, with an associated index sorted on the upper 8 bits of

each data item. Once the data and index is generated, the file is closed and then reopened in sharing mode, and all of the data items are displayed in sorted order (based on the index).

```

MODULE Example;
IMPORT Btree, IO, Lib;

TYPE
  RecType = CARDINAL;
  KeyType = SHORTCARD;

PROCEDURE Comp(a,b: ADDRESS): Btree.CmpRes;
BEGIN
  IF KeyType(a^) > KeyType(b^) THEN
    RETURN Btree.Greater;
  ELSIF KeyType(a^) = KeyType(b^) THEN
    RETURN Btree.Eq;
  ELSE
    RETURN Btree.Less;
  END;
END Comp;

PROCEDURE Make(keyA, recA : ADDRESS);
VAR
  key : POINTER TO KeyType;
  rec : POINTER TO RecType;
BEGIN
  key := keyA;
  rec := recA;
  key^ := SHORTCARD(rec^ DIV 256);
END Make;

VAR
  fh : Btree.FHandle;
  dh,
  ih : Btree.IHandle;
  rec : RecType;

PROCEDURE Gen();
VAR
  i : CARDINAL;
BEGIN
  FOR i := 1 TO 5000 DO
    Btree.Add(dh, Lib.Random(65535), 0);
  END;
END Gen;

PROCEDURE Open(Create : BOOLEAN);
BEGIN
  fh := Btree.Open('Example.DAT', 2, Btree.FixSize, NOT Create,
    NOT Create, Create);
  dh := Btree.OpenData(fh, 1, SIZE(RecType), Create);
  ih := Btree.OpenIndex(fh, dh, 2, Comp, Make, SIZE(KeyType), TRUE, Create);
END Open;

BEGIN
  Open(TRUE);
  Gen();
  Btree.Close(fh);
  Open(FALSE);
  Btree.Reset(dh);

```



```

LOOP
  IF Btree.Next (iH, rec) THEN
    IO.WrCard(rec);
    IO.WrLn;
  ELSE
    IF Btree.LastErr (iH) # Btree.DataLocked THEN
      EXIT;
    END;
  END;
END;
Btree.Release (iH);
Btree.Close (fH);
END Example.

```

In this example, each call to **Next**

- automatically releases any implicit locks associated with the specified index file
- locks the index
- finds the next data record
- implicitly locks the data record (with the lock associated with the index file, not the data file), and
- unlocks the index

Next returns **TRUE** if the record was found and locked, and returns **FALSE** otherwise. If the record was found but couldn't be locked, then **Next** does not advance its internal pointer, and sets **LastErr** for the index to **DataLocked**. The example will print all of the records in the file, even if it must loop indefinitely waiting for locked records to become unlocked. The call to **Release** causes all implicit locks associated with the index file to be freed. This has the effect of unlocking the record located by the final call to **Next**.

The above example shows how **Btree**'s implicit locking can be used. If necessary, applications may also use the **Lock** and **UnLock** procedures to manage explicit locks on locations within physical files — this is necessary when accessing data items allocated with **Allocate**, and read/written with **Read** and **Write**.

Applications may also use **LockIHandle** and **UnLockIHandle** to manage explicit locks on index and data files. This is useful for synchronizing updates to multiple data/index sets. Explicit locks (which are set by **Lock**, **LockIHandle**, and **Allocate**) are reset only by **UnLock**, **UnLockIHandle**, and **Deallocate**. **Btree** never unlocks an explicit lock through internal mechanisms.

Applications which share files must follow certain rules:

1. When indexes are being shared, the applications must have key derivation and comparison procedures that produce identical results.
2. If a shared data file has indexes, then the applications which add and delete from the data file must open all of the sharable indexes that require automatic updating.

3. If a shared data file is to have an unshared index, then either the shared data file must be in a read-only physical file, or the shared data file must be locked for the entire life of the auxiliary index.

Chapter 7's descriptions of the individual **Btree** procedures give detailed information on the locking behaviour used by the procedures.

The **Address2** program is a file-sharing version of the **Address** program. See the **ADDRESS2.MOD** file on the distribution disk.

Chapter 6

Btree and B-Tree Indexing

This chapter is only for those who are not faint of heart, who wish to know more about how **Btree** works, and why it is useful in the first place. To better explain this, let's look at an analogy.

Suppose you go to the store at night. There is only one other car in the parking lot. Do you spend a lot of time worrying about finding your car when you come back out? Of course not. This is the same with data files. If there are only a few records, there is no need for extensive record keeping, as all records are fairly "close" at all times.

Now, suppose you go grocery shopping on a Saturday afternoon, along with several hundred other hungry souls. In this case, you may wish to remember which row your car was in, so that you can find it on the way out before your ice cream melts.

The next step is to go shopping at the mall at Christmas. Better be prepared to know which lot you are in, as well as which row within the lot.

As if that weren't bad enough, suppose you are leaving your car at the Atlanta airport for a week, while you are out of town. Remember which building you parked in, which floor within the building, whether you were in the back or front lot on that floor, and what row you were in.

Although this sounds like a lot to remember, you may notice that there were only four pieces of information required to put you within about twenty cars of yours —

the building, floor, lot, and row. Not much harder than finding your car in the grocery lot at night, right?

Btree uses a method of file indexing known as B-tree file handling. The need for this arose from the types of scenarios mentioned above — that is, where more data are available, and required, than can be taken care of in a simple manner.

If a file is sorted, then a simple way of finding a required record is by using a binary search. This requires that we first look at the item in the middle of the file. If that record's key is greater than the key we are searching for, we then limit our search to the bottom part of the file. Again, we look at the middle record (now the middle of the bottom half), and compare it with our search record. This searching continuously divides the data file, and therefore takes very little time to complete the search. Imagine searching 10,000 records from start to back without this method!

Unfortunately, as our data files get larger, it gets to be prohibitive to keep them in sorted order. This is especially true when the data records are very large, and must be moved around each time a record is added or deleted.

Therefore, *indexing* was invented. Indexing allows you to leave your data file in any order, but to maintain an *index*, a smaller file (perhaps in memory) which only contains the key for each record, and its location in the file. This is kept in sorted order, and a binary search allows for quick location of the necessary records.

Of course, someone had too many records for the index to fit comfortably in memory. This caused problems. If the index itself was to be kept on disk, and since the disk is the slowest part of the entire loop, searches would start getting slower.

In 1970, two gentlemen by the names of Bayer and McCreight developed a solution known as *B-trees*, or *balanced trees*. Their proposal was to create an index of the index, thus needing much smaller amounts of memory to maintain the index for a large file.

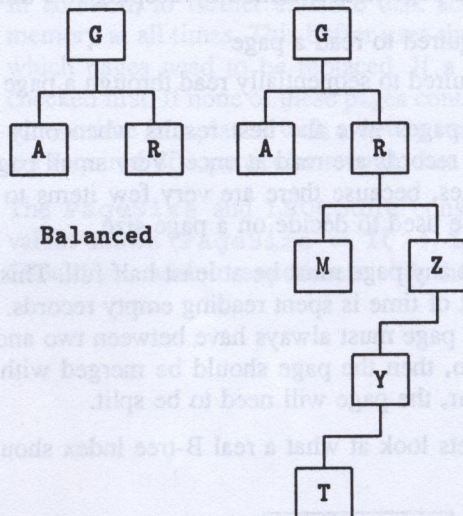
To make this work, the new index would be made up of “pages” of the old index entries. Each page would hold several items (keys), and would therefore reduce the number of disk accesses necessary to find a record. Also, the index could be maintained on disk because continuous disk reads were no longer required.

There were a few rules that had to be followed.

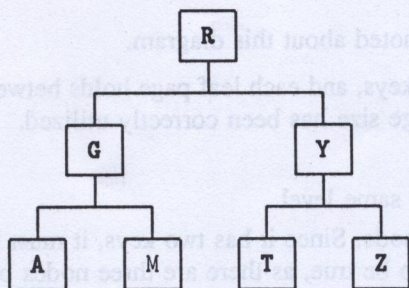
- the tree was balanced if and only if for every node in the tree, the subtree heights could differ by at most 1.
- every page except the root must contain at least n items.
- every page must contain at most $2n$ items.
- each page is either a leaf (having no descendants) or has $m + 1$ descendants, where m is the number of keys on the page.
- all “leaf pages” are at the same level.

Balanced Trees

Lets look at an illustration to better explain this concept. Here are two binary trees. One is balanced, and one is not!



Having a balanced tree involves making sure that as keys are added and deleted from the tree, that no subtree has a "limb" that is too long. In the drawing above, the unbalanced tree has several "nodes" that are unbalanced. The node containing "R" has a left node with only one child, and a right node with three generations of children. Since $3 - 1 > 1$, we have an unbalanced tree. This is also true of node "Z," which has two generations on the left and none on the right. The correct, balanced version of this tree is shown below.



You may notice in this example that all nodes are now "balanced."

Page Size

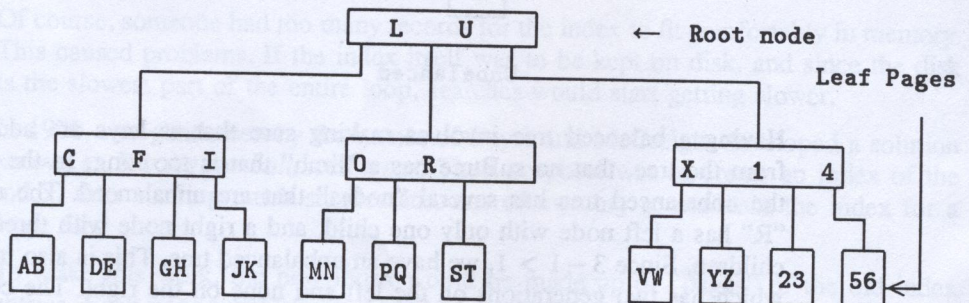
The concept of a page came from the slowness of disk drives. It was more efficient to read a *page* of data (several items at once) so that the disk would only be required once per page. The size of a page is determined by keeping two things in balance:

1. The amount of time required to read a page
2. The amount of time required to sequentially read through a page for a given key.

In other words, very large pages give the best results when only looking at disk access times, because many records are read at once. Very small pages give the best results for sequential searches, because there are very few items to search. Thus, a balance of these two must be used to decide on a page size.

The B-tree rules specify that any page must be at least half full. This also helps with disk access time — not a lot of time is spent reading empty records. So, if we select a page size of four, then the page must always have between two and four entries. If the number drops below two, then the page should be merged with other pages. If the number grows above four, the page will need to be split.

Using a page size of four, let's look at what a real B-tree index should look like.



Several observations should be noted about this diagram.

- The root node contains two keys, and each leaf page holds between two and four keys per page. Thus, the page size has been correctly utilized.
- The tree is balanced.
- All the leaf nodes are at the same level.
- The root node is not a leaf node. Since it has two keys, it must have 2 + 1, or 3 descendants. This is found to be true, as there are three nodes below it. Each of these nodes has $m + 1$ descendants, where m is the number of keys within the node. Leaf pages do not have any further descendants.

Btree Implementation

Btree has been implemented using the B-tree file handling technique. By default, the page size has been set to 1024 bytes.

In addition, to further enhance disk access, a buffer of 32 pages is maintained in memory at all times. This buffer uses the "Least Recently Used" method to determine which pages need to be replaced. If a record is needed, the pages in memory are checked first. If none of these pages contain the required data, the page with the oldest access time is replaced with a newly read page. This assures that if several accesses are required of a particular record, that the record will probably still be in memory.

The **PageSize** and **LRUCount** constants are in the **Btree.MOD** module. The values shown (**PageSize** = 1024, **LRUCount** = 32) are a good place to start. Should you wish to experiment with them, the change is very simple.

This chapter presents explanations of all **Btree** and **FICx** variables, structures, and procedures. It also presents a support module called **Misc**, which contains several routines used throughout the examples.

Btree Module

DEFINITION MODULE Btree;

CONST

Copyright (C) 1983, 1984 Jensen & Bartoore International

TYPE

RecordNode = RECORD NodeID: Integer; NodeSize: Integer; NodeData: Integer;

IndexNode = RECORD NodeID: Integer; NodeSize: Integer; NodeData: Integer;

PageFunction = PROCEDURE (ADDRESS: Integer) : Integer;

KeyFunction = PROCEDURE (ADDRESS: Integer) : Integer;

Index = POINTER TO IndexFile;

Page = POINTER TO PageFile;

IndexFile = RECORD

Index: INDEX;

IndexSize: CARDINAL;

IndexData: INDEX;

IndexSize: CARDINAL;

IndexData: INDEX;

IndexSize: CARDINAL;

IndexData: INDEX;

IndexSize: CARDINAL;

IndexData: INDEX;

IndexSize: CARDINAL;

IndexData: INDEX;

IndexSize: CARDINAL;

IndexData: INDEX;

IndexSize: CARDINAL;

IndexData: INDEX;

IndexSize: CARDINAL;

IndexData: INDEX;

IndexSize: CARDINAL;

IndexData: INDEX;

(* no error *)
 (* unable to open *)
 (* not yet available *)
 (* not an Btree *)
 (* not an Btree *)
 (* not an Btree *)
 (* not a data file *)
 (* not an index file *)

Chapter 7

B-Tree Toolkit Module Descriptions

This chapter presents explanations of all **Btree** and **FIOx** variables, structures and procedures. It also presents a support module called **Misc**, which contains several routines used throughout the examples.

Btree Module

```
DEFINITION MODULE Btree;
(*
    Copyright (C) 1988,1989 Jensen & Partners International
*)

TYPE
    AccessMode      = (FixSize,Size16,Size32,NoDealloc);
    CmpRes           = (Less,Eq,Greater);
    CompareFunction  = PROCEDURE (ADDRESS,ADDRESS) : CmpRes;
    KeyFunction      = PROCEDURE (ADDRESS,ADDRESS);
    FHandle          = POINTER TO IndexFile;
    IHandle          = RECORD
        ID : FHandle;
        In : CARDINAL;
    END;
    Errors           = (OK,
        BadOpen,
        NoSlot,
        NotFHandle,
        NotIHandle,
        NotData,
        NotIndex,
        (* no error *)
        (* unable to open *)
        (* no slot available *)
        (* not an FHandle *)
        (* not an IHandle *)
        (* not a data file *)
        (* not an index file *)
    );
```



```

BadIndex, (* incorrect index *)
BadSize, (* illegal size *)
KeyTooBig, (* key size is too large *)
ErrDupKey, (* duplicate key *)
BadFree, (* deallocate not allowed *)
BadRead, (* error during read *)
BadWrite, (* error during write *)
Locked, (* cannot acquire lock *)
NotLocked, (* must already be locked *)
FileError, (* file i/o error *)
LockOverflow, (* too many locks *)
UnknownError; (* other error *)

ErrorProc = PROCEDURE (Errors, ARRAY OF CHAR);

VAR
  ErrorHandler : ErrorProc;

CONST
  MaxKeySize = 128;
  Null = IHandle (NIL, MAX (CARDINAL));
  LockQSize = 20;

PROCEDURE Open (Name: ARRAY OF CHAR; MaxIHandle: CARDINAL;
  AMode : AccessMode;
  readOnly, shared, Create: BOOLEAN): FHandle;
PROCEDURE OpenData (F: FHandle; I, RecordSize: CARDINAL; New: BOOLEAN): IHandle;
PROCEDURE OpenIndex (F: FHandle; D: IHandle; I: CARDINAL;
  CompFct: CompareFunction; KeyFct: KeyFunction;
  KeySize: CARDINAL; DupKey, New: BOOLEAN): IHandle;
PROCEDURE Close (VAR F: FHandle);
PROCEDURE FreeIHandle (VAR I: IHandle);
PROCEDURE Flush (F: FHandle);

PROCEDURE LastError (H: IHandle) : Errors;
PROCEDURE LastFError (F: FHandle) : Errors;

PROCEDURE Add (D: IHandle; Data: ARRAY OF BYTE; Length: CARDINAL);
PROCEDURE AddIndex (I: IHandle; Key: ARRAY OF BYTE; DataLoc: LONGCARD);

PROCEDURE Delete (D: IHandle);
PROCEDURE DeleteIndex (I: IHandle; Key: ARRAY OF BYTE; DataLoc: LONGCARD);

PROCEDURE Change (D: IHandle; Data: ARRAY OF BYTE; Length: CARDINAL);

PROCEDURE Find (I: IHandle; Key: ARRAY OF BYTE;
  VAR Data: ARRAY OF BYTE): BOOLEAN;
PROCEDURE FindIndex (I: IHandle; Key: ARRAY OF BYTE;
  VAR DataLoc: LONGCARD): BOOLEAN;

PROCEDURE Search (I: IHandle; Key: ARRAY OF BYTE;
  VAR Data: ARRAY OF BYTE): BOOLEAN;
PROCEDURE SearchIndex (I: IHandle; Key: ARRAY OF BYTE;
  VAR DataLoc: LONGCARD): BOOLEAN;

PROCEDURE Next (I: IHandle; VAR Data: ARRAY OF BYTE): BOOLEAN;
PROCEDURE NextIndex (I: IHandle; VAR DataLoc: LONGCARD): BOOLEAN;

PROCEDURE Prev (I: IHandle; VAR Data: ARRAY OF BYTE): BOOLEAN;
PROCEDURE PrevIndex (I: IHandle; VAR DataLoc: LONGCARD): BOOLEAN;

PROCEDURE Reset (H: IHandle);

```



```

PROCEDURE SetSyncMode(D: IHandle; On: BOOLEAN);

PROCEDURE ClearIndex(VAR I: IHandle);

PROCEDURE Release(H: IHandle);
PROCEDURE Lock (F: FHandle; DataLoc: LONGCARD): BOOLEAN;
PROCEDURE Unlock(F: FHandle; DataLoc: LONGCARD);
PROCEDURE LockIHandle (H: IHandle): BOOLEAN;
PROCEDURE UnlockIHandle(H: IHandle);

PROCEDURE Allocate (F: FHandle; Length: LONGCARD) : LONGCARD;
PROCEDURE DeAllocate(F: FHandle; Position: LONGCARD);

PROCEDURE Read (F: FHandle; Position: LONGCARD; Length: CARDINAL;
VAR Data: ARRAY OF BYTE);
PROCEDURE Write(F: FHandle; Position: LONGCARD; Length: CARDINAL;
Data: ARRAY OF BYTE);

END Btree.

```

The Btree module provides all of the facilities for managing logical data and index files, including file locking support.

Btree's file locking calls may be called for non-shared files. In this case, the calls are ignored. This means that applications written to share files will run unaltered in an environment where sharing is not supported.

TYPE

```

AccessMode      = (FixSize, Size16, Size32, NoDealloc);
CmpRes          = (Less, Eq, Greater);
CompareFunction = PROCEDURE (ADDRESS, ADDRESS) : CmpRes;
KeyFunction     = PROCEDURE (ADDRESS, ADDRESS);
FHandle         = POINTER TO IndexFile;
IHandle         = RECORD
    ID : FHandle;
    In : CARDINAL;
END;

Errors          = (OK,
    BadOpen,
    NoSlot,
    NotFHandle,
    NotIHandle,
    NotData,
    NotIndex,
    BadIndex,
    BadSize,
    KeyTooBig,
    ErrDupKey,
    BadFree,
    BadRead,
    BadWrite,
    Locked,
    NotLocked,
    FileError,
    LockOverflow,
    UnknownError);

(* no error *)
(* unable to open *)
(* no slot available *)
(* not a FHandle *)
(* not an IHandle *)
(* not a data file *)
(* not an index file *)
(* incorrect index *)
(* illegal size *)
(* key size is too large *)
(* duplicate key *)
(* deallocate not allowed *)
(* error during read *)
(* error during write *)
(* cannot acquire lock *)
(* must already be locked *)
(* file i/o error *)
(* too many locks *)
(* other error *)

ErrorProc = PROCEDURE (Errors, ARRAY OF CHAR);

```

VAR

```
ErrorHandler : ErrorProc;
```



```

CONST
  MaxKeySize   = 128;
  Null         = IHandle(NIL, MAX(CARDINAL));
  LockQSize    = 20;

```

The **AccessMode** data type is used to indicate the type of access that can be expected for the physical file. **FixSize** indicates that only fixed-size data records are to be used. **Size16** indicates that variable-length data records up to 64K bytes each may be expected. **Size32** allows variable-length data records whose size is limited only by the operating system. **NoDealloc** indicates that no deallocation will ever take place.

CmpRes is used as the return value from a **CompareFunction** (below), and indicates the outcome of the comparison.

CompareFunction is the type of the user-defined comparison routine. Functions of this type are used by the **Btree** index system to order keys in the B-tree. The **ADDRESS** parameters contain the addresses of the two keys which should be compared. A function for comparing **CARDINAL** keys might look like this:

```

PROCEDURE Compare(key1, key2: ADDRESS): Btree.CmpRes;
BEGIN
  IF CARDINAL(key1^) < CARDINAL(key2^) THEN
    RETURN Btree.Less;
  ELSIF CARDINAL(key1^) = CARDINAL(key2^) THEN
    RETURN Btree.Eq;
  ELSE
    RETURN Btree.Greater;
  END;
END Compare;

```

CompareFunctions must not make any **Btree** calls. The comparison function used for an index must always produce the same result for the same keys. Failure to abide by these rules may cause the index to be corrupted.

The next procedure type, **KeyFunction**, is the type of the user-defined function for creating a key from a record of the type being stored. The first parameter is the address where the key is to be created, while the second parameter is the address of the record the key must be created from. In the spirit of the above example:

```

TYPE
  RecType = RECORD
    a, b : CARDINAL;
    c : ARRAY[0..30] OF CHAR;
  END;
  KeyType = CARDINAL;

PROCEDURE MakeKey(keyA, recA: ADDRESS);
VAR
  key : POINTER TO KeyType;
  rec : POINTER TO RecType;

```



```

BEGIN
    key := keyA;
    rec := recA;
    key^ := ORD(rec^.c);
END MakeKey;

```

KeyFunctions must not make any **Btree** calls. The key derivation function used for an index must always derive the exact same key from a given data record. Failure to abide by these rules may cause the index to be corrupted.

FHandle is the handle for a **Btree** physical file, while **IHandle** is a handle of a **Btree** logical file (both indexes and data tables). The **IHandle** fields contain the **FHandle** (physical file) in which the logical file is stored, as well as the internal slot number being used. (Applications don't need to make use of the fields within objects of type **IHandle**.)

The **Errors** enumeration defines the possible error codes generated by the **Btree** module. The **ErrorProc** procedure type is the procedure type necessary for application defined error handlers. The **ErrorHandler** procedure variable contains the error handler which **Btree** will call for serious errors. Specific error return information is given below, with each procedure description.

The constant **MaxKeySize** indicates the maximum size of an index key. If this size is increased, the user should be sure that there is enough stack space to accommodate the local variables of this size in the **Btree** system, and should make sure that the page size (declared in the implementation module) is large enough to contain at least three keys and associated work space.

The **Null** constant aggregate is used as a return value when errors prevent the **OpenData** and **OpenIndex** procedure from completing their operation.

In **Btree**, if file locking is in use, files may be protected from other tasks. To do this, a queue of the currently set locks is kept with each open **FHandle**. The value **LockQSize** determines the size of this internal queue, and is the limit on the number of concurrently locked file positions within each **FHandle**. (Each file position may be locked explicitly up to 255 times, and implicitly up to 255 times.)

Open

```

PROCEDURE Open(Name: ARRAY OF CHAR;
               MaxIHandle: CARDINAL;
               Mode: AccessMode;
               ReadOnly,
               Shared,
               Create: BOOLEAN): FHandle;

```

Open is used to open a physical file. If **Create** is **TRUE**, the file will be created.

The file name is passed in the variable **Name**. **MaxIHandle** specifies how many logical handles (data and index files) will be available for use within this physical file. At least two should almost always be available — one for a data file, and one for its index. More may be used if desired. The available logical handles are numbered 1 .. **MaxIHandle**.

Mode is the type of record access which will be used for this file, and is one of **FixedSize**, **Size16**, **Size32**, or **NoDealloc** as described above.

ReadOnly and **Shared** are used in multi-tasking programs and environments, including OS/2. **ReadOnly** determines whether the task opening the file will be able to write to the file, or only read its contents. If **Shared** is **TRUE**, other tasks may also read (and write, if **ReadOnly** is **FALSE**) this file concurrently. Locking will be performed, as explained in Chapter 5.

It is illegal to open a file with both **Create** and **ReadOnly** set to **TRUE**, or with **Create** and **Shared** set to **TRUE**.

When reopening a file (that is, **Create** is **FALSE**), **MaxIHandle** and **Access-Mode** must match the values which were given when the file was created.

When this procedure fails, it does not have a handle with which to associate an error value. Thus, errors are only reported by calling the error handler procedure.

Error condition	Last Error Value	ErrorHandler Called
ReadOnly with Create	n/a	yes (BadOpen)
Shared with Create	n/a	yes (BadOpen)
File not found	n/a	yes (FileError)
AccessMode mismatch	n/a	yes (BadOpen)
MaxIHandle mismatch	n/a	yes (BadOpen)

If the open attempt fails, the procedure returns **NIL** for the **FHandle**.

OpenData

```
PROCEDURE OpenData (F: FHandle;
                   Slot,
                   RecordSize: CARDINAL;
                   New: BOOLEAN): IHandle;
```

OpenData opens a logical data file. If **New** is **TRUE**, a new data file structure is initialized. **F** is the handle of the actual physical file.

Slot indicates which of the logical handles belonging to the physical file should be used for this data file. **Slot** must be between 1 and the maximum value specified when the **Open** procedure was called on the physical file.

RecordSize indicates the size of the data records to be used. If the size is given as 0, then it is assumed that variable length records will be used — in this case, each time a record is added (see the **Add** procedure), the length for that variable record must be specified.

It is illegal to create a new logical data file within a physical file which is being shared.

The **RecordSize** may not be 0 (meaning variable length records), if the physical file was created with an **AccessMode** of **FixedSize**.

When reopening a file (that is, **New** is **FALSE**), **RecordSize** must match the value used when the data file was created.

This procedure associates its error codes with the **F** physical file.

Error condition	Last Error Value	ErrorHandler Called
Not a valid FHandle	n/a	yes (NotFHandle)
New with sharing	BadOpen	yes
Slot not empty	NoSlot	yes
RecordSize=0 and FixSize	BadSize	yes
RecordSize mismatch	BadSize	yes

If an **OpenData** attempt fails, the value **Null** is returned.

OpenIndex

```
PROCEDURE OpenIndex(F: FHandle;
  D: IHandle;
  Slot: CARDINAL;
  CompFct: CompareFunction;
  KeyFct: KeyFunction;
  KeySize: CARDINAL;
  DupKey,
  New: BOOLEAN): IHandle;
```

OpenIndex is used to open (or create) a logical index file. If **New** is **TRUE**, then the handle (logical file) will be initialized as an empty B-tree; otherwise an existing index will be reopened. **F** is the handle of the actual physical file.

D is the handle of the logical data file with which this index will be associated. If **Null** is used, the index will not be associated with any file.

CompFct and **KeyFct** are the functions used by the **Btree** system to create and compare keys for this index. Their use is described above. (If **KeyFct** is **NULL-PROC**), then the index will not be associated with a data file.) **DupKey** indicates whether duplicate key entries should be allowed.

Slot indicates which of the logical handles belonging to the physical file should be used for this index. **Slot** must be between 1 and the maximum value specified when the **Open** procedure was called on the physical file. **KeySize** indicates the maximum size of the keys for this index.

Creating an index associated with a data file does not automatically cause all records in the data file to be indexed. It only causes additions and deletions to the data file by this process to update the index.

It is illegal to create a new logical index file within a physical file which is being shared.

When reopening a file (that is, **New** is **FALSE**), **KeySize** and **DupKey** must match the values used when the index file was created.

The **D** parameter must reference a data file.

This procedure associates its error codes with the **F** physical file.

Error condition	Last Error Value	ErrorHandler Called
Not a valid FHandle	n/a	yes (NotFHandle)
New with sharing	BadOpen	yes
Slot not empty	NoSlot	yes
KeySize mismatch	BadOpen	yes
DupKey mismatch	BadOpen	yes
Not a data handle	NotData	yes

If an **OpenIndex** attempt fails, the value **Null** is returned.

Close

PROCEDURE Close(VAR **F**: FHandle);

Close is used to close a given physical file. All logical data and index files are also closed at this time, and all locks are released.

This procedure associates its error codes with the **F** physical file.

Error condition	Last Error Value	ErrorHandler Called
Not a valid FHandle	n/a	yes (NotFHandle)
Unable to lock	Locked	no

If the **Close** procedure is successful, then **F** is given the value **NIL**. If the **Close** fails, then **F** is unchanged.

FreeHandle

PROCEDURE FreeIHandle (VAR H: IHandle);

This procedure unconditionally releases the index handle. It is essentially a close operation for data and index files.

This procedure associates its error codes with the **H** index or data file.

Error condition	Last Error Value	ErrorHandler Called
Not a valid handle	n/a	yes (NotIHandle)
Unable to lock	Locked	no

If the procedure is successful, then **I** is given the value **Null**. If it fails, then **I** is unchanged.

Flush

PROCEDURE Flush (F: FHandle);

The **Flush** procedure instructs the **Btree** module to write any buffered data for the **FHandle** to the disk file, and to update the disk file's directory entry. An application is never required to call this procedure. Rather, the procedure is provided for applications which desire to save data in case of system crash.

This procedure associates its error codes with the **F** physical file.

Error condition	Last Error Value	ErrorHandler Called
Not a valid FHandle	n/a	yes (NotFHandle)
Unable to lock	Locked	no

If the procedure fails, no damage is done — other than the fact that the data has not been written to the disk.

LastError

```
PROCEDURE LastError (H: IHandle): Errors;
```

```
PROCEDURE LastFError (F: FHandle): Errors;
```

These procedures allow the application to determine if an error was generated by one of the **Btree** routines. **LastError** returns the error status of either a data file or an index file, while **LastFError** returns the error status of a physical file.

If an invalid handle is passed to either of these procedures, they return the value **NotIHandle** and **NotFHandle**, respectively. Neither procedure clears the last error value.

Add

```
PROCEDURE Add(D: IHandle; Data: ARRAY OF BYTE; Length: CARDINAL);
```

```
PROCEDURE AddIndex(I: IHandle; Key: ARRAY OF BYTE; DataLoc: LONGCARD);
```

Add is used to add a data record, **Data**, to the logical data file **D**, with automatic updates of the associated indexes. If the logical data file was opened with variable sized records, then the **Length** parameter is used to determine the length of the data record.

AddIndex adds a **Key** to the logical index **I**. The data record is at position **DataLoc** within the file.

Neither procedure requires any locks to be in place before being called; nor does either procedure leave any additional locks in place upon return.

The **Add** procedure only accepts data file handles. The **AddIndex** procedure only accepts index handles.

The **Add** procedure attempts to lock the data file and all of its associated indexes. The **AddIndex** procedure attempts to lock only the specified index file. Neither procedure requires any locks be in place before being called, or leaves any additional locks in place upon return.

The **Add** procedure associates its error codes with the **D** data file, while the **AddIndex** procedure associates errors codes with the **I** index file.

Error condition	Last Error Value	ErrorHandler Called
Not a data handle	NotData	yes
Not an index	NotIndex	yes
Unable to lock	Locked	no

If the procedure fails, then the record has not been added, and none of the indexes have been affected.

Delete

```
PROCEDURE Delete(D: IHandle);
```

```
PROCEDURE DeleteIndex(I: IHandle; Key: ARRAY OF BYTE;  
DataLoc: LONGCARD);
```

Delete is used to delete a record previously found with calls to **Add**, **Search**, **Find**, **Next**, or **Prev**. All necessary indexes will be updated. The **Delete** procedure attempts to lock the data file and all of its associated index files. There must be at least one lock on the record being deleted, and all locks on that record will be freed after the record is deleted.

DeleteIndex deletes an entry from an index. **DataLoc** is the address in the file associated with the given data record. The **DeleteIndex** procedure attempts to lock only the specified index file. The data location is not required to be locked, and it will not be unlocked if it is locked.

The **Delete** procedure only accepts data file handles. The **DeleteIndex** procedure only accepts index handles.

The **Delete** procedure associates its error codes with the **D** data file, while the **DeleteIndex** procedure associates errors codes with the **I** index file.

Error condition	Last Error Value	ErrorHandler Called
Not a data handle	NotData	yes
Not an index	NotIndex	yes
Not locked	NotLocked	yes
Unable to lock	Locked	no

If the procedure fails, then the record has not been deleted, and none of the indexes have been affected.

Change

```
PROCEDURE Change(D: IHandle; Data: ARRAY OF BYTE; Length: CARDINAL);
```

The **Change** procedure is used to update the data in a data record. It may be used only if the changes do not affect any of the keys derived from the data. The last data record accessed in the data file is the record which will be changed. The application must already have a lock on the data item. This lock is not released by this procedure. If variable length records are being used, then the **Length** parameter must match the current record length.

This procedure only accepts data handles.

Error condition	Last Error Value	ErrorHandler Called
Index key change	BadIndex	yes
Record length change	BadSize	yes
Not a data handle	NotData	yes
Not locked	NotLocked	yes

If the procedure fails, then the data have not been written.

Find

```
PROCEDURE Find(I: IHandle; Key: ARRAY OF BYTE;
               VAR Data: ARRAY OF BYTE): BOOLEAN;
PROCEDURE FindIndex(I: IHandle; Key: ARRAY OF BYTE;
                   VAR DataLoc: LONGCARD): BOOLEAN;
```

Find is used to locate a data record based on the passed **Key**, using the data file associated with the index file **I**. It locates the record using the B-tree; then it reads the data into **Data**, and returns a **TRUE** if the record was found. If no match was discovered, **FALSE** is returned. If the procedure is successful, it places an implicit lock on the data location associated with the index file used for the search.

FindIndex performs the same function, but returns the location of the record in the file, without actually reading the data. **Find** actually uses **FindIndex** to locate the record, and then reads the data into the **Data** parameter. It does not place a lock on the data location.

Both procedures remove all implicit locks associated with the index file, regardless of whether the search is successful or not, and regardless of whether an error occurs or not.

These two procedures only accept index handles.

Both procedures associate their error codes with the **I** index file.

Error condition	Last Error Value	ErrorHandler Called
Not an index	NotIndex	yes
Unable to lock	Locked	no

If an error occurs, both procedures return **FALSE**.

Search

```
PROCEDURE Search(I: IHandle; Key: ARRAY OF BYTE;
  VAR Data: ARRAY OF BYTE): BOOLEAN;
```

```
PROCEDURE SearchIndex(I: IHandle; Key: ARRAY OF BYTE;
  VAR DataLoc: LONGCARD): BOOLEAN;
```

Search is used to find a record with a key greater than or equal to the key passed into the routine. The search is performed in the data file associated with the index file **I**. If a record is found, **TRUE** is returned, and the record is placed in the **Data** parameter. If no match is found, **FALSE** is returned. If the procedure is successful, it places an implicit lock on the data location associated with the index file used for the search.

SearchIndex performs the same function, but returns the location of the data in **DataLoc**, rather than actually reading in the data. **Search** actually uses procedure **SearchIndex** to locate the record, and then reads the data into the **Data** parameter. It does not place a lock on the data location.

Both procedures remove all implicit locks associated with the index file, regardless of whether the search is successful or not, and regardless of whether an error occurs or not.

These two procedures only accept index handles.

Both procedures associate their error codes with the **I** index file.

Error condition	Last Error Value	ErrorHandler Called
Not an index	NotIndex	yes
Unable to lock	Locked	no

If an error occurs, both procedures return **FALSE**.

Next

```
PROCEDURE Next(I: IHandle; VAR Data: ARRAY OF BYTE): BOOLEAN;
```

```
PROCEDURE NextIndex(I: IHandle; VAR DataLoc: LONGCARD): BOOLEAN;
```

Next and **NextIndex** are used to proceed through an index in order. **Next** retrieves the next data record from the data file associated with index file **I**. **NextIndex** returns the location in the data file of the record.

Both routines return **TRUE** if another record existed in the file, or **FALSE** if a record was not available. **Next** places an implicit lock (associated with the index file) on the data location. **NextIndex** does not place an implicit lock on the data location.

Both procedures remove all implicit locks associated with the index file, regardless of whether the search is successful or not, and regardless of whether an error occurs or not.

These two procedures only accept index handles.

Both procedures associate their error codes with the **I** index file.

Error condition	Last Error Value	ErrorHandler Called
Not an index	NotIndex	yes
Unable to lock	Locked	no

If an error occurs, both procedures return **FALSE**.

Prev

```
PROCEDURE Prev(I: IHandle; VAR Data: ARRAY OF BYTE): BOOLEAN;
```

```
PROCEDURE PrevIndex(I: IHandle; VAR DataLoc: LONGCARD): BOOLEAN;
```

Prev and **PrevIndex** are used to proceed through an index in reverse. **Prev** retrieves the previous data record from the data file associated with index file **I**. **PrevIndex** returns the location in the data file of the record.

Both routines return **TRUE** if another record existed in the file, or **FALSE** if a record was not available.

Both procedures remove all implicit locks associated with the index file, regardless of whether the search is successful or not, and regardless of whether an error occurs or not.

These two procedures only accept index handles.

Both procedures associate their error codes with the **I** index file.

Error condition	Last Error Value	ErrorHandler Called
Not an index	NotIndex	yes
Unable to lock	Locked	no

If an error occurs, both procedures return **FALSE**.

Reset

PROCEDURE Reset (H: IHandle);

Reset is used to reset index files. If the handle passed to it is a data file, then all of the indexes associated with the data file are reset. If the handle is an index file, then only that index file is reset.

Resetting an index file causes a call to **Next** to return the first record, while a call to **Prev** returns the last record.

Error condition	Last Error Value	ErrorHandler Called
Not a valid handle	n/a	yes (NotIHandle)

SetSyncMode

PROCEDURE SetSyncMode (D: IHandle; On: BOOLEAN);

Normally, each index maintains an independent "current record" pointer. If procedure **SetSyncMode** is called with its second parameter set to **TRUE**, then any index operation will cause all indexes related to that data file to be moved to the located record. The procedure may be called with the second parameter as **FALSE**, to disable index synchronization.

Immediately upon calling this procedure to start synchronization, all of the indexes are synchronized on the record which was last referenced in the data file. When this procedure is called to make the indexes independent, each index's "current record" pointer is set to the last record to which they were synchronized.

This procedure may only be called for data file handles.

Error condition	Last Error Value	ErrorHandler Called
Not a data handle	NotData	yes

ClearIndex

```
PROCEDURE ClearIndex(VAR I: IHandle);
```

This procedure removes a logical index from use. In doing so, the storage used is freed, the connection with the (optional) data file is removed, and the handle is made available for other use. This procedure cannot be called for an index file within a physical file which is being shared.

This procedure may only be called for index handles.

The procedure associates its error codes with the **I** index file.

Error condition	Last Error Value	ErrorHandler Called
File is shared	BadAccess	yes
Not an index	NotIndex	yes

If an error occurs, then the index has not been affected. If the procedure is successful, then **I** is given a value of **Null**.

Release

```
PROCEDURE Release(H: IHandle);
```

The **Release** procedure is used to free all of the implicit locks associated with an index file. If called with an index handle, then only the locks associated with that index are released. If called with a data file handle, then all of the implicit locks placed on all of the indexes of that data handle are released.

Error condition	Last Error Value	ErrorHandler Called
Not a valid handle	n/a	yes (NotIHandle)

Lock

```
PROCEDURE Lock(F: FHandle; DataLoc: LONGCARD): BOOLEAN;
```

```
PROCEDURE UnLock(F: FHandle; DataLoc: LONGCARD);
```

Lock and **UnLock** are used by applications to control explicit locks.

Btree automatically locks and unlocks data records in certain instances (see above). **Lock** and **UnLock** allow the programmer to perform these functions at any time. The **Lock** and **UnLock** calls can be nested, so that within one application, multiple locks may exist on one data record. Outside the program, the operating system knows of only one lock for the record.

To use **Lock** and **UnLock**, pass the handle of the physical file, and the actual position within the file at which the record starts. This position may be obtained from a call to **FindIndex**, **SearchIndex**, **NextIndex**, **PrevIndex**, or **Allocate**.

Lock returns a **BOOLEAN** value, which describes the outcome of the function. If **Lock** returns **TRUE**, either the record was locked, or it was previously locked within the same task, and a layer of nesting has been added. If **Lock** returns **FALSE**, another task has already locked the data record.

Both procedures associate their error codes with the **F** physical file.

Error condition	Last Error Value	ErrorHandler Called
Not a valid handle	n/a	yes (NotFHandle)
Too many locks	LockOverflow	yes
Not locked	NotLocked	yes
Unable to lock	Locked	no

LockIHandle

```
PROCEDURE LockIHandle(H: IHandle): BOOLEAN;
```

```
PROCEDURE UnlockIHandle(H: IHandle);
```

LockIHandle and **UnlockIHandle** are used to manage explicit locks on data or index files. They behave in much the same way that **Lock** and **Unlock** do.

Locking a data file prevents other processes from adding or deleting records. Locking an index file prevents other processes from using the index in any way.

Both procedures associate their error codes with the **I** index file.

Error condition	Last Error Value	ErrorHandler Called
Not a valid handle	n/a	yes (NotIHandle)
Too many locks	LockOverflow	yes
Not locked	NotLocked	yes
Unable to lock	Locked	no

Allocate

```
PROCEDURE Allocate(F: FHandle; Length: LONGCARD): LONGCARD;
```

Allocate is used to allocate a block in the physical file **F** of length **Length**. The value returned is the absolute location within the file of the allocated block. The location will have an explicit lock placed upon it. Note that this is the only case where the module's automatic locking mechanism places an explicit lock. The application must use **Unlock** to allow access to other processes.

The procedure associates its error codes with the **F** physical file.

Error condition	Last Error Value	ErrorHandler Called
Not a valid handle	n/a	yes (NotFHandle)
Unable to lock	Locked	no

If an error occurs, the procedure returns **MAX (LONGCARD)**.

DeAllocate

PROCEDURE DeAllocate(F: FHandle; Position: LONGCARD);

DeAllocate releases a block previously allocated with a call to **Allocate**. The location must already be locked, and all locks upon it will be removed after it has been freed.

The procedure associates its error codes with the **F** physical file.

Error condition	Last Error Value	ErrorHandler Called
Not a valid handle	n/a	yes (NotFHandle)
Not locked	NotLocked	yes
Unable to lock	Locked	no

If the procedure fails, then the space has not been deallocated.

Read

PROCEDURE Read(F: FHandle; Position: LONGCARD; Length: CARDINAL;
VAR Data: ARRAY OF BYTE);

Read reads **Length** bytes from physical file **F**, beginning at position **Position**, and places them into the variable **Data**. The application must have a lock (either implicit or explicit) already placed on the location. This procedure does not affect that lock.

The procedure associates its error codes with the **F** physical file.

Error condition	Last Error Value	ErrorHandler Called
Not a valid handle	n/a	yes (NotFHandle)
File error	FileError	yes
Not locked	NotLocked	yes
Unable to lock	Locked	no

If the procedure fails, then the data has not been read.

Write

```
PROCEDURE Write(F: FHandle; Position: LONGCARD;
               Length: CARDINAL; Data: ARRAY OF BYTE);
```

Write writes **Length** bytes from the variable **Data** into the physical file **F**, at absolute offset **Position**. The application must have a lock (either implicit or explicit) already placed on the location. This procedure does not affect that lock.

The procedure associates its error codes with the **F** physical file.

Error condition	Last Error Value	ErrorHandler Called
Not a valid handle	n/a	yes (NotFHandle)
File error	FileError	yes
Not locked	NotLocked	yes
Unable to lock	Locked	no

If the procedure fails, then the data has not been written.

FIOx Module

```
DEFINITION MODULE FIOx;
(*
    Copyright (C) 1988,1989 Jensen & Partners International
*)
IMPORT FIO;

TYPE
    LockRec = RECORD
        pos,
        len : LONGCARD;
    END;
    CountProc = PROCEDURE(): CARDINAL;

VAR
    RetryCount,
    DelayCount : CountProc;

PROCEDURE Multi(): BOOLEAN;
PROCEDURE Lock(f: FIO.File; x: LockRec): BOOLEAN;
PROCEDURE Unlock(f: FIO.File; x: LockRec);
PROCEDURE LockRange(f: FIO.File; x: ARRAY OF LockRec): BOOLEAN;
    (* stops before n if x[n].pos = MAX(LONGCARD) *)
PROCEDURE UnlockRange(f: FIO.File; x: ARRAY OF LockRec);
    (* stops before n if x[n].pos = MAX(LONGCARD) *)
PROCEDURE Open(Name: ARRAY OF CHAR; Share,ReadOnly,
               Create: BOOLEAN): FIO.File;
PROCEDURE Flush(f: FIO.File);
```


END FIOx.

The **FIOx** module provides low-level file locking support. It is designed as an extension to the **FIO** standard library.

```

TYPE
  LockRec = RECORD
    pos,
    len : LONGCARD;
  END;
  CountProc = PROCEDURE(): CARDINAL;

VAR
  RetryCount,
  DelayCount : CountProc;

```

LockRec is a data structure used to represent a portion of a file which is to be locked or unlocked. This structure includes a position and a length. These refer to the actual physical position within the file, and the length of the piece of data in question.

CountProc is a **Procedure** type that returns a **CARDINAL** value. This is used to return requested numbers. For instance, the **RetryCount** and **DelayCount** variables are **CountProcs** that return the requested time.

Multi

```
PROCEDURE Multi(): BOOLEAN;
```

The procedure **Multi** is used to determine if a multi-user processing environment is in effect. Under OS/2, this will always return a **TRUE** value. If the program is running under DOS, a **TRUE** value will only be returned if the **SHARE** program has been run, and file sharing is in effect.

Lock

```
PROCEDURE Lock(f: FIO.File; x: LockRec): BOOLEAN;
```

Lock is used to reserve a specific record for the current task, and disallow access for others. The call includes the file in question, and a **LockRec** which points to the position and length of the item to be locked. The procedure returns **FALSE** if the lock is unsuccessful.

UnLock

```
PROCEDURE UnLock(f: FIO.File; x: LockRec);
```

UnLock works identically to **Lock**, but removes the restrictions placed on a record by a previous call to **Lock**.

LockRange

```
PROCEDURE LockRange(f: FIO.File; x: ARRAY OF LockRec): BOOLEAN;
```

LockRange is identical to **Lock**, except that an array of the **LockRecs** are passed into the routine, and all the locks are then made from the array. If an array element of **x** has its **pos** field equal to **MAX (LONGCARD)**, then **LockRange** will not attempt to lock that record or any record listed after it in the array. Otherwise, **LockRange** will attempt to lock all of the records listed in the array. The procedure returns **FALSE** if any of the locks are unsuccessful. In that case, the procedure does not leave any of the locks in place.

UnLockRange

```
PROCEDURE UnLockRange(f: FIO.File; x: ARRAY OF LockRec);
```

UnLockRange operates like **LockRange** and **UnLock**. This procedure accepts an array of **LockRecs**, and then allows further access to the items in the list by unlocking them from the system.

Open

```
PROCEDURE Open(Name: ARRAY OF CHAR;  
               Share,ReadOnly,Create: BOOLEAN): FIO.File;
```

The **Open** procedure is used to open a file, for shared access if **Multi()** is **TRUE**. This can be used in place of the standard **Open** routine in the **FIO** module. It returns **MAX (CARDINAL)** if the attempt to open is unsuccessful.

Flush

```
PROCEDURE Flush(f: FIO.File);
```

The **Flush** procedure causes the operating system to flush all of its internal buffers, and to update the file's directory entry.

Misc Module

```
DEFINITION MODULE Misc;
```

```
(* This file contains some useful procedures that
   you may wish to incorporate in your application(s) *)
```

```
TYPE
```

```
DayType = (Sunday, Monday, Tuesday, Wednesday, Thursday,
           Friday, Saturday);
```

```
PROCEDURE GetTime ( VAR Hrs,Mins,Secs,Hsecs : CARDINAL ) ;
```

```
(*
   Returns the time as returned by DOS
   Hour : 0..23 ; Mins : 0..59 ; Seconds 0..59 ; Hsecs : 0..99
*)
```

```
PROCEDURE GetDate ( VAR Year,Month,Day : CARDINAL ;
                   VAR DayOfWeek : DayType ) ;
```

```
(*
   Returns the date as returned by DOS
   Year : 1980..2099 ; Month : 1..12 ; Day 1..31
*)
```

```
PROCEDURE InputStr ( VAR S : ARRAY OF CHAR; MaxLen : CARDINAL ) ;
```

```
(* Inputs a string, using Window module, allowing normal line editing *)
```

```
END Misc.
```

The **Misc** module provides several general purpose procedures.

GetTime

```
PROCEDURE GetTime (VAR Hrs,Mins,Secs,Hsecs: CARDINAL);
```

The **GetTime** procedure returns the current time, as reported by DOS. The variables have the ranges shown below.

Hrs (Hours)	0..23
Mins (Minutes)	0..59
Secs (Seconds)	0..59
Hsecs (1/100 seconds)	0..99

GetDate

```
PROCEDURE GetDate (VAR Year,Month,Day: CARDINAL;
                   VAR DayOfWeek: DayType);
```

TYPE

```
DayType = (Sunday, Monday, Tuesday, Wednesday, Thursday,
           Friday, Saturday);
```

GetDate returns the date as currently reported by DOS. The **Year**, **Month** and **Day** parameters are returned in the ranges shown below. The **DayOfWeek** parameter returns a variable of the **DayType** type, as shown above.

Year	1980..2099
Month	1..12
Day	1..31

InputStr

```
PROCEDURE InputStr (VAR S: ARRAY OF CHAR; MaxLen: CARDINAL);
```

InputStr allows the entry of a string, using the **Window** module for cursor movement. The **MaxLen** parameter determines the length of the input field. During the entry of the string, cursor keys allow movement within the string, and insert and delete work as expected.

Index

AccessMode, 44, 46, 47
Add, 47, 50, 51
AddIndex, 50
AddName, 15
Address program, 5, 13, 14,
22, 23, 34
ADDRESS, 44
Address Specific
 Routines, 15
ADDRESS.MOD, 5
Address2 program, 34
ADDRESS2.MOD, 34
AddrRec, 15
All, 23
Allocate, 33, 57, 58, 59
Allocation, 4

B-tree indexing, 35
B-trees, 36
Balanced Trees, 37
Beep, 14
Btree, passim
Btree.Add, 15
Btree.DEF, 2, 3, 31
Btree.Delete, 15
Btree.Find, 15
Btree.MOD, 39
Btree.Next, 15, 19
Btree.Open, 11
Btree.Prev, 15
Btree.Reset, 15, 19
Btree.Search, 15

Change, 52

Clear, 20, 21
ClearIndex, 56
CLIENT, 19, 21
Client, 23
CLIENT.DAT, 18, 19
CLIENT.DEF, 17, 18, 19
CLIENT.MOD, 17, 19
Client.Template, 20
Close, 20, 48, 49
Club, 17, 19, 22
Club program, 19
CLUB.MOD, 17, 19
CmpRes, 44
CNum, 18
Compare, 11, 13, 23
CompareFunction, 44
CompFct, 47
Copy, 11
CountProc, 61
Create, 13, 30, 45, 46

DataBack, 23
DataFore, 23
DataLoc, 50, 51, 53
DataLocked, 33
Deallocate, 33, 59
Deallocation, 4, 12, 44, 59
DelayCount, 30, 61
Delete, 51
DeleteIndex, 51
DeleteName, 15
Display, 15, 21
DisplayBack, 23
DisplayBorder, 23
DisplayFore, 23

- DupKey, 48
- DupKeys, 13
- Error handling, 4
- ErrorHandler, 45
- ErrorProc, 45
- Errors, 4, 45
- FHandle, 45, 46, 49
- File access
 - shared, 26
- File sharing, 26, 31, 56
- FileKey, 21
- Files
 - Non-shared, 30, 43
- Find, 15, 51, 52
- FindIndex, 52, 57
- FindName, 15
- FIO, 61, 62
- FIO.Open, 27
- FIOx, 25 - 30, 41, 60, 61
- FIOx.DEF, 2
- FIOx.DOS, 26
- FIOx.MOD, 26
- FIOx.Multi, 26, 30
- FIOx.OS2, 26
- FixedSize, 46, 47
- FixSize, 12, 44
- Flush, 49, 63
- FreeHandle, 49
- GetDate, 64
- GetMenu, 14, 23
- GetTime, 64
- GLOBALS, 21, 22, 23
- GLOBALS.DEF, 17
- GLOBALS.MOD, 17
- Help, 19
- IHandle, 45,
- Index synchronization, 55
- Indexing, 36
- Indexing
 - B-tree, 35
- Input, 21
- InputStr, 64
- INum, 18
- Item, 23
- ITEMS, 19, 21
- ITEMS.DAT, 18
- ITEMS.DEF, 17, 18, 19
- ITEMS.MOD, 17, 19
- KeyFct, 48
- KeyFunction, 44, 45
- KeySize, 48
- LastErr, 33
- LastError, 4, 50
- LastFError, 4, 50
- Length, 50, 52, 58, 59, 60
- Lock, 28, 29, 33, 57, 58, 61, 62
- LockIHandle, 31, 33, 58
- LockQSize, 45
- LockRange, 28, 29, 62
- LockRec, 26, 27, 29, 61, 62
- LRUCount, 39
- MAKEKEY, 11
- MakeKey, 21
- MaxIHandle, 46
- MaxKeySize, 45
- MaxLen, 64
- Misc, 41, 63
- Mode, 46
- Multi, 61, 62
- Name, 46
- NameFile, 12
- New, 46, 47, 48
- Next, 33, 51, 54, 55
- NextIndex, 54, 57
- NIL, 46, 49
- NoDealloc, 12, 44
- NoDealloc, 46
- Non-shared files, 43
- NotFHandle, 50
- NotIHandle, 50
- Null, 45, 47, 48, 49, 56

NULLPROC, 48

OKChars, 14

Open, 11, 19, 27, 45, 47, 48, 62

OpenClientFiles, 19

OpenData, 12, 19, 45, 46, 47

OpenIndex, 13, 19, 45, 47, 48

OpenItemFiles, 19

Pages, 36, 38

PageSize, 39

Position, 59, 60

Prev, 51, 54, 55

PrevIndex, 54, 57

Procedure, 61

Prompt, 14

PromptBack, 23

PromptFore, 23

Read, 33, 59

ReadOnly, 27, 30, 46

RecordSize, 47

Release, 33, 56

Reset, 55

RetryCount, 30, 61

SALES, 19, 21, 22

Sales.Close, 22

SALES.DEF, 17, 18, 19

SALES.MOD, 17, 19

Sales.Template, 20, 22

Search, 15, 21, 51, 53

SearchIndex, 53, 57

SetSyncMode, 55

Share, 30

SHARE.EXE, 25, 26, 61

Shared, 27, 46

Shared file access, 26

Shared files, 30

SHARED.DAT, 27

ShowNames, 15

Size16, 12, 44, 46

Size32, 12, 44, 46

Slot, 46, 47, 48

Store.Close, 20

StoreName, 23

Str, 11

Summary, 22

Synchronization

Index, 55

TaxRate, 23

Template, 15, 20

Trees

Balanced, 37

UnLock, 33, 57, 58, 62

UnLockIHandle, 31, 33, 58

UnLockRange, 28, 62

UpdateTitle, 23

Window, 64

Window.GotoXY, 14

Work, 22

Write, 33, 60

WriteAt, 14

x, 62



Jensen & Partners UK Ltd.
63 Clerkenwell Road
London EC1M 5NP
England

Jensen & Partners International, Inc.
1101 San Antonio Road, Suite 301
Mountain View, CA 94043
USA