E-Tree Plus

By EllTech Development, Inc.

4374 Shallowford Industrial Parkway
Marietta, GA 30066

Technical Support:	(404) 928-8960 9am - 5pm M-F TONY
Sales:	(800) 553-1327
BBS:	(404) 928-7111 (HST)

FAX: 404-924-2807

Documentation Revision Date: 10/91

Section 1
Quick Start

Although we recommend that all E-Tree Plus users review this entire manual (except for Section 7) before attempting to use the product, it is not entirely necessary for those who have experience with Btrieve or PDS ISAM. If you fall into this category and would like to "jump right in," we suggest that you follow the guidelines set forth below:

- Review Section 2, "Introduction" and Section 3, "Getting Started" for an overview of the documentation organization and installation instructions.
- If you will be designing a network database program where more than one user can simultaneously access your database files, please review Section 4(D), " Introduction to File and Record Locking" for an overview of these fundamental and essential network programming concepts.
- And finally, load and run one or more of the example programs. They are well commented and designed to help you see how the E-Tree Plus routines are used in your BASIC programs. You will gain the most from this exercise if you run them in the "step-trace" mode available in your compiler's interactive development environment ("IDE"). Refer to the appendix in this manual that applies to your compiler for specific instructions on loading and running the example programs.
- If you are familiar with Btrieve, Appendix A, "Differences Between E-Tree, PDS ISAM, and Btrieve" will be helpful to you. This section contains a table which describes each of the Btrieve functions and the E-Tree equivalents.
- If you are familiar with PDS ISAM, Appendix E, "Converting PDS ISAM Code to E-Tree" will be of particular interest to you.

All of the routines available in this product are categorized and listed alphabetically in Section 6, "Routine Reference." For detailed information about the routines used in the demonstration programs, please refer to section 7(A), "User-Level Routines."

Section 2
Introduction

2 (A) Product Description

Welcome to the E-Tree Plus Library! We have put every effort into making this one of the finest database management products available to BASIC programmers.

E-Tree Plus was designed to function on single-user and multi-user "networking" platforms. Although there are numerous complexities involved in securely sharing data files on a network, most of the hard work (i.e. low-level record locking) is handled for you automatically by E-Tree Plus. You need only to specify the *type* of lock (if any) you wish to place on a record that is being inserted, updated, or retrieved. More information about specific types of record locks can be found in Section 4 (D), "Introduction to File and Record Locking." Additionally, if you write your program for a network environment, it will also function properly on a single-user system with no additional work required by you. E-Tree Plus automatically detects the presence (or lack) of a Novell or Microsoft compatible network and handles file and record locking requests appropriately.

E-Tree Plus's memory, file and record locking routines automatically detect potentially dangerous situations, such a exiting your program without unlocking all records and handle it for you automatically. Upon program termination, locks are released, all files are closed and all memory in use by E-Tree routines is released.

2 (B) Compatible Compilers

E-Tree Plus is compatible with Microsoft QuickBASIC versions 4.00b and 4.50, Microsoft BASIC Compiler 6.x and Microsoft BASIC Professional Development System 7.x (AKA "PDS").

2 (C) System Requirements

During software development, E-Tree Plus requires one of the compilers mentioned in Section 2(B), a hard disk with at least 1.5 megabytes of available space, and DOS version 2.1 or later. At runtime, the E-Tree Plus software requires DOS 2.1 or later on single-user platforms or DOS 3.1 or later on multi-user or networking platforms.

2 (D) Product Registration

If E-Tree Plus was purchased directly from EllTech Development, it has already been registered to the person who bought it. If it was purchased through a dealer, a product registration card can be found in the diskette envelope. Please fill it out and return it to us immediately. We will be able to provide technical support only to registered users.

2 (E) Technical Support

EllTech Development provides free, full-time technical support to all registered users of E-Tree Plus. Our hours are Monday thru Friday, 9:00am to 5:00pm Eastern time. You can reach us at (404) 928-8960.

Or if you prefer, call our 24 hour bulletin board system ("BBS"). We run PCBoard BBS software and a U.S. Robotics Courier HST modem (supporting baud rates from 1200 to 38400). We have a dedicated message base for E-Tree Plus as well as the latest version of the product available for download. Many times you can get an answer to your tech support question by calling our BBS and scanning messages. If you've run into a snag, the chances are pretty good that others have had a similar problem and a solution already awaits you.

As a registered user of E-Tree Plus, you already have an account established on the BBS as well as access to the private E-Tree conference. Log on using the name that appears on your invoice (no middle initials). Your password is your Zip or Postal code. Be sure to use the BBS's "W" command to change your password (for security reasons) during your first session.

Here are some phone numbers you'll need to know:

- (404) 928-8960 Technical Support
- (404) 928-7111 EllTech Development's BBS
- (404) 924-3351 Fax

2 (F) Memory Requirements

Depending on various factors, E-Tree Plus can add between 60K and 100K to the size of your .EXE file or extended runtime library (RTM). When you open the first E-Tree database file, memory is allocated for file I/O buffers and for an internal "scratch pad." The amount of memory allocated is dependent on the file page size (discussed later in this manual) and the record length (between 2K and 128K). Part of this memory will be allocated from expanded memory (EMS) if an expanded memory manager is installed in the system and if sufficient EMS memory is available.

Section 3
Getting Started

3 (A) Making Backup Copies

Before installing E-Tree Plus for the first time, be sure to make a backup copy of the distribution diskette(s) for safe keeping. The DOS "DISKCOPY" command is best suited for this purpose. If you require any help using DISKCOPY, please refer to your DOS reference manual.

3 (B) Installation

To install E-Tree Plus on your hard disk drive, insert the distribution diskette into the appropriate floppy disk drive and close the latch. Log onto that floppy drive by typing the drive letter followed by a colon and press <Enter>. Next, you'll need to run the "INSTALL.BAT" program. This will decompress and copy the appropriate files to the specified subdirectory on your hard drive. Once that has been accomplished, the appropriate libraries will be constructed for you automatically. Be sure that the programs BC.EXE, LINK.EXE and LIB.EXE are in your DOS PATH. Also, the drive and directory where your BASIC support libraries are located must be indicated by the "LIB" environment variable. To set the LIB environment variable, type the following at the DOS prompt, or add it to your AUTOEXEC.BAT file:

```
SET LIB = C:\BASIC\LIB;C:\ETREE;
```

The above example lists two directories where the LINK program will search automatically for .OBJ and .LIB files. Each drive and directory entry is delimited with a semicolon ";".

The syntax for INSTALL is as follows:

```
INSTALL Drive:\Directory [Switch]
```

Where :

Drive:\ is the letter of the hard drive where you wish to install E-Tree.

Directory is the name of the subdirectory in which you wish to install the E-Tree files.

Switch is a command line switch that indicates the compiler that you'll be using with E-Tree. The default (no switch) is Microsoft PDS. The other switch is:

/Q for QuickBASIC 4.00b, 4.50, and Microsoft BASIC 6.x

For example, to install E-Tree for QuickBASIC 4.5 in C:\ETREE, you would type in the following at the DOS prompt:

```
INSTALL C:\ETREE /Q
```

To install E-Tree for Microsoft PDS, you would type the following:

```
INSTALL C:\ETREE
```

To get a summary of the above information, type "INSTALL" without any command line arguments.

Section 4
Using E-Tree Plus

4 (A) General Information

Every possible effort has been made to ensure that the routines provided in the E-Tree Plus Library are as easy and intuitive to use as possible. In designing any library of routines there are the inevitable trade-offs between simplicity and flexibility, and between functionality and data security. More often than not we will err toward flexibility and data security and will take data security over flexibility if necessary.

Much of the syntax and the implementations of the routines provided are driven by the need to assume the product will be used to manipulate databases in networked environments. The impact of these assumptions on the users who will be working strictly in non-networked situations has been mitigated as much as possible, but is unavoidable.

At every step in every function the assumption must be made, by us and by you when you design your programs, that access to each piece of data stored in a file is desired by more than one process at any given time. This assumption increases the quantity of code required for error avoidance, error detection, and error recovery by at least an order of magnitude over similar code intended for a strictly single-user application. It also forces several additional arguments on many of the functions in order to provide flexible but secure record locking mechanisms. For the vast majority of programs this added complexity is hidden in the lower-level functions. Using the E-Tree Plus Library, you can create fully functional, multi-user, database applications without ever explicitly setting or releasing a data-record lock.

4 (B) Procedure Organization

All the routines in the E-Tree Plus Library are grouped according to how likely it is that you will need to use them directly. The routines you will use most often are called "User-Level routines." The ones you will have little need to use directly are call "Low-Level routines." In between there are "High-Level" and "Middle-Level" routines which you might find convenient to use in special circumstances, but which you should avoid if you can. We have attempted to keep much of the call-syntax complexity in the lower level routines and to keep the most frequently used routines as simple to use as possible. The higher level routines are much safer for you to use because they perform many more checks on the correctness of the arguments you have supplied and on whether the current state of the database is appropriate for the function you are requesting. By providing full source code and documentation for all of the routines used in the library, we give you the choice of using the low-level routines if you need them. However, most applications needing database management tools can be written using only the simple, User-Level routines.

A breakdown of the various E-Tree routines and the source module in which they can be found is in Section 6(E).

User-Level routines are those that will be used the most in the normal database manipulations. These include routines to Create, Open, Read, Write, and Search the database , and routines to move from one record to another.

High-Level routines are those that report on the state of a database and provide access to the data and file structures used to maintain a database. In general the data structures they access and report on should not be manipulated directly.

Middle-Level routines are usually called by the User-Level and High-Level routines in support of their work. These are routines that would be used directly only in special cases where a more direct access to the database structures is needed. You will not need to use these routines for most programs, but they are here if you need them.

Low-Level routines provide direct, and usually dangerous access to the data and files structures of the database. Use of these routines is not recommended, but they are here for those who like to live on the edge.

There are also numerous general purpose routines included and fully documented that are not specific to the E-Tree Plus Library. These include file I/O routines, memory management routines, network access routines, and miscellaneous support routines. These routines are used by the E-Tree Plus Library and should not be modified. They are also available for use in any programs you write.

In general you should use the highest level function that accomplishes the task you need. The higher level functions will provide more data validation and error avoidance than the lower level functions. The extra protection is left at the higher levels to reduce redundant error checking, minimize the code size, and increase performance. For example, to read a data record you could get the address of the current record, convert it to the physical location in the file, and read the data record directly; however, using EtRetrieve is much safer, probably faster, and certainly easier.

4 (C) Glossary of Terms

Here are some words and phrases that you will be seeing a lot of in this manual along with explanations of what we mean when we use them. We have tried to keep with the common usage of all our buzzwords and catch phrases, and when we haven't we have at least tried to be consistent.

Data Dictionary

Contains the all of the information that describes the database and index structures. This includes the record definition and the index key definitions.

Database

A single file that contains data, indexes, and other information.

Field

The smallest unit of data the database can understand. Each record contains one or more fields. PDS ISAM uses the term "column" to describe a field.

Fixed-Length Data

Part of the data record made up of zero or more fields that always have a defined, unchanging length and are stored together in the database. If the record is defined as having only a portion, the fixed-length portion will have zero data fields, but will still exist to allow management of the data.

Hard-Lock

A lock placed using the operating system. This type of lock denies all access to the file in whole or in part.

Index Key Definition

A definition of how to sort the data stored in the records. Each index has one key definition.

Index Key

The result of applying the Index Key Definition to a particular data record to get information used to sort the data.

Index

A part of the database that is used to allow the data to be accessed in a defined, sorted order. There can be several indexes for a database.

Lock

A control placed on a file or part of a file that limits access to the data by other programs.

Record Definition

A definition of each field in the data record that describes the type of data in the field, the location of the field in the record, the length of the field, and the name of the field.

Record

A unit of data stored in the database. The database is made of records; records are made of fields.

Segments

All or part of one or more fields which are combined to from a single index.

Semaphore

A signal (like semaphore flags) usually used in E-Tree Plus to indicate a soft-lock.

Soft-Lock

A lock placed using a semaphore. This type lock is a marker in the file itself that signals other programs that the file or part of the file is currently read-only and should not be modified. This type of lock actually allows other programs to read or write the file. All programs accessing files that use soft-locks should check and honor the soft-lock.

Data

Part of the data record that can vary in length.

4 (D) Introduction to File and Record Locking

Creating applications that will be used in a network environment means having to share resources with other applications. Luckily, sharing hardware resources, like printers and physical access to disk drives, is usually handled by the network operating system. Sharing file space is usually handled by a cooperative effort between the operating system and the applications that need access to the files.

The operating system provides methods of controlling access to all or parts of a file through file and record locking mechanisms. Applications that need to restrict access to a file use the locking functions provided by the operating system to gain access to a file and to limit other applications' access to the file.

Applications can also control access to files by following a predefined set of rules (called a protocol) that strictly defines the conditions under which it is safe to read or write to portion of a file. Using a method other than the functions provided by the operating system is usually required if the types of access control provided by the operating system are incomplete or do not provide as much flexibility as you want.

There are three types of access that are important for sharing database files between applications on a network:

- Shared (Deny None) - no limit is placed on when or who reads the file or file region.
- Read-Only (Deny Write) - other applications can read from, but cannot write to, the file or file region.
- Exclusive (Deny All) - other applications are denied all access to the file or file region.

The E-Tree Plus Library uses a combination of operating system functions and application protocols to provide all three types of file access control. We have tried to hide as much of the complexity of the file access as possible. However, it is impossible for us to know exactly what amount of control you will need in every situation. If it were possible for us to determine your every intention when you write your programs, we would hide all the locking actions away so you would not have to bother with it (we would also be making lots of money with our mind reading program).

It turns out that there are only four locking conditions that are needed to control access to data records in most applications:

- No locking needed for the current operation
- Lock during the current operation, unlock when it is done
- Lock during the current operation and leave locked for further processing
- Lock is already in place from previous work, unlock when the current operation is completed

Only two of these four conditions place locks on parts of the file. We need variations on those to determine the action to take if the lock fails (because someone else already has that part of the file locked). In any operation where a lock is placed, a lock collision is possible. A lock collision occurs when one application locks part of a file and then a second application attempts to lock the same region of the file, too. The lock for the second application will fail.

The second application will want to either retry the lock for a short time to see if it is released or it will want to simply wait until the lock is released regardless of how long it takes (both cases must allow user intervention, but that's a side issue right now).

Examples of whether to use a timeout delay or to retry the lock without a time limit are:

- When reading a data record just to display it, and

- When reading a data record during the processing of a report.

In the first case, you do not want the program to be delayed so long that the user thinks there is a problem and you do not want to keep the user from doing useful work just because one record is locked. If the particular record they want to view is locked, their time will be better spent working on another record until that one is free. When a report is running, however, you probably want the program to simply wait for a locked record to be released and then continue with the report instead of quitting every time it finds a record locked.

Adding these timeout considerations to the list brings the number of possible locking conditions to six:

- No locking needed for the current operation
- Lock during the current operation, unlock when it is done -- lock attempt times out after X seconds
- Lock during the current operation and leave locked for further processing -- lock attempt times out after X seconds
- Lock during the current operation, unlock when it is done -- lock attempt will wait until record is available
- Lock during the current operation and leave locked for further processing -- lock attempt will wait until record is available
- Lock is already in place from previous work, unlock when the current operation is completed

The E-Tree Plus routines EtInsert, EtUpdate and EtRetrieve require you to specify a lock type using a separate parameter in the CALL syntax to indicate the desired lock type. This "LockFlag%" parameter can have a value ranging from 0 to 5 to indicate:

0) No locks needed
1) Lock to access record, unlock when finished (~~system~~ Inf timeout)
2) Lock to access record, leave locked (~~system~~ Inf timeout)
3) Lock to access record, unlock when finished (~~infinite~~ sys. timeout)
4) Lock to access record, leave locked (~~infinite~~ sys. timeout)
5) Assume the record is locked, unlock it when finished

All E-Tree Plus locks can be placed and released without requiring an explicit call to a lock or unlock routine. For instance, a lock can be placed when EtRetrieve() is used to read a record (LockFlag% = 2 or 4) and then released with EtUpdate() after the record is edited (LockFlag% = 5).

We chose this method of defining the lock actions because it provides a consistent method regardless of whether the lock is a hard lock or a soft lock, and because it allows you to place and release locks more efficiently. Most of the locks placed on parts of a file will be on the data records. Locking and unlocking a data record require the page the data is on be read (or at least part of it). Putting the work of locking and unlocking inside routines that are already accessing the data reduces the number of file accesses required.

Along with knowing what locking conditions are likely, you need to know how to use them in your applications. There are a few simple guidelines to follow in your applications:

- Always lock a record before reading or writing it to be certain you are allowed access to it.
- Don't keep records locked any longer than necessary.
- Any lock placed must be released

Examples of Using Locks

Reading data just to display it is the simplest operation in any application in terms of the locking requirements. All that is required is for the data to be available (not hard-locked by someone else). To make sure you can access the data, you must lock it before you read. Since you are not going to be making any changes to the data, you don't want to leave it locked. This is probably a time when you want fairly quick response. If there is a lock collision, the program should not sit and wait forever for it to be resolved. Use LockFlag% = 1 to Lock/Unlock the record using the system timeout.

```
LockFlag% = 1
Call EtRetrieve( ... LockFlag% ... )
```

Processing a report on many records in the database is the second easiest lock situation. It is almost exactly like the previous one except you don't want the report to be interrupted because of a lock collision. Use LockFlag% = 3 to Lock / UnLock the record using an infinite timeout.

```
LockFlag% = 3
Call EtRetrieve( ... LockFlag% ... )
```

Reading a record to edit and update is a little trickier because you definitely do not want to stop editing the record after you start and you don't want to be editing the record if someone else is already editing it. You want to lock the record from the time you read until you update it. In this case the lock attempt made by EtRetrieve() will fail if someone else has the record locked already (returning an error code of LockedBySemaphore). If EtRetrieve() succeeds then it is safe to edit and update the record.

```
'Lock/Leave Locked (with system timeout)
LockFlag% = 2
Call EtRetrieve( ... LockFlag%, Status% )

IF Status% = 0 THEN
   'Edit the record

   'The record is already locked, unlock it when update is
   ' finished
   LockFlag% = 5
   Call EtUpdate( ... LockFlag% ... )

   'If you should decide to abort the update, you will still need
   ' to unlock the current record. The routine
   ' EtUnlockCurrentRecord is used for this purpose.
   Call EtUnlockCurrentRecord( ... )
END IF
```

Lock Collisions

A lock collision occurs when one user attempts to lock a record already locked by another user. This usually occurs on a retrieve operation, but can also occur during an update (if you didn't lock the record when retrieving it) or a delete.

When you request a lock to be placed on a record during a retrieve or update operation, you can instruct the E-Tree routines exactly how to react should a lock collision occur: Retry the lock for the amount of time specified by the current "SystemTimeout" value (LockTypes 3 and 4), or retry the lock for an infinite period of time (LockTypes 1 and 2).

The default SystemTimeout value is 10 seconds. If a LockType of 3 or 4 is used and the lock request cannot be satisfied within this period of time, the Retrieve or Update operation will fail with a LockedBySema error (indicating another workstation already has the record locked). You can change the

SystemTimeout value using the EtSetLockTimeout routine, or fetch its current setting using the EtGetLockTimeout routine.

When a lock collision occurs, sometimes it would be desirable to display a message to your user indicating that someone else has the record locked, and then prompt "<R>etry or <A>bort?" To handle this scenario, you can check the Status% variable after each Update and Retrieve operation and branch to a special part of your program designed to display the message and to wait for an answer to the prompt. Although this approach isn't very difficult to implement, it can get a bit tedious. For this reason, we've devised an alternative that allows you to handle lock collisions more generically, without the need for custom code following every Retrieve and Update operation.

In the module ETLOCK.SUB, there are two functions:

- EtLockTimeoutMsg% - Invoked by E-Tree's internal routines when a LockType 3 or 4 (system timeout) fails. In other words, it's invoked when a lock was not successfully placed in the current SystemTimeout period. If a value of -1 (True) is returned by this function, the lock attempt is retried. If a value of 0 (False) is returned, the lock attempt is not retried and control is returned to your program with the Status% reflecting a LockBySema error. By default this function returns a 0 (False).

- EtLockNoTimeoutMsg% - Invoked by E-Tree's internal routines when a LockType 1 or 2 (infinite timeout) fails. By default, this function returns a value of -1 (True) indicating that the lock should be retried (retry continuously).

These routines provide you with a centralized location to insert your own, custom lock collision code. Remember, if you add code to these routines within the ETLOCK.SUB file, you must recompile the E-Tree source files and rebuild your libraries before the changes will take effect. The BLDLIB.BAT file can accomplish this for you in one easy step.

4 (E) Data Types Supported By E-Tree Plus

The following list defines the data types that are understood by E-Tree Plus and the named constant for each (if defined). These are the only data type values that will be correctly evaluated. The named constants used represent the numeric values currently assigned to designate data types. All type numbers above the ones listed are reserved for future additions. The constants are defined in ETCONST.BI.

Type	Named Constant	Length
Variable Length Data	EtVariable	UserDefined
Signed Binary Integer	EtSigned	UserDefined
Unsigned Binary Integer	EtUnSigned	UserDefined
Auto Increment	EtAuto	4 bytes (Long Int)
Integer	EtInteger	2 bytes
Long Integer	EtLong	4 bytes
IEEE Single Precision	EtSingle	4 bytes
IEEE Double Precision	EtDouble	8 bytes
IEEE Extended Precision	EtIEEEx	10 bytes
PB Fixed Point BCD	EtPBBCDFixed	8 bytes
PB Floating Point BCD	EtPBBCDFloat	10 bytes
MBF Single Precision	EtMBFs	4 bytes
MBF Double Precision	EtMBFd	8 bytes
PDS Currency	EtCurrency	8 bytes
ASCII Case Insensitive	EtString	UserDefined
ASCII Case Sensitive	EtStringCS	UserDefined

Notes:

- The lengths for data types with 'user defined' lengths are defined when the index key or data field is defined.

- The signed and unsigned-integer types can be any length. It is assumed that the least significant byte is stored in the first position in the field or key and that the bytes continue in order of significance to the most significant in the last byte of the field. This follows the pattern set by Intel in storing word and double-word integer values in memory. Unsigned-integers are also assumed to be stored in 2's complement form.

- Autoincrement fields are signed long integers.

4 (F) Putting E-Tree Plus to Work

Using the E-Tree Plus routines is very straight forward. For most applications there are four main areas of interest:

- Creating a new database
- Creating indexes
- Inserting, updating, and deleting records

- Moving around the database

1. Creating a Database

The EtCreate() routine is used to create a new, empty database. The call to EtCreate() establishes the record structure for the database and initializes the file structure. It also opens the database and initializes data structures required to manage the database. At this point the database does not have any data records or index structures defined.

An existing E-Tree database must be opened for access using the EtOpen() routine. Once the database is opened, either as part of EtCreate() or using EtOpen(), it can be accessed and manipulated by the other E-Tree routines.

The EtFileExist routine can be used to determine if a specific file already exists. This will help you to determine if you need to create the database or simply open it.

2. Creating Indexes

Keeping data in a defined, sorted order is the main reason for using an ISAM-type database such as E-Tree Plus. Before E-Tree Plus can order the data records, you must define the structure of the key for each index that will be used. The routines EtCreateIndex and EtCreateIndex2 are used to store a key description in the database and initialize the index, including adding all existing data records to the index. The EtCreateIndex routine is used to define keys that consist of one or more complete fields (like PDS ISAM's "combined indexes"). EtCreateIndex2 is used to define keys that can consist of smaller pieces or "segments" of fields instead of entire fields. EtCreateIndex is easier to use, but not as flexible as EtCreateIndex2.

Using EtCreateIndex, you define the key description by passing a string array containing the names of the fields that will comprise this index in the order in which they should be sorted. In other words, each field defines a segment for the key definition.

Using EtCreateIndex2, you must specifically define each segment that will describe the index. This means that for each key segment you must provide the offset, length, data type, sort direction, and a modify-protect flag. Each segment can be made of any part of the data record; this includes complete fields or pieces of one or more fields. As with EtCreateIndex, you pass an array with each element describing a segment of the key.

The only limit on the number of indexes or number of segments to a single index that you can define is that all the key definitions must fit on a single page in the database. The page size defined when EtCreate() originally created the database limits the number of index segments

E-Tree indexes can be added or deleted at any time using EtCreateIndex() and EtDeleteIndex(). Adding and deleting indexes requires a considerable effort in terms of both time and file manipulation. If an index will be used frequently it should be created very early in the life of the database (say, just after EtCreate) and then left to be maintained incrementally. If an index is used only for a special purpose and then only infrequently, it should be added when needed and deleted when not needed. This will save file space and time when inserting data records (since every index must be updated after an insertion).

3. Inserting, Updating, and Deleting Records

New data records are added to the data base using the EtInsert() routine. This routine stores the new data in the database and updates all defined indexes to reflect the addition.

Existing data records are read from the database by making the desired record the current record using one of the EtSeekXX (searches the index for the specified key value) or EtMoveXX (moves the current record pointer) functions and then using EtRetrieve to read it from the file.

You can modify or "update" the current record using the EtUpdate routine. EtUpdate() replaces the old data with new data and updates all the defined indexes.

You can delete the current record by using the EtDelete routine. EtDelete removes the data record from all defined indexes and releases the record's disk space for use by future insertions.

4. Navigating Through the Database

The EtSetIndex routine is used to specify which of the databases defined indexes to use when attempted to locate a specific record or group of records. To access data by a defined index , use EtSetIndex() to make the desired index "current." All EtSeekXX functions use the current index to find their way through the index to access data. The 'Null' index is defined as the raw order of the data in the database.

If the Null index is the current index, either an index is not defined, EtSetIndex has not been used, or EtSetIndex has been used to make the Null index current. The Null index can only be traversed using the EtMoveXX functions. The EtSeekXX functions will not work on the Null index.

Once a true index has be selected (other than 'Null'), the EtSeekXX functions can be used to find a record based on a target key value or partial key value. There are five different flavors of EtSeek. Each search the current index for the *first* entry that is EQual to (EtSeekEQ), Greater Than (EtSeekGT), Greater than or Equal to (EtSeekGE), less than (EtSeekLT), or less then or equal to (EtSeekLE) the key value you provide. If an entry is not found, the EtEOF function will return -1 (TRUE). If an entry is found, EtEOF returns 0 (FALSE) and the current record is set to the first record matching your seek criteria. The following pseudocode describes the procedure for locating a specific record:

```
EtSetIndex Handle, "IndexName", Status
EtSeekEQ Handle, KeyValue, Status
IF NOT EtEOF(Handle%)
  EtRetrieve Handle, RecordBuffer, LockType, Status
  PRINT RecordBuffer
ELSE
  PRINT "Record not found."
END IF
```

If you want to retrieve a group of records matching your seek criteria, EtSeekXX locates the first record and EtMoveNext moves to the next and subsequent records. The EtEOF flag will be set when you attempt to EtMoveNext past the end of the index. The following pseudocode demonstrates how to locate and display all records matching your seek criteria:

```
EtSetIndex Handle, "IndexName", Status
EtSeekEQ Handle, KeyValue, Status
DO UNTIL EtEOF(Handle)
  EtRetrieve Handle, RecordBuffer, LockType, Status
  IF RecordBuffer.KeyField = KeyValue THEN
      PRINT RecordBuffer
  ELSE                         'If the key values don't match, we've
      EXIT DO                  ' moved past the last one we're interested
  END IF                       ' in, so exit the loop.
  EtMoveNext Handle, Status
LOOP
```

Once any record has been made current, EtMoveNext() and EtMovePrevious() can be used to move to records in the order defined by the current index. EtMoveFirst and EtMoveLast are used to move to the first and last record in the current index (respectively).

5. Pseudocode Examples

The following is a short pseudocode example of the steps used to create a database with an index, and add a record to it. This is a descriptive example without the complete argument lists for the routines used. For the full argument lists refer to the appendix.

```
'Define the record structure and create the database
Call EtCreate( ... )

'Create an index
Call EtCreateIndex( ... )

'Insert a new data record
Call EtInsert( ... )

'Close the database
Call EtClose( ... )
```

The following is a short pseudocode example of the steps used to open an existing database and read all the records in the order defined by an existing index.

```
'Open it
Call EtOpen( ... )

'Select the index to use and go to the first record
Call EtSetIndex( ... )

Do Until EtEOF( ... )
   'read it
   Call EtRetrieve( ... )

   'do something

   'next
   Call EtMoveNext( ... )

Loop

'Close the database when done
Call EtClose( .. )
```

4 (G) Variable-length Fields

The may be occasions where you need to store a piece of variable-length data along with your fixed-length data. For example, the fixed-length portion of your record can contain "header" information and the variable-length field can be used for memo fields, invoice line-item entries, binary data such as graphics images, etc. In many cases a variable-length field can eliminate the need for an additional sequential file for such data.

The following are some things to keep in mind when you are thinking about using variable-length fields:

- You can define *one* variable-length field per record and it must be the *last* field defined (EtCreate is used to define your record structure and to create an empty database file).

- Technically, the maximum length of an E-Tree Plus variable-length field is limited only by available disk space. However, because of the 32K limit on length BASIC places on variable-length strings, the E-Tree routines EtRetrieveVariable and EtUpdateVariable must also limit variable-length fields to 32K.

- The high-level routines EtRetrieveVariable2 and EtUpdateVariable2 are used if you want to specify your own memory buffer area (by segment and offset addresses) instead using a BASIC variable-length string as the buffer. Using these routines, the maximum length of a variable-length field can be up to 64K in size.

- Optimum performance will be achieved if you limit the length of your variable-length data to the page size of the file (which is defined during the file creation).

- The EtRetrieve, EtUpdate and EtInsert routines are used to manipulate the fixed-length portion of your record. The EtRetrieveVariable and EtUpdateVariable routines are used to manipulate the variable-length field. In other words, if you wish to insert a new record containing fixed and variable-length data, two steps are required; EtInsert and EtUpdateVariable. The same applies to retrieving data: EtRetrieve retrieves the fixed-length data and EtRetrieveVariable retrieves the variable-length data.

- The EtUpdateVariable routine is used to insert new variable-length data for a record as well as to update data in an existing variable-length field. Updating a variable-length field completely replaces the old data with the new.

- When retrieving record data, you can retrieve just the fixed-length portion, just the variable-length portion or you can retrieve both. The same applies to updating a record containing variable-length data.

4 (H) Opening More than Four E-Tree Files Simultaneously

By default, you can open a maximum of four E-Tree database files simultaneously. For each open database, we must allocate around 1K of BASIC data space (mostly far memory, but some in DGroup). We felt a default value of four open files would meet the needs of most users while not requiring a great deal of overhead. If your needs exceed this default limitation, there is a simple solution.

Before calling EtCreate or EtOpen for the first time, invoke the EtInitManager function. With this procedure, you can specify the desired number simultaneously open files. For example

```
REM $INCLUDE:'ETREE.BI'
MaxFiles% = 8
Result% = EtInitManager%(MaxFiles%)
IF Result% THEN
   PRINT "Error"; Result%;" initializing record manager!"
   END
END IF
'At this point, you can now open up to 8 E-Tree files at a time
```

4 (I) E-Tree Database Version ID

Embedded in the header of each E-Tree database file is a "file ID" string. By default, it is defined as "EllTech Development Corp. E-Tree ISAM Version 1.00". Some of our customers feel uncomfortable with another company's name embedded within their database files. If you would like to change this ID, you have a couple of options:

- Load the ETREEHI.SUB file into a text editor. It is located in the "SUB\" directory off of your default ETREE subdirectory. In this file, search for "EtFileStructureID$ =". There you will find the ID definition. Change it to whatever you want. It must be 64 characters in length or less. Save the file and run BLDLIB.BAT to recompile E-Tree's BASIC source files and to rebuild libraries.

- If you don't want to modify E-Tree's source code file, you can redefine the ID string within your program. Before you call EtCreate or EtOpen for the first time, do the following:

```
REM $INCLUDE:'ETREE.BI'
Result% = EtInitManagerDefault%  'Or EtInitManager
IF Result% THEN
   PRINT "Error";Result%;"initializing E-Tree's record manager!"
   END
END IF
EtFileStructureID$ = "New ID data goes here"
```

That's it! Just make sure that the new ID is 64 characters in length or less.

4 (J) Error Codes Generated by E-Tree Routines

All routines that return a Status% code, either as an argument to the routine or as the value of a function, will return a zero to indicate that the routine completed successfully or a non-zero value to indicate an error. All Status% codes greater than zero correspond to DOS errors. DOS non-critical errors are returned as the error value from DOS plus 100. DOS critical errors are returned as the actual value DOS returns. E-Tree Library errors are returned as negative values. In addition, some functions have specific negative Status% codes that relate only to the specific function.

See Appendix C for a complete list of E-Tree, DOS and EMS error codes.

Section 5
Manipulating Databases With EUTIL.EXE

EUTIL.EXE is E-Tree's external database maintenance utility. It provides the following functions:

- Converts existing Btrieve, dbase, PDS ISAM, and comma-delimited ASCII files to E-Tree's file format.
- Allows you to modify an existing database's record structure (add fields and delete fields).
- Exports data (entire records or specific fields) from an E-Tree file to a comma-delimted file.
- Rebuilds an E-Tree database in the event of data record or index corruption.
- Packs the database to remove unused data pages (should only be necessary should you delete several hundred records at a time).
- Resets the semaphores in the database should a network workstation lockup or reboot while having records locked.

You can distribute EUTIL with your application programs free of royalties and restrictions.

Because we are enhancing and adding features to EUTIL on almost a weekly basis, we chose not to document it in this revision of the manual. In your default ETREE directory, you'll find a file called EUTIL.DOC. It's an ASCII text file describing the EUTIL command-line syntax and it's newest features. EUTIL updates are available free of charge from EllTech's BBS at (404) 928-7111. If you don't have a modem, we'll be happy to send you an update disk for a nominal charge of $10. Just give our technical support department a call at (404) 928-8960.

Section 6
Routine Reference

6 (A) User-Level Routines

The following routines are the User Level routines. For most applications these are the only routines you will need to use to create and maintain your databases.

EtBOF
EtClose
EtCloseAll
EtCreate
EtCreateIndex
EtCreateIndex2
EtDelete
EtDeleteIndex
EtEOF
EtGetCurRecAdr
EtGetCurVarAdr
EtGetIndex
EtInsert
EtLOF
EtMoveFirst
EtMoveLast
EtMoveNext
EtMovePrevious
EtOpen
EtPhoneHome (just kidding)
EtRetrieve
EtRetrieve2
EtRetrieveVariable
EtRetrieveVariable2
EtSeekEQ
EtSeekGE
EtSeekGT
EtSeekLE
EtSeekLT
EtSetIndex
EtTextComp
EtUnlockAllRecords
EtUnlockCurrentRecord
EtUpdate
EtUpdate2
EtUpdateVariable
EtUpdateVariable2

Function EtBOF% (Handle%)

Tests whether the last Move or Seek operation went past the beginning of the current index. It's generally used in conjunction with EtMovePrevious to determine when the beginning of the current index has been reached.

Parameters

Handle%

The E-Tree handle for the database.

Return Values

The value of the function is True (-1) if the last move or seek function placed the current record past the beginning of the current index (no more records past this point in the EtMovePrevious direction). For example, EtMoveFirst sets the current record to the first record in the current index. At this point, EtBOF would return False (0) because the pointer is at the first record and not past it. If, at this point, EtMovePrevious was invoked, the EtBOF function would return True because an attempt was made to move past the beginning of the index.

Also note that the EtBOF and EtEOF functions will both return True if no matches were found based on the last EtSeek.

Sub EtClose(Handle%)

Close a database and release all resources allocated for it.

When the last open database is closed, all the resources allocated by the E-Tree Memory and Lock managers are released.

Parameters

Handle%
    The E-Tree handle for the database to close.

Return Values

None. If an invalid handle is passed or if the database previously accessed with Handle% has already been closed, EtClose simply ignores the request.

Sub EtCloseAll

Closes all E-Tree database files and releases all resources allocated to them.

When the last open database is closed, all the resources allocated by the E-Tree Memory and Lock managers are released.

Parameters

Handle%
The E-Tree handle for the database to close.

Return Values

None.

In QB4.x, you cannot keep E-Tree files open across a CHAIN. However, you can keep them open if the E-Tree routines are in a BC6 or PDS extended runtime library (RTM).

If the E-Tree code is *not* in an RTM, you must call the EtCloseAll and EtUnhook routines immediately prior to the CHAIN.

If the E-Tree code *is* in an RTM, you must close all E-Tree files prior to SHELLing. This is because BASIC unloads the runtime library from memory in order to make more DOS memory available during the SHELL.

See Appendix D, Section H for more information about E-Tree, RTMs, CHAINing and SHELLing.

Sub EtCreate(FileName$, Pagesize&, PreAllocation&, _ RecordInfo() as EtRecordInfoType,_ MaxKeys%, Handle%, Status%)

Create, initialize, and open a new, empty E-Tree database file. The file is left open and should be explicitly closed using EtClose() before your program ENDs.

Parameters

FileName$

The name of the file to create. Use standard MSDOS file name conventions. If a file of the same name already exists in the target directory, an error will be returned. The EtFileDelete routine can be used to delete an existing file.

PageSize&

Defines the page size to use. If <= 0 then the optimum page size is calculated based on the number of fields, record length, and MaxKeys%.

PreAllocation&

Number of bytes you wish to preallocate for the file on disk. The value you provide will be rounded up to an even multiple of the file page size. A value of zero indicates that no extra disk space (over and above what is required to initialize the database) will be preallocated for the file. A value greater than the amount of available disk space will return an error. The EtFreeDiskSpace& function can be used to determine the amount of available disk space before creating the database.

RecordInfo()

An array of type EtRecordInfoType where each element will describe one field in the database. The bounds (size) of the array are checked and each element is considered a valid field descriptor unless .FldName is empty (all spaces). Each element in the RecordInfo array is comprised of the following components:

.FldName	Field name (30 character maximum)
.FldType	Field type (use a CONSTant listed below)
.FldLength	Field length (applies only to certain data types)

For data types with a predetermined length, the .FldLength element is returned as the length for that data type. For data types requiring a UserDefined length, the .FldLength must be supplied as desired for that field.

The following list defines the data types that are understood by E-Tree Plus and the named constant for each. Fields that are of a type with the length listed as UserDefined must have the .FldLength correctly assigned. For all other data types, .FldLength is ignored on entry. .FldLength for a variable-length field is also ignored.

If the record definition will include a variable-length field, it must be the *last* field defined in the RecordInfo array. You can have only one variable-length field in a record definition. etconst.bi

Type	Named Constant	Length
Variable Length Data	EtVariable 0	UserDefined
Signed Binary Integer	EtSigned 1	UserDefined
Unsigned Binary Integer	EtUnSigned 2	UserDefined
Auto Increment	EtAuto 3	4 bytes (long integer)
Integer (signed)	EtInteger 4	2 bytes
Long Integer (signed)	EtLong 5	4 bytes
IEEE Single Precision	EtSingle 6	4 bytes
IEEE Double Precision	EtDouble 7	8 bytes
IEEE Extended Precision	EtIEEEx 8	10 bytes
PB Fixed Point BCD	EtPBBCDFixed 9	8 bytes
PB Floating Point BCD	EtPBBCDFloat 10	10 bytes
MBF Single Precision	EtMBFs 11	4 bytes
MBF Double Precision	EtMBFd 12	8 bytes
PDS Currency	EtCurrency 13	8 bytes
ASCII Case Insensitive	EtString 15	UserDefined
ASCII Case Sensitive	EtStringCS 14	UserDefined

MaxKeys%

The maximum number of indexes expected to be added to the file at one time (if PageSize% is > 0, the value of MaxKeys% is ignored).

Handle%

If you pass a value of 0, the routine will automatically assign the next available E-Tree file handle. However, if you wish to use one of your own, you can pass a value between 1 and the maximum number of open E-Tree files allowed. The default maximum value is 4. See the EtInitManager routine for information on changing this default. If you pass an invalid value for Handle%, the EtCreate and EtOpen functions will fail and return an error code.

Return Values

Handle%

A value used by other ISAM functions to access the database that is created. DO NOT alter this number or none of the other E-Tree functions will work correctly. This value is not a DOS file handle but an index into the various tables that are maintained within the E-Tree code.

RecordInfo(x).FldOffset

Offsets into record for each field (zero based). The offsets can be used for creating key fields for indexes (see EtCreateIndex2).

PageSize&

The page size that was used when creating the file.

PreAllocation&

If >0 on entry, it is returned with the actual number of bytes allocated to the file. If an error occurred as a result of an excessive PreAllocation request, the maximum number of bytes available on disk to allocate to the file will be returned.

MaxKeys%

If you passed a PageSize& > 0 then this is the maximum number of indexes that can be added to the database at one time.

Status%

Returned as zero if the database was successfully created and initialized or as a non-zero error/status code. If an error occurs during preallocation, the database will not be created (if a partial creation was managed, the file will be deleted).

See Section 4 (G) for a complete list of E-Tree and DOS error codes.

Sub EtCreateIndex (Handle%, IndexName$, Unique%,_ Columns$(), Status%)

Create a new index in a database where the key is comprised of one or more complete fields.

Parameters

Handle%

The E-Tree handle for the database in which you wish to create the index.

IndexName$

The name for the index, must be 30 characters or less.

Unique%

If non-zero then duplicate keys are not allowed in the index

Columns$()

A variable-length string array containing the names of the Fields that you wish to include in the key description. The index will be ordered according to the order you list the field names in the Columns$() array.

Return Values

Status%

Returns zero if successful. A non-zero value represents an error code. See Section 4(J) for a complete list of E-Tree and DOS error codes.

For example, if you had a customer list database and you wanted to create an index that would order your data by AmountDue then by CompanyName:

```
REDIM Columns$(1 TO 2)         'Only need two elements
Columns$(1) = "AmountDue"      'Name of field that contains AmountDue data
Columns$(2) = "CompanyName"    'Name of field containing company name
IndexName$ = "MyIndex"
CALL EtCreateIndex(Handle%, IndexName$, Unique%, Columns%(), Status%)
IF Status% THEN
  PRINT "Error";Status%;" creating index ";IndexName$
END IF
```

If you need greater flexibility when defining the segments of your data record to be included in an index, see EtCreateIndex2. It's more difficult to use than EtCreateIndex, but significantly more flexible.

Sub EtCreateIndex2 (Handle%, IndexName$, _ Desc() AS EtKeyDescType, _ Segments%, Unique%, Status%)

Create a new index in a database allowing segments definitions comprised of pieces of fields, unmodifiable keys, ascending and descending sorts, etc.

Parameters

Handle%

The E-Tree handle for the database in which you wish to create the index.

IndexName$

The name for the index, must be 30 characters or less.

Desc()

An array of EtKeyDescType that defines the:

.Offset

The offset into the data record for the beginning of the data for this key segment.

.Length

The number of bytes for this segment.

.Type

The Data type for this segment.

.Direction

0 for ascending, non-zero for descending.

.NoModify

If non-zero then this segment is protected from modification by updates to the data record.

Segments%

The number of elements, beginning at the array's LBOUND, used in the Desc() array to describe all of the segments comprising this index.

Unique%

If non-zero then duplicate keys are not allowed in this index

Return Values

Status%

Returns zero if successful. A non-zero value represents an error code. See Section 4(J) for a complete list of E-Tree and DOS error codes.

Sub EtDelete (Handle%, Status%)

Delete the current record from the database including any variable length data stored with the record. All indexes are updated to reflect the deletion.

Makes the record that would be accessed by EtMoveNext() the current record if the delete is successful.

Parameters

Handle%

The E-Tree handle for the database.

Return Values

Status%

Returns zero if successful. A non-zero value represents an error code. See Section 4(J) for a complete list of E-Tree and DOS error codes.

Sub EtDeleteIndex (Handle%, IndexName$, Status%)

Delete an index from a database.

Once an index database is deleted it can no longer be used to access data. The index must not be in use by another process in order for the deletion to be successful.

Parameters

Handle%
    The E-Tree handle for the database.

IndexName$
    The name of the index to delete.

Return Values

Status%
    Returns zero if successful. A non-zero value represents an error code. See Section 4(J) for a complete list of E-Tree and DOS error codes.

If the index is in use by any other programs a LockedBySemaphore error will be returned.

Function EtEOF% (Handle%)

Tests whether the last Move or Seek operations went past the end of the current index. It's generally used in conjunction with EtMoveNext to determine when the end of the current index has been reached.

Parameters

Handle%

The E-Tree handle for the database.

Return Values

The value of the function is True (-1) if the last move or seek function placed the current record past the end of the current index (no more records past this point in the EtMoveNext direction). For example, EtMoveLast sets the current record to the last record in the current index. At this point, EtEOF would return False (0) because the pointer is at the last record and not past it. If, at this point, EtMoveNext was invoked, the EtEOF function would return True because an attempt was made to move past the end of the index.

Also note that the EtBOF and EtEOF functions will both return True if no matches were found based on the last EtSeek.

Function EtGetCurRecAdr& (Handle%)

Returns the physical address for the current record in a database. Every active data record in the database has a unique address which was assigned when the record was inserted. The address cannot be changed or used by any other record as long as the record is not deleted. Once the record is deleted the address will be re-used when another new record is inserted.

The address defines the page and offset into the page where the data record and information about the data record are stored. Direct manipulation of the database using this address should only be handled by the low-level routines in the E-Tree Library.

Parameters

Handle%

The E-Tree handle for the database.

Return Values

The value of the function is the physical address for the current record. A value of zero indicates that no record is current. This can occur when the database is first opened (before any EtSeek or EtMove functions are used), if there are no records in the database, or if a previous error caused the current position in an index to be lost.

EtGetCurVarAdr& (Handle%)

Returns the physical address for the variable-length data for the current record in a database.

Parameters

Handle%
: The E-Tree handle for the database.

Return Values

The value of the function is the physical address for the position discriptor for the variable length data for the current record. The descriptor contains information which defines the actual physical location and length of the data.

EtGetCurVarLen& (Handle%)

Returns the length of the variable-length data for the current record.

Parameters

Handle%

The E-Tree handle for the database.

Return Values

The function returns the length of the variable-length data for the current record.

Function EtGetIndex$ (Handle%)

Returns the name of the current index being used to access data in the database.

Parameters

Handle%
: The E-Tree handle for the database.

Return Values

The function returns a string that contains the name of the index. If a string of zero length is returned then the Null index is being used.

Sub EtInsert (Handle%, Buffer&, LockFlag%, Status%)

Insert a new fixed-length data record into a database.

Parameters

Handle%

The E-Tree handle for the database.

Buffer&

The memory address of the buffer containing the data record to be inserted into the database. This can be a "near" offset to a variable in DGroup (as in VarPtr(TypedVariable)) or a "far" address pointing to a PDS Far String (as in SSEGADD(A$)). If you have your own special type of record buffer defined at a specific location in memory, you can use the EtFarAddress routine to convert the segment and offset components into a 4-byte far address required by this routine.

The data record should be stored in a static data type (a piece of data that will always be at a fixed address). It can be a user defined type, a fixed length string, or a static array of user types or fixed-length strings. It should not be stored in a variable-length string or an element of a variable-length string array because it has a possibility of being moved during the runtime "garbage collection" process.

LockFlag%

Defines the locking condition for the new record:

0 -- no locks placed or removed
1 -- lock before, unlock after (infinite time out)
2 -- lock before, leave locked (infinite time out)
3 -- lock before, unlock after (system time out)
4 -- lock before, leave locked (system time out)
5 -- assume locked, unlock after

If a lock is requested, the lock applied is a READ-ONLY soft-lock on the data record only.

If a "leave-locked" type (2 or 4) is requested, the data record will be soft-locked if the insert operation is successful.

Return Values

Status%

Returns zero if successful. A non-zero value represents an error code. See Section 4(J) for a complete list of E-Tree and DOS error codes.

Function EtLOF& (Handle%)

Report the number of active records in the database.

Parameters

Handle%
    The E-Tree handle for the database.

Return Values

The value of the function is the number of active data records in the database.

A value of zero usually means there are zero records in the database, but it can also indicate that the handle is invalid, the lock failed, or the file read failed. In either case zero is a reasonable result.

Sub EtMakeRecordCurrent (Handle%, Address&, Status%)

Makes the record referenced by Address& the current record in the database referenced by Handle%.

If the null index is current, this routine takes you directly to that record's physical location in the database.

If an index other than the null index is current, this routine makes the record referenced by Address& the current record in the current index.

Parameters

Handle%

The E-Tree handle of the database file.

Address&

The physical address of the desired data record. This value can be determined by using the EtGetCurRecAdr routine while the desired record was previously current.

Return Values

Status%

Returns zero if successful. A non-zero value represents an error code. See Section 4(J) for a complete list of E-Tree and DOS error codes.

Sub EtMoveFirst (Handle%, Status%)
Sub EtMoveLast (Handle%, Status%)
Sub EtMoveNext (Handle%, Status%)
Sub EtMovePrevious (Handle%, Status%)

Make the record indicated (first, last, next, or previous) in the current index the current record.

This routine does not retrieve any record data. It simply moves the current record pointer.

Parameters

Handle%

The E-Tree handle for the database.

Return Values

Status%

Returns zero if successful. A non-zero value represents an error code. See Section 4(J) for a complete list of E-Tree and DOS error codes.

Sub EtOpen (FileName$, Handle%, Status%)

Open an existing file as an E-Tree database file.

To create and initialize a new E-Tree file, see the EtCreate routine.

Parameters

FileName$

The file name of the database to open.

Handle%

If you pass a value of 0, the routine will automatically assign the next available E-Tree file handle. However, if you wish to use one of your own, you can pass a value between 1 and the maximum number of open E-Tree files allowed. The default maximum value is 4. See the EtInitManager routine for information on changing this default. If you pass an invalid value for Handle%, the EtCreate and EtOpen functions will fail and return an error code.

Return Values

Handle%

A valid E-Tree handle for subsequent access to the database.

Status%

Returns zero if successful. A non-zero value represents an error code. See Section 4(J) for a complete list of E-Tree and DOS error codes.

If the file exists, has a length > 0, and is not an E-Tree database file, it is closed and the open fails.

Sub EtRetrieve (Handle%, Buffer&, LockFlag%,_ Status%)

Retrieve the current record from the database referenced by Handle%.

Parameters

Handle%

The E-Tree handle for the database.

Buffer&

The memory address of the TYPEd variable or block of memory in which you wish to place the incoming data record. This can be a "near" offset to a variable in DGroup (as in VarPtr(TypedVariable)) or a "far" address pointing to a PDS Far String (as in SSEGADD(A$)). You can also load the record into a specific memory location. The EtFarAddress routine is used to convert the segment and offset components into a 4-byte far address.

The data record should be stored in a static data type (a piece of data that BASIC will always keep at a fixed address). It can be a user defined type, a fixed length string, or a static array of user types or fixed-length strings. It should not be stored in a variable-length string or an element of a variable-length string array because it has a possibility of being moved during the runtime "garbage collection" process.

LockFlag%

Defines the locking condition for the new record:

0 --	no locks placed or removed
1 --	lock before, unlock after (infinite time out)
2 --	lock before, leave locked (infinite time out)
3 --	lock before, unlock after (system time out)
4 --	lock before, leave locked (system time out)
5 --	assume locked, unlock after

If a lock is requested, the lock applied is a READ-ONLY soft-lock.

If a type 2 lock is used, the code will wait indefinitely for the record's semaphore to become available. If you are using a type 4 lock, the code will wait for the EtSystemTimeout time for the record's semaphore to become available. If the semaphore is not available within this time, a LockedBySema error will be returned.

If an unlock-after type (1,3,5) is requested the data record will be unlocked (the semaphore will be decremented) and unlogged if the retrieve is successful.

See the EtLockTimeoutMsg and EtLockNoTimeoutMsg routines if you wish to add custom code to handle lock collisions.

Return Values

Status%

Returns zero if successful. A non-zero value represents an error code. See Section 4(J) for a complete list of E-Tree and DOS error codes.

Sub EtRetrieve2 (Handle%, Address&, Buffer&,_ LockFlag%, Status%)

Retrieve the fixed-length portion of a record stored at a specific location in the database.

Parameters

Handle%

The E-Tree handle for the database.

Address&

The address within the E-Tree database file where the record you wish to retrieve is located. The EtGetCurRecAdr function can be used to determine the address of the current record in the database.

Buffer&

The memory address of the TYPEd variable or block of memory in which you wish to place the incoming data record. This can be a "near" offset to a variable in DGroup (as in VarPtr(TypedVariable)) or a "far" address pointing to a PDS Far String (as in SSEGADD(A$)). You can also load the record into a specific memory location. The EtFarAddress routine is used to convert the segment and offset components into a 4-byte far address.

The data record should be stored in a static data type (a piece of data that BASIC will always keep at a fixed address). It can be a user defined type, a fixed length string, or a static array of user types or fixed-length strings. It should not be stored in a variable-length string or an element of a variable-length string array because it has a possibility of being moved during the runtime "garbage collection" process.

This routine assumes that the buffer referenced by Buffer& is long enough to hold the entire fixed-length portion of the record. If it is not, memory corruption will certainly result.

LockFlag%

Defines the locking condition for the new record:

0 --	no locks placed or removed
1 --	lock before, unlock after (infinite time out)
2 --	lock before, leave locked (infinite time out)
3 --	lock before, unlock after (system time out)
4 --	lock before, leave locked (system time out)
5 --	assume locked, unlock after

If a lock is requested, the lock applied is a READ-ONLY soft-lock.

If a type 2 lock is used, the code will wait indefinitely for the record's semaphore to become available. If you are using a type 4 lock, the code will wait for the EtSystemTimeout time for the record's semaphore to become available. If the semaphore is not available within this time, a LockedBySema error will be returned.

If an unlock-after type (1,3,5) is requested the data record will be unlocked (the semaphore will be decremented) and unlogged if the retrieve is successful.

See the EtLockTimeoutMsg and EtLockNoTimeoutMsg routines if you wish to add custom code to handle lock collisions.

Return Values

Status%

Returns zero if successful. A non-zero value represents an error code. See Section 4(J) for a complete list of E-Tree and DOS error codes.

Sub EtRetrieveVariable (Handle%, Buffer$, LockFlag%,_ Status%)

Retrieve the variable length data for the current record.

Parameters

Handle%

The E-Tree handle for the database

Buffer$

A variable-length string that the incoming data will be placed in. If you wish to place the data in a variable type or a memory location other than a conventional string, the EtRetrieve2 routine can be used instead for this purpose.

LockFlag%

Defines the locking condition for the new record:

0 --	no locks placed or removed
1 --	lock before, unlock after (infinite time out)
2 --	lock before, leave locked (infinite time out)
3 --	lock before, unlock after (system time out)
4 --	lock before, leave locked (system time out)
5 --	assume locked, unlock after

If a lock is requested, the lock applied is a READ-ONLY soft-lock placed on the fixed-length portion of the record only. If you already have a lock placed on the fixed-length portion using EtInsertX, EtUpdateX, or EtRetrieveX, you should use a LockFlag% of 0 when calling this routine.

If a type 2 lock is used, the code will wait indefinitely for the record's semaphore to become available. If you are using a type 4 lock, the code will wait for the EtSystemTimeout time for the record's semaphore to become available. If the semaphore is not available within this time, a LockedBySema error will be returned.

If an unlock-after type (1,3,5) is requested the data record will be unlocked (the semaphore will be decremented) and unlogged if the retrieve is successful.

See the EtLockTimeoutMsg and EtLockNoTimeoutMsg routines if you wish to add custom code to handle lock collisions.

Return Values

Buffer$

A variable-length string that receives the incoming data.

Status%

Returns zero if successful. A non-zero value represents an error code. See Section 4(J) for a complete list of E-Tree and DOS error codes.

Sub EtRetrieveVariable2 (Handle%, Address&, Buffer&,_ BufferLen&, LockFlag%, Status%)

Retrieve the variable length data for the current record.

Parameters

Handle%

The E-Tree handle for the database

Address&

The address in the database where the *fixed-length* portion of the record is stored. The EtGetCurRecAdr function can be used to return the address of the fixed-length portion of the current record.

Buffer&

The memory address of the TYPEd variable or block of memory in which you wish to place the incoming data record. This can be a "near" offset to a variable in DGroup (as in VarPtr(TypedVariable)) or a "far" address pointing to a PDS Far String (as in SSEGADD(A$)). You can also load the record into a specific memory location. The EtFarAddress routine is used to convert the segment and offset components into a 4-byte far address.

The data record should be stored in a static data type (a piece of data that BASIC will always keep at a fixed address). It can be a user defined type, a fixed length string, or a static array of user types or fixed-length strings. It should not be stored in a variable-length string or an element of a variable-length string array because it has a possibility of being moved during the runtime "garbage collection" process.

BufferLen&

The maximum length of the buffer referenced by the Buffer& address.

LockFlag%

Defines the locking condition for the new record:

0 --	no locks placed or removed
1 --	lock before, unlock after (infinite time out)
2 --	lock before, leave locked (infinite time out)
3 --	lock before, unlock after (system time out)
4 --	lock before, leave locked (system time out)
5 --	assume locked, unlock after

If a lock is requested, the lock applied is a READ-ONLY soft-lock placed on the fixed-length portion of the record only. If you already have a lock placed on the fixed-length portion using EtInsertX, EtUpdateX, or EtRetrieveX, you should use a LockFlag% of 0 when calling this routine.

If a type 2 lock is used, the code will wait indefinitely for the record's semaphore to become available. If you are using a type 4 lock, the code will wait for the EtSystemTimeout time for the record's semaphore to become available. If the semaphore is not available within this time, a LockedBySema error will be returned.

If an unlock-after type (1,3,5) is requested the data record will be unlocked (the semaphore will be decremented) and unlogged if the retrieve is successful.

See the EtLockTimeoutMsg and EtLockNoTimeoutMsg routines if you wish to add custom code to handle lock collisions.

Return Values

Buffer&

If the length of the buffer is sufficient to hold the entire variable-length portion of the record, the memory address referenced by Buffer& will receive the incoming data.

BufferLen&

If the buffer is too small, BufferLen& will return the size of a buffer required to receive the entire variable-length portion of the record.

Status%

Returns zero if successful. A non-zero value represents an error code. See Section 4(J) for a complete list of E-Tree and DOS error codes.

Sub EtSeekEQ (Handle%, KeyAddress&, Status%) =
Sub EtSeekGT (Handle%, KeyAddress&, Status%) >
Sub EtSeekGE (Handle%, KeyAddress&, Status%) >=
Sub EtSeekLT (Handle%, KeyAddress&, Status%) <
Sub EtSeekLE (Handle%, KeyAddress&, Status%) <=

Make the first record matching the data according to the relationship specified the current record. These routines do not work on the "null" index.

Parameters

Handle%

The E-Tree handle for the database.

KeyAddress&

The memory address of a key value to use in the Seek operation. This can be a "near" offset into DGroup (Varptr) of a scalar or TYPEd variable, or a far address to another type data structure. TYPEd variables (with each element representing a key segment) or a fixed-length string (for keys containing only a single segment) are ideal for this purpose. The data structure that KeyAddress& points to is expected to have the same structure and length as the key defined by EtCreateIndex or EtCreateIndex2.

Return Values

The EtEOF and EtBOF functions will return False (0) if a match was found and the current record pointer will be positioned at that record's address in the database file. One of the EtRetrieveX routines can then be used to retrieve the data associated with the record. The EtMoveXX routines can be used to move in the desired direction in the current index so that additional records can also be retrieved.

Status%

Returns zero if successful. A non-zero value represents an error code. See Section 4(J) for a complete list of E-Tree and DOS error codes.

Sub EtSetIndex (Handle%, IndexName$, Status%)

Make a specified index the current index and make the first record in that index the current record.

Parameters

Handle%
The E-Tree handle for the database.

IndexName$
The name of an index that exists in the database.

Return Values

Status%
Returns zero if successful. A non-zero value represents an error code. See Section 4(J) for a complete list of E-Tree and DOS error codes.

Function EtTextComp% (A$, B$)

Performs a case-insensitive ASCII comparison between A$ and B$.

The dealing with indexes based on string fields, the EtSeekXX routines will locate the first record matching your search criteria. If you wish to retrieve a group of records, you must then EtMoveXX to the next and subsequent records and compare the key values yourself in order to determine when you've reached the end of your desired selection set. This routine is useful when you need to make such case-insensitive comparisons of ASCII strings.

Parameters

A$ and B$

The two strings that will be compared.

Return Values

The result of the functions will be one of the following:

0	A$ = B$
-1	A$ < B$
1	A$ > B$

Sub EtUnlockAllRecords (Handle%)

Removes all record locks placed by the current process in the database referenced by Handle%.

Parameters

Handle%

The E-Tree handle for the database.

Return Values

None.

When locking several records as part of a transaction, this routine makes is easy to globally remove all of the locks with one function call when the transaction is completed or aborted.

Sub EtUnlockCurrentRecord (Handle%)

Removes a lock placed on the current record by the EtInsert, EtUpdate or EtRetrieveX routines. This call is ignored if the current record is not currently locked.

After placing a lock on a record during an Insert, Update or Retrieve operation, this routine can simply remove the lock without requiring a call to the EtUpdate routine. This is useful when you retrieve and lock a record for editing and then decide not to update the record in the database file (thus unlocking the record).

Parameters

Handle%
    The E-Tree handle for the database.

Return Values

None.

Sub EtUpdate (Handle%, Buffer&, LockFlag%, Status%)

Update the fixed-length portion of the current data record. Update all indexes to reflect any changes in the data.

Parameters

Handle%

The E-Tree handle for the database.

Buffer&

The memory location where the NEW data for this record is stored. This can be a "near" offset to a variable in DGroup (as in VarPtr(TypedVariable)) or a "far" address pointing to a PDS Far String (as in SSEGADD(A$)). If you have a record buffer of your own defined at a specific location in memory, you can use the EtFarAddress routine to convert the segment and offset components into a 4-byte far address.

The data record should be stored in a static data type (a piece of data that will always be at a fixed address). It can be a user defined type, a fixed length string, or a static array of user types or fixed-length strings. It should not be stored in a variable-length string or an element of a variable-length string array because it has a possibility of being moved during the runtime "garbage collection" process.

LockFlag%

Defines the locking condition for the new record:

0 --	no locks placed or removed
1 --	lock before, unlock after (infinite time out)
2 --	lock before, leave locked (infinite time out)
3 --	lock before, unlock after (system time out)
4 --	lock before, leave locked (system time out)
5 --	assume locked, unlock after

If a type 2 lock is used, the code will wait indefinitely for the record's semaphore to become available. If you are using a type 4 lock, the code will wait for the EtSystemTimeout time for the record's semaphore to become available. If the semaphore is not available within this time, a LockedBySema error will be returned.

If an unlock-after type (1,3,5) is requested the data record will be unlocked (the semaphore will be decremented) and unlogged if the retrieve is successful.

See the EtLockTimeoutMsg and EtLockNoTimeoutMsg routines if you wish to add custom code to handle lock collisions.

Return Values

Status%

Returns zero if successful. A non-zero value represents an error code. See Section 4(J) for a complete list of E-Tree and DOS error codes.

Sub EtUpdate2 (Handle%, Address&, Buffer&, LockFlag%,_ Status%)

Update the fixed-length portion of the record referenced by Address&. Updates all indexes to reflect any changes in the data.

Parameters

Handle%

The E-Tree handle for the database.

Address&

The address of the fixed-length portion of the data record. The EtGetCurRecAdr function can be used to determine the address of the fixed-length portion of the current record.

Buffer&

The memory location where the NEW data for this record is stored. This can be a "near" offset to a variable in DGroup (as in VarPtr(TypedVariable)) or a "far" address pointing to a PDS Far String (as in SSEGADD(A$)). If you have a record buffer of your own defined at a specific location in memory, you can use the EtFarAddress routine to convert the segment and offset components into a 4-byte far address.

The data record should be stored in a static data type (a piece of data that will always be at a fixed address). It can be a user defined type, a fixed length string, or a static array of user types or fixed-length strings. It should not be stored in a variable-length string or an element of a variable-length string array because it has a possibility of being moved during the runtime "garbage collection" process.

LockFlag%

Defines the locking condition for the new record:

0 -- no locks placed or removed
1 -- lock before, unlock after (infinite time out)
2 -- lock before, leave locked (infinite time out)
3 -- lock before, unlock after (system time out)
4 -- lock before, leave locked (system time out)
5 -- assume locked, unlock after

If a type 2 lock is used, the code will wait indefinitely for the record's semaphore to become available. If you are using a type 4 lock, the code will wait for the EtSystemTimeout time for the record's semaphore to become available. If the semaphore is not available within this time, a LockedBySema error will be returned.

If an unlock-after type (1,3,5) is requested the data record will be unlocked (the semaphore will be decremented) and unlogged if the retrieve is successful.

See the EtLockTimeoutMsg and EtLockNoTimeoutMsg routines if you wish to add custom code to handle lock collisions.

Return Values

Status%

Returns zero if successful. A non-zero value represents an error code. See Section 4(J) for a complete list of E-Tree and DOS error codes.

Sub EtUpdateVariable (Handle%, Buffer$, LockFlag%,_ Status%)

Update the variable-length portion of the current data record.

Parameters

Handle%

The E-Tree handle for the database.

Buffer$

A variable-length string containing the new data to be stored with this record.

LockFlag%

Defines the locking condition for the new record:

0 -- no locks placed or removed
1 -- lock before, unlock after (infinite time out)
2 -- lock before, leave locked (infinite time out)
3 -- lock before, unlock after (system time out)
4 -- lock before, leave locked (system time out)
5 -- assume locked, unlock after

If a lock is requested, the lock applied is a READ-ONLY soft-lock placed on the fixed-length portion of the record only. If you already have a lock placed on the fixed-length portion using EtInsertX, EtUpdateX, or EtRetrieveX, you should use a LockFlag% of 0 when calling this routine.

If a type 2 lock is used, the code will wait indefinitely for the record's semaphore to become available. If you are using a type 4 lock, the code will wait for the EtSystemTimeout time for the record's semaphore to become available. If the semaphore is not available within this time, a LockedBySema error will be returned.

If an unlock-after type (1,3,5) is requested the data record will be unlocked (the semaphore will be decremented) and unlogged if the retrieve is successful.

See the EtLockTimeoutMsg and EtLockNoTimeoutMsg routines if you wish to add custom code to handle lock collisions.

Return Values

Status%

Returns zero if successful. A non-zero value represents an error code. See Section 4(J) for a complete list of E-Tree and DOS error codes.

Sub EtUpdateVariable2 (Handle%, Address&, Buffer&, _ BufferLen&, LockFlag%, Status%)

Update the variable-length portion of the record referenced by Address&.

Parameters

Handle%

The E-Tree handle for the database.

Address&

The address of the *fixed-length* portion of the data record. The EtGetCurRecAdr function can be used to determine the address of the fixed-length portion of the current record.

Buffer&

The memory location where the NEW data for the variable-length portion of this record is stored. This can be a "near" offset to a variable in DGroup (as in VarPtr(TypedVariable)) or a "far" address pointing to a PDS Far String (as in SSEGADD(A$)). If you have a record buffer of your own defined at a specific location in memory, you can use the EtFarAddress routine to convert the segment and offset components into a 4-byte far address.

The data record should be stored in a static data type (a piece of data that will always be at a fixed address). It can be a user defined type, a fixed length string, or a static array of user types or fixed-length strings. It should not be stored in a variable-length string or an element of a variable-length string array because it has a possibility of being moved during the runtime "garbage collection" process.

BufferLen&

The length of the valid data stored at the memory location referenced by Buffer&. This is how many bytes will be written to the variable-length portion of the record.

LockFlag%

Defines the locking condition for the new record:

0 --	no locks placed or removed
1 --	lock before, unlock after (infinite time out)
2 --	lock before, leave locked (infinite time out)
3 --	lock before, unlock after (system time out)
4 --	lock before, leave locked (system time out)
5 --	assume locked, unlock after

If a lock is requested, the lock applied is a READ-ONLY soft-lock placed on the fixed-length portion of the record only. If you already have a lock placed on the fixed-length portion using EtInsertX, EtUpdateX, or EtRetrieveX, you should use a LockFlag% of 0 when calling this routine.

If a type 2 lock is used, the code will wait indefinitely for the record's semaphore to become available. If you are using a type 4 lock, the code will wait for the EtSystemTimeout time for the record's semaphore to become available. If the semaphore is not available within this time, a LockedBySema error will be returned.

If an unlock-after type (1,3,5) is requested the data record will be unlocked (the semaphore will be decremented) and unlogged if the retrieve is successful.

See the EtLockTimeoutMsg and EtLockNoTimeoutMsg routines if you wish to add custom code to handle lock collisions.

Return Values

Status%

Returns zero if successful. A non-zero value represents an error code. See Section 4(J) for a complete list of E-Tree and DOS error codes.

6 (B) High Level Routines

High-Level routines are those that report on the state of a database and provide access to the data structures used to maintain a database. In general the data structures they access and report on should not be manipulated directly.

EtFarAddress
EtFarToSegment
EtFarToOffset
EtGetCurrentRecordCRC
EtGetFileHandle
EtGetLockTimeout
EtGetPageSize
EtGetRecordSize
EtInitManager
EtInitManagerDefault
EtLockTimeoutMsg
EtLockNoTimeoutMsg
EtSetLockTimeout

FUNCTION EtFarAddress& (Segment&, Offset&)
FUNCTION EtFarToSegment& (FarAddress&)
FUNCTION EtFarToOffset& (FarAddress&)

Routines to convert to and from a four-byte far address.

This is an assembly language routine contained in ETCPYMEM.OBJ.

Although we use long integers to represent Segment and Offset, integers can actually be used instead. We chose longs because segment and offset addresses have a range from 0 to 65535 and they are easier to deal with in BASIC when in unsigned form because BASIC display's signed integer values > 32767 as negative numbers.

Parameters

Segment& and Offset&

> The segment and offset components that will be combined into a 4-byte far address by the EtFarAddress routine.

FarAddress&

> Contains the 4-byte far address you are converting from.

Return Values

> EtFarAddress returns a long integer which contains the combined segment and offset addresses.
>
> EtFarToSegment returns a long integer that represents the segment portion of the FarAddress&.
>
> EtFarToOffset returns a long integer that represents the offset portion of the FarAddress&.

Function EtGetCurrentRecordCRC& (Handle%)

Returns the CRC value of the current data record. The CRC for a data record is updated on the following actions: EtInsert, EtUpdate, and EtRetrieve. This value is used as a check for damaged data. Each time the data is read, the CRC is recalculated and checked against the CRC stored in the file. If the two values are not the same then either the database is damaged or the specific record is damaged.

Parameters

Handle%

The E-Tree handle for the database.

Return Values

The value of the function is a 32-bit unsigned integer value of the CRC.

Note: This function does not distinguish between an invalid handle and one that simply does not have a current record defined. To make sure the handle is valid, use one of the other EtGetXXX() functions first.

Function EtGetFileHandle% (Handle%)

Returns the DOS file handle associated with an E-Tree database.

Parameters

Handle%
    The E-Tree handle for the database.

Return Values

The value of the function is the DOS file handle, -1 if the E-Tree handle is invalid, or zero if the DOS file handle is invalid.

The Handle% value returned by E-Tree's EtCreate and EtOpen routines is not the actual DOS file handle. Instead, it is an index into the various tables of information that must be maintained within the E-Tree record management code. Although direct manipulation of an E-Tree database file is not recommended, the EtGetFileHandle routine will return the DOS file handle for a given E-Tree database file should you choose to do so.

Function EtGetLockTimeOut& ()

Returns the current value used for the lock time out timer.

Before this routine is used EtInit() MUST be called to initialize the file lock manager.

The routine EtSetLockTimeout is used to change the "system timeout" setting.

Parameters

None

Return Values

The function returns the number of timer ticks that define the duration of time in which the EtLock%() routine will attempt to set a lock.

Function EtGetPageSize& (Handle%)

Returns the page size used by an E-Tree database.

Parameters

Handle%
    The E-Tree handle for the database.

Return Values

The value of the function is the page size used by the data base, -1 if the E-Tree handle is invalid, or zero if the page size in invalid.

Function EtGetRecordSize& (Handle%)

Returns the size of the fixed length data records in an E-Tree data base.

Parameters

Handle%
    The E-Tree handle for the database.

Return Values

The value of the function is the size of the data record for the database, -1 if the E-Tree handle is invalid, or zero.

A zero length record size means the record has either not been defined or contains only a variable length field. Check the variable length flag to decide. If the variable length flag is false (zero) then the record layout has not yet been defined.

Function EtGetVariableFlag% (Handle%)

Returns the Variable Length Data flag for an E-Tree database. This flag indicates whether the records in the database have a variable length field defined.

Parameters

Handle%
The E-Tree handle for the database.

Return Values

The value of the function is true (<>0) if the record has a variable length field defined, or false (zero) if it doesn't.

This function does not distinguish between an invalid handle and one that simply does not have a variable length field. To make sure the handle is valid use one of the other EtGetXXX() functions first.

Function EtInitManager% (MaxOpenFiles%)

Each time this routine is called the E-Tree Manager is initialized. If it is not the first time then all open databases are closed, all allocated memory is released, and the memory and lock managers are re-installed.

You can call this routine yourself before opening the first E-Tree database file, should you have a need to open more than four E-Tree files simultaneously.

Parameters

MaxOpenFiles%
: Defines the maximum number of open databases allowed. It is used to dimension the arrays used to manage the databases.

Return Values

The value of the functions is a zero for success or a non-zero for an error. If an error is reported, DO NOT use any of the E-Tree Manager routines.

Function EtInitManagerDefault% ()

Installs the E-Tree manager using the default maximum of 4 open databases.

Parameters

None

Return Values

The value of the functions is a zero for success or a non-zero for an error. If an error is reported, DO NOT use any of the E-Tree Manager routines.

Function EtLockNoTimeoutMsg% ()

This function is called whenever a lock with infinite timeout fails.

Before this routine is used EtInit() MUST be called to initialize the file lock manager.

If you wish to write your own generic "handler" for "infinite timeout" lock collisions, you can add your own code to this routine. It is located in the source code module ETLOCK.SUB. See Section 4(D) "Introduction to Record Locking" for more information.

Parameters

None

Return Values

The function is expected to return TRUE if the timeout is to be respected (the lock fails), or FALSE to try it again.

Function EtLockTimeOutMsg% ()

This function is called when ever a lock attempt with a timeout time greater than zero times out.

Before this routine is used EtInit() MUST be called to initialize the file lock manager.

If you wish to write your own generic "handler" for "system timeout" lock collisions, you can add your own code to this routine. It is located in the source code module ETLOCK.SUB. See Section 4(D) "Introduction to Record Locking" for more information.

Parameters

None

Return Values

The function is expected to return TRUE if the timeout is to be respected (the lock fails), or FALSE to try it again.

Sub EtSetLockTimeOut (T&)

Sets the "system" lock timeout value.

This is the time out used for data record locks and most system level locks. Some system level locks use an infinite time out and some use a zero time out where appropriate.

Before this routine is used EtInit() MUST be called to initialize the file lock manager.

The routine EtGetLockTimeout returns the current timeout setting.

Parameters

T&

The value defines the number of clock ticks for the time frame during which a lock will be retried. There are about 18.2 clocks ticks per second.

Any attempt to set the time out to a value less than 18 (about 1 second) will cause it to be forced to 18. The maximum value allowed is 1,572,480 (24 hours).

Return Values

None

6 (C) Library Support Routines

The following general purpose assembly language routines are available for use in your programs. They may even be used in programs which do not invoke other E-Tree Plus database functions.

- EtCopyMem
- EtCRC32
- EtCVI
- EtCVL
- EtDiskSpace
- EtErrCode
- EtFileClose
- EtFileDelete
- EtFileExist
- EtFileFlush
- EtFileOpen
- EtFileRead
- EtFileSize
- EtFileWrite
- EtFillMem
- EtFreeDiskSpace
- EtGetFilePointer
- EtInit
- EtLockRegion
- EtMemAlloc
- EtMemRelease
- EtMemReleaseAll
- EtMemSeg
- EtMemSize
- EtNetworkType
- EtPeekB
- EtPeekI
- EtPeekL
- EtPokeB
- EtPokeI
- EtPokeL
- EtReleaseAllLocks
- EtSetFilePointer
- EtUnLockRegion

```
Sub EtCopyMem (    BYVAL  SrcSeg%,_
                   BYVAL  SrcOffset%,_
                   BYVAL  DestSeg%,_
                   BYVAL  DestOffset%,_
                   BYVAL  Bytes&)
```

Copies the contents of memory from location to another.

Parameters

SrcSeg%
    The segment of the memory block to copy from.

SrcOffset%
    The offset of the memory block to copy from.

DestSeg%
    The segment of the memory block to copy to.

DestOffset%
    The offset of the memory block to copy to.

Bytes&
    The number of bytes to copy.

Return Values

None

```
Function EtCRC32& ( BYVAL   MemSeg%, _
                    BYVAL   MemOffset%, _
                    Bytes&, _
                    StartingCrc& )
```

Calculates a 32-bit CRC value on a block of memory.

Parameters

MemSeg%
: Segment of memory block on which to compute CRC

MemOffset%
: Offset of memory block

Bytes&
: Size of the memory block in bytes (64535 max)

StartingCRC&
: Set to Zero if the current block of memory is the first of several blocks (or the only block). Set to the CRC value from a previous block of memory if it is the second (or third, or fourth, etc.) in a group of memory blocks.

This routine will calculate the CRC for a memory block up to 65535 bytes long. If you need to calculate a CRC on a block of memory larger than that, set StartingCRC& to zero for the first block and then set it to the result of each block calculated for the remaining blocks.

Return Values

The function returns the 32-bit CRC for the block of memory.

Function EtCVI% (Value&)

Convert a long into an integer (4-bytes down to 2-bytes).

Parameters

Value&
: The value to convert

Return Values

Returns the long integer's least significant word (truncates the LONG into an INTEGER)

Function EtCVL& (Value%)

Converts an integer to an long (2-bytes up to 4-bytes).

Parameters

Value%
: The integer value to convert

Return Values

The function returns a value from zero 65535 that results from storing the 2-bytes of the integer into the low order two bytes of a long.

The sign bit is not extended in the long value returned. This can result in negative integer values being converted to positive long values.

Function EtErrCode% ()

Get the error code set by the last E-Tree Library function called that generated a DOS error.

Parameters

None.

Return Values

The function returns the error code.

Sub EtFileClose (FileHandle%)

Close a file opened using EtFileOpen().

Parameters

FileHandle%
The DOS file handle for the file (as returned by EtFileOpen%())

Return Values

The function returns zero for success or an error code.

Function EtFileDelete% (FileName$)

Deletes a file. Wildcard characters are not allowed.

Parameters

FileName$

The file specification for the file or files to delete. DOS wildcard characters are accepted.

Since wildcard characters can be used, many files can be deleted with one call. Be careful that the files specified are indeed the files you want deleted.

Return Values

The function returns zero for success or an error code.

Function EtFileExist% (FileName$)

Check for the existence of a file.

Parameters

FileName$
The path/name of the file in question. The drive/path is optional.

Return Values

The function returns TRUE (-1) if the file is present and FALSE if the file does not exist or if a DOS critical error occurred. The EtError() function can be used to fetch DOS error codes (if any).

Function EtFileFlush% (BYVAL Handle%)

Flush the DOS file buffers associated with an open file.

Parameters

Handle%.
The DOS file handle for the file

Return Values

The function returns zero (false) if successful or non-zero (true) if an error occurred.

Function EtFileOpen% (FileName$,_
BYVAL AccessMode%,_
Handle%)

Open a file in random (binary) mode. When a file is opened the file pointer is set to zero (the beginning of the file). All file I/O using EtFileRead%() and EtFileWrite%() is performed starting at the file read/write pointer location in the file. To read or write a record from the file you must set the file pointer to the correct location before performing the read or write.

Parameters

FileName$

Name of the file to open. Drive/path is optional.

AccessMode%

Bit-mapped to represent the type of lock and network access rights, where one of the following values is used. (Use a value of zero if not running on a network):

0 -	Compatibility mode (Network services not used)
16-	Deny all (No access permitted)
32-	Deny write (Read access only allowed)
48-	Deny read (Write access only allowed)
64-	Deny none (Read/write access allowed)

Return Values

The function returns zero for success or an error code. Routine-specific error codes include:
-1 if a null FileName$ is passed
-2 is returned if our internal file handle table is full (max of 20 open files)

Any other value represents a DOS error code

Function EtFileRead% (BYVAL Handle%,_ BYVAL Segment%, BYVAL Offset%, Bytes&)

Function EtFileWrite% (BYVAL Handle%,_ BYVAL Segment%, BYVAL Offset%, Bytes&)

Read/Write bytes from/to a file starting at the current File Pointer location. The file read/write pointer is set using EtSetFilePointer%(). All file I/O using EtFileRead%() and EtFileWrite%() is performed starting at the file read/write pointer location in the file. To read or write a record from the file you must set the file pointer to the correct location before performing the read or write. When a file is opened the file pointer is set to zero (the beginning of the file).

Parameters

Handle%

The DOS file handle for the file to read from or write to.

Segment%

The segment of the memory block to write data from or read data to.

Offset%

The offset of the memory block to write data from or read data to

Bytes&

The number of bytes to be read or written. A max value of 65535 bytes may be used. If a value of zero is passed, the file will be truncated or extended to the current pointer position.

Return Values

The function returns zero for success or an error code. A value of -1 indicates that a file handle number of 0 was passed. All other values represent DOS error codes.

If the result is zero, Bytes& is returned containing the actual number of bytes read from or written to the file. These values can be different if you attempt to read past the end of the file, or write more bytes than can be held on the current drive (disk full during write).

Function EtFileSize& (BYVAL Handle%)

Get the current size of an open file.

Parameters

Handle%.

The DOS file handle for the file

Return Values

The function returns the file size or a negative number as an error code. A value of -1 is returned if an invalid handle was passed.

Sub EtFillMem (BYVAL MemSeg%, _ BYVAL MemOffset%, BYVAL Bytes&,_ BYVAL FillByte%)

```
Sub EtFillMem (   BYVAL MemSeg%, _
                  BYVAL MemOffset%, BYVAL Bytes&,_
                  BYVAL FillByte% )
```

Fills a block of memory with a specified value.

Parameters

MemSeg%
: The segment of the memory block to fill

MemOffSet%
: The offset of the memory block to fill

Bytes&
: The number of bytes to fill

FillByte%
: The fill value to use (Only the low order byte of FillByte% is used).

Return Values

None

Function EtFreeDiskSpace& (Drive%)

Report the amount of free space on a disk.

Parameters

Drive% =

0 for default drive
1..26 for drive A: ... Z:

Return Values

The function returns the amount of free disk space on the drive indicated. Negative values indicate an error.

See also EtDiskSpace&()

Function EtGetFilePointer& (BYVAL Handle%)

Get the current position of the file read/write pointer for an open file.

Parameters

Handle%.
: The DOS file handle for the file

Return Values

The function returns zero for success or an error code. A value of -1 is returned if an invalid handle was passed.

Function EtSetFilePointer% (BYVAL Handle%,_ BYVAL Position&)

Sets the file read/write pointer for an open file. All file I/O using EtFileRead%() and EtFileWrite%() is performed starting at the file read/write pointer location in the file. To read or write a record from the file you must set the file pointer to the correct location before performing the read or write. When a file is opened the file pointer is set to zero (the beginning of the file).

Parameters

Handle%.
: The DOS file handle for the file

Position&
: The desired location for the file read/write pointer. This is an absolute byte offset from the beginning of the file. A value of zero will position the pointer at the first byte in the file.

Return Values

The function returns zero for success or an error code. A value of -1 indicates that a handle number of zero was passed to the routine.

Sub EtInit ()

Initialize the E-Tree Memory manager to provide a means to release all resources acquired regardless of how the program terminates.

This routine installs a hook into int 21h function 4ch (terminate). When the program terminates normally, we make sure all locks are removed, all memory is released, and all files are closed.

If this routine is called a second time during the execution of a process, it still takes the above actions. This is a safeguard in the event you are running the program within the QB/PB development and have to restart it.

This routine is called by EtInitManager() as part of its processing. The only reason you would need to call this routine directly is if you want to use the E-Tree memory management, file I/O, or file locking routines in a program that does not use the other E-Tree Manager routines.

Parameters

None

Return Values

None

Function EtLockRegion% (BYVAL Handle%,_ BYVAL FileOffset&, BYVAL RecordLength&)

Sub EtUnLockRegion (BYVAL Handle%,_ BYVAL FileOffset&, BYVAL RecordLength&)

Lock and unlock a region of a file using locks appropriate to the current type of network being used. The lock is attempted only once and the result returned immediately. Use EtLock%() if you want to have a lock timeout value used during the lock attempt.

For PDS 7.x EtUnLock() is declared as an ALIAS for EtUnLockRegion(). For all other compilers EtUnLock() simply executes EtUnLockRegion() with the arguments passed.

The arguments used when EtUnLockRegion() is called must match exactly to the arguments used by an earlier call to EtLockRegion%()

Before either of these routines is used EtInit() MUST be called to initialize the file lock manager.

Parameters

Handle%
: The DOS file handle for the file

FileOffSet&
: The offset of the start of the region in the file

RecordLength&
: The size of the region in bytes

Return Values

The EtLockRegion%() function returns zero for success or an error code. If unsuccessful, the value returned will be one of the following:

-1 -	Bad lock length
-2 -	Lock Table full

Positive error codes reflect a DOS or Novell error.

Function EtMemAlloc% (Bytes&, MemType%)

Reserves a block of DOS or EMS memory.

Before this routine is used EtInit() MUST be called to initialize the memory manager.

Parameters

Bytes&

The number of bytes to allocate.

DOS memory will be allocated in paragraph (16 byte) increments and EMS memory will be allocated in page (16K byte) increments. A maximum of 65536 bytes can be allocated in a single call.

MemType%

Defines the type of memory to allocate. If zero, we'll attempt to allocate the memory from EMS first. If sufficient EMS is not available, we'll attempt to allocate it from DOS. A value of 1 directs us to allocate the memory from EMS only. A value of 2 directs us to allocate the memory from DOS only.

Return Values

The function returns a value greater than zero for success or negative error code.

If the memory allocation is successful, the result of the function is the "handle" to the memory block. Use the EtMemSeg% function to get the actual segment address of the memory block.

If the number of bytes requested is not available, Bytes& will be returned with the size of the largest available memory block.

Function EtMemRelease% (BYVAL MemHandle%)

Releases a DOS or EMS memory block previously allocated with the EtMemAlloc routine.

Before this routine is used EtInit() MUST be called to initialize the memory manager.

Parameters

MemHandle%

A handle to a memory block previously assigned by the EtMemAlloc() routine.

Return Values

This function returns FALSE (0) if successful or TRUE (-1) if unsuccessful.

Sub EtMemReleaseAll ()

Releases all memory blocks previously allocated with the EtMemAlloc routine.

Parameters

None

Return Values

None

Function EtMemSeg% (BYVAL MemHandle%)

Returns the segment address for a memory block allocated using EtMemAlloc()

If the memory block is in EMS, the block is mapped into the page frame beginning at physical page 0. There can be only one EMS buffer accessed at any given time even though more than one can be allocated. You cannot copy directly from one EMS buffer to another.

If the memory block is DOS memory, the block will always start at offset zero in the segment returned.

Before this routine is used EtInit() MUST be called to initialize the memory manager.

Parameters

MemHandle%

The handle to a memory block previously allocated by the EtMemAlloc() routine.

Return Values

The result of the function is the target memory segment. If zero is returned, the MemHandle% passed was invalid.

Function EtMemSize& (BYVAL MemHandle%)

Returns the actual size of a memory block allocated with EtMemAlloc.

Before this routine is used EtInit() MUST be called to initialize the memory manager.

Parameters

MemHandle%
A handle to a block of memory allocated using the EtMemAlloc() routine.

Return Values

The function returns the actual size of the memory block allocated. Since EMS memory is allocated in 16k-byte blocks and DOS memory is allocated in 16-byte blocks the block allocated may not be exactly the size requested when the block was originally allocated.

Function EtNetworkType% ()

Determine the type of network locks currently being used by EtLockRegion%().

Before this routine is used EtInit() MUST be called to initialize the file lock manager.

Parameters

None

Return Values

The function returns:

-1 -	EtInit() has not yet been called
0 -	No network locks are being used
1 -	Novell
2 -	NETBIOS (or other network which uses DOS 3.1 locks)

Sub EtSetNetworkType (BYVAL NetType%)

Set the type of network locks to be used by EtLockRegion%()

Calls to this routine will be ignored after the EtLockRegion%() routine has been called once.

Parameters

NetType% is:

0 -	No Network locks
1 -	Novell network locks
2 -	DOS 3.1 network locks

Return Values

None

```
Function EtPeekB% ( BYVAL MemSeg%,_
                    BYVAL MemOffset% )

Function EtPeekI% ( BYVAL MemSeg%,_
                    BYVAL MemOffset% )

Function EtPeekL& ( BYVAL MemSeg%,_
                    BYVAL MemOffset% )
```

Read a memory location and return the value stored there.

Parameters

MemSeg%
: The segment of the memory location to read

MemOffSet%
: The offset of the memory location to read

Return Values

The function returns the value stored at the memory location.

Notice that EtPeekL& returns a Long Integer value.

EtPeekI and EtPeekL can return negative values depending on the data stored in the memory being peeked because BASIC treats the values returned as signed integers. If the data stored at the memory location has the high-order bit set, BASIC will consider the value negative.

EtPeekB returns a value from 0-255.

```
Sub EtPokeB ( BYVAL MemSeg%, BYVAL MemOffset%,_
              BYVAL Value% )
Sub EtPokeI ( BYVAL MemSeg%, BYVAL MemOffset%,_
              BYVAL Value% )
Sub EtPokeL ( BYVAL MemSeg%, BYVAL MemOffset%,_
              BYVAL Value& )
```

Poke (write) a value at a location in memory.

Parameters

MemSeg%

The segment of the memory location where the value will be written

MemOffSet%

The offset of the memory location where the value will be written

Value (& or %)

The value to write to the memory location

Notice that EtPokeL writes a LONG integer argument. EtPokeB only writes the low order byte of the integer value.

Return Values

None

Sub EtReleaseAllLocks ()

Releases all active locks put into place by the EtLockRegion routine.

Before this routine is used EtInit() MUST be called to initialize the file lock manager.

Parameters

None

Return Values

None

6 (D) Low-Level Support Routines

Since the majority of E-Tree Plus customers don't have a need for or an interest in the low-level index and file management routines, we chose not to document them in this revision of the manual. However, you will find a file on your distribution diskette called LOWLEVEL.EXE. Just run this program and the documentation for the low-level functions will be extracted into a standard ASCII text file called LOWLEVEL.DOC.

If you would like to see the low-level routines included in future revisions of the E-Tree Plus documentation, drop us a line. If there is sufficient interest in it, we will certainly be happy to do so.

6 (E) Routine Organization

The following is a list of all E-Tree's BASIC procedures and the source code module in which that can be found:

ETLINK.SUB	Function EtDLinkInsert%
	Function EtDLinkRemove%
	Function EtSLinkInsert%
	Function EtSLinkRemove%
ETLOCK.SUB	Function EtGetLockTimeOut&
	Function EtLock%
	Function EtLockNoTimeOutMsg%
	Function EtLockTimeOutMsg%
	Sub EtSetLockTimeOut
	Sub EtUnLock
ETREECRI.SUB	Function EtAddKeyDef%
	Sub EtCreateIndex
	Sub EtCreateIndex2
	Sub EtDeleteIndex
ETREECRR.ASC	Function EtAddFields%
	Sub EtCreate
ETREEHI.ASC	Function EtGetCurRecAdr&
	Function EtGetCurrentRecordCRC&

	Function EtGetCurVarAdr&
	Function EtGetCurVarLen&
	Function EtGetFileHandle%
	Function EtGetIndex$
	Function EtGetPageSize&
	Function EtGetRecordsize&
	Function EtGetVariableFlag%
	Function EtInitManager%
	Function EtInitManagerDefault%
	Sub EtMakeRecordCurrent
ETREELOW.SUB	Function EtCheckDataPageMap%
	Function EtCheckMemoryBufferSize%
	Function EtDeleteVarData%
	Function EtFindIndexDescription%
	Function EtGetFreePage%
	Function EtLogSoftLock%
	Function EtPageToFromList%
	Function EtRWPage%
	Function EtSetIndexInternalMin%
	Function EtSetUpCommon%
	Function EtUnGetFreePage%
	Function EtUnLogSoftLock%
	Function EtVarFindDesc%
	Sub EtCopyCommonToTable Static
	Sub EtMapDataPage
ETREEMID.SUB	Function EtAddToAllIndexes%
	Function EtCompareKeys%
	Function EtDeleteFromAllIndexes%
	Function EtDeleteIndexDescription%
	Function EtDeleteIndexPages%
	Function EtDeleteRecord%
	Function EtGetAutoInc%
	Function EtGetFields%
	Function EtGetKeyDef%
	Function EtGetKeyNames%
	Function EtRebuildIndex%
	Function EtResetDatabase%
	Function EtResetHeader%
	Function EtResetRecord%
	Function EtSetAutoInc%

	Function EtSoftLockDataRecord%
	Function EtStat%
	Function EtUnSoftLockAllDataRecords%
	Function EtUnSoftLockDataRecord%
	Function EtUpdateAllIndexes%
	Sub EtBuildKey
	Sub EtCalcOptimumPageSize
	Sub EtRetrieveByAddress
ETREEMOV.SUB	Function EtBOF%
	Function EtEOF%
	Function EtLOF&
	Function EtSeekRecord%
	Sub EtMoveFirst
	Sub EtMoveLast
	Sub EtMoveNext
	Sub EtMovePrevious
	Sub EtSeekEQ
	Sub EtSeekGE
	Sub EtSeekGT
	Sub EtSeekLE
	Sub EtSeekLT
ETREEUSR.SUB	Function EtTextComp%
	Sub EtClose
	Sub EtCloseAll
	Sub EtDelete
	Sub EtInsert
	Sub EtOpen
	Sub EtRetrieve
	Sub EtRetrieve2
	Sub EtSetIndex
	Sub EtUnLockCurrentRecord
	Sub EtUpdate
	Sub EtUpdate2
ETREEVAR.SUB	Function EtAddVarData%
	Function EtChangeVarData%
	Function EtGetVarPage%
	Function EtRetrieveVarSection%
	Function EtVarFillDesc%

	Function EtVarFindPage%
	Sub EtRetrieveVariable
	Sub EtRetrieveVariable2
	Sub EtRetrieveVariableByAddress
	Sub EtUpdateVariable
	Sub EtUpdateVariable2
	Sub EtUpdateVariableByAddress
ETSUNDRY.ASC	Function EtDiskSpace&
ETINDEX.ASC	Function EtDeleteDupe%
	Function EtDeleteKeyFromIndex%
	Function EtDeleteKeyFromNode%
	Function EtFindEndOfIndex%
	Function EtGetNodeStats%
	Function EtGetSiblings%
	Function EtIndexGetVarPage%
	Function EtIndexLock%
	Function EtIndexPageToFromList%
	Function EtIndexVarFindPage%
	Function EtInsertDupe%
	Function EtInsertKeyInIndex%
	Function EtInsertKeyInNode%
	Function EtMergeNodes%
	Function EtRotateNodes%
	Function EtSearchDupe%
	Function EtSearchIndex%
	Function EtSearchNode%
	Function EtSetParents%
	Function EtSplitNode%
	Sub EtIndexUnLock

Section 7
The Nitty-Gritty

7 (A) General Information

For those of you interested in the low-level details about how E-Tree Plus works, how it is organized, the structure of an E-Tree database file, or how b+trees are created and maintained, this section is for you. This is by no means "required reading."

7 (B) Low-Level Locks

This section gives a more in-depth look at how E-Tree Plus deals with locks and locking data records. All the details of locking that are covered in this section are handled automatically by the routines provided in the E-Tree Plus Library. This information is provided just so you'll know what we are up to.

There are a few clearly defined and simple protocols for accessing pages in the file and data on the pages. As long as these protocols are followed by all tasks trying to get at the data in these files, the data can be safely accessed.

Most of the protocols involve the use of 'semaphore' flag data items. Semaphores are integer values that are incremented to place a control on a resource and decremented to release the control from the resource. The purpose of the control can be to limit the number of tasks making use of a resource, as for a program that is licensed to only 5 users, or to limit the type of access to a resource, as in making a file region read-only. A resource is not completely free unless the semaphore has a value of zero.

For our file access protocols, every page we read or write will be controlled by a file-region lock on the page. The page cannot be accessed in any way until the lock succeeds. Once the lock succeeds, a semaphore may need to be checked before further action is taken. Some pages and data items use semaphores and some don't. In general, operations that alter data must check any semaphores used to control the data; operations that only read the data may not need to check the semaphore. If the semaphore is not zero you must not change any part of the data item controlled by the semaphore.

If you need to make the data item read only then increment the semaphore. When you are past the point of needing to ensure that the item didn't change then you must release your control of the semaphore by decrementing it. Semaphores must only be incremented or decremented when the page they are on has a file-region lock applied to prevent two tasks from attempting to change the semaphore at the same time.

Here are the steps:

<u>Raise a Semaphore</u>

1) Lock the page
2) Increment semaphore
3) UnLock the page

<u>Lower a Semaphore</u>

1) Lock the page
2) Decrement semaphore
3) Unlock the page

Since no other process can access the page while it is locked, the semaphore is guaranteed to be stable during the raise and lower steps. If a process successfully raises a semaphore, it can be reasonably sure that it will be safe to lower it since no other process will alter the data in the page if the semaphore is already set. This assumes, or course, that everybody plays by the rules.

Simply using semaphores and hard file-region locks does not make all multi-user data accesses a snap. You will have to balance the time a lock is in place and the impact of that lock against ensuring that access by another task doesn't preempt completion of your present task.

The simple case of reading the database doesn't actually require any file locking. However, other file operations will be carried out by other tasks as your task is reading each record. This makes it possible, and even likely, that the record you are about to read is locked by another task. This requires that you lock the record before you attempt to read it to ensure that you have access to it. If the lock fails you have the choice of waiting to retry the lock or abandoning the operation all together.

The more complex task of editing a record gives a more complete picture of the potential problems. When you need to edit a data record you will first read the current data, make changes to the data, and then write the data back after you are done editing. The edit cycle requires that you prevent others from updating the record while your are editing and that you be assured of read/write access when it comes time to write your changes to the file.

There are three methods of handling the situation. The first is to lock the file region and leave it locked for the duration. The second and third involve using semaphores to control write access to the data while you are processing your edits.

If you place a file-region lock on the data for the duration of the read-edit-update cycle, you will have absolute assurance that the record will not be altered during your editing and that the record will be available when you are ready to write the updates back to the file. However, the record will be unavailable for any kind of access by other tasks for the duration, no matter how long it takes.

If you lock the region, raise the semaphore, read the data, and unlock the region, other tasks can read the data without hesitation while you are processing your edits. As long as the other tasks do not need to write to the file this will work well until you need to write your changes back out. When it comes time to write your changes, you will have to lower the semaphore and then check it to see if anyone else raised it since you started your edit cycle. If it has been raised you must either abort the edit or sit in a loop checking the semaphore until it is zero before you can write your updates. This is not much different from having a hard-lock on the data, except you have to wait instead of others.

The only reasonable solution is to require all tasks that must make updates to data to check the semaphore first and not even begin the edit or update procedure unless the semaphore is clear. This allows any task that needs to read the data to have fairly free access to it and guarantees that a semaphore that can be raised will provide clear access to the data for updates until the semaphore is lowered. It still requires that tasks needing to alter or edit the data must wait until a lock (the semaphore in this case) on the data is released. Since the edit-cycle will tend to be less frequent that simple reads, this method will provide the best compromise between data availability and data protection.

At first glance this might make it seem that only binary semaphores are needed to protect data. However, there are situations where it is desirable to write-protect a data item that you are not going to be updating, but that needs to be controlled for a significant length of time. The answer is to use a semaphore that is incremented at the beginning of each access and decremented at the end. This lets several tasks access the data without worry of it changing until they are all done. Any process that needs to update the data would wait until the semaphore was clear and then hard-lock the data during the update.

7 (C) Global Data Used Internally by E-Tree Procedures

The file ETCOMMON.BI contains the declarations for all the global data used by the E-Tree Plus Library. These data are stored in a blank common block and must be available to all routines during the full course of the program. These data items should not be manipulated directly except by the E-Tree Plus routines and in accordance with their definitions. Improper alteration of this data can cause irreparable damage to any databases currently open or opened during the course of the program. Basically, you should leave this stuff alone.

EtManagerStatus%

Defines the current state of the E-Tree Manager

0	-	not initialized
1	-	initialized, databases are either currently opened or have never been opened
-1	-	initialized, databases were open but are all closed now

EtFileStructureID$

Used to tag databases with embedded version information and verify that an opened database is compatible with the currently executing version of the E-Tree Manager.

Defined in EtInitManager%()

EtMaxOpen%

The current maximum number of open databases allowed. Used to define the limits of the arrays used by the E-Tree Manager to manage open databases.

Defined by the argument passed to EtInitManager%()

EtCurrentOpen%

The Current number of databases that are opened using EtOpen().

If EtCurrentOpen% is greater than EtMaxOpen% then no more databases can be opened.

Incremented by EtOpen(), Decremented by EtClose(), Zeroed by EtInitManager%().

EtLockTimeOut&

This defines the standard system time out for lock collisions. A lock that fails will be retried until the number of clock ticks (18.2 per second) defined by this value expires.

The default value of 182 (10 seconds) is set by EtInitManager%().

Can be read using EtGetNetworkTimeOut&(); can be Set using EtSetNetworkTimeOut()

Even though this is a global variable that can be accessed directly, you should only use the routines provided to read or set it. Future versions of E-Tree Plus will always make the assumption that it is accessed using only those routines.

EtFileBufferHandle%

An E-Tree Memory Handle (see EtMemAlloc) that indicates a memory buffer used for the E-Tree Manager's file I/O space.

Defined and memory allocated in EtOpen() and EtCreate(). Don't dare alter this value under any circumstance.

EtFileBufferType%

Defines the type of memory that will be used for the E-Tree Manager's file I/O buffer. This defaults to zero which causes the buffer type to default to EMS if EMS is available or to DOS memory if EMS is not available.

If you need to change the type of memory used for the buffer you must set this variable before EtInitManager%() is ever called. Once EtInitManager%() has been called, either directly or through EtOpen(), THIS VALUE MUST NOT CHANGE or you will find your memory hopelessly corrupted and your databases hopelessly damaged.

EtScratchHandle%

An E-Tree Memory Handle (see EtMemAlloc) that indicates a memory buffer used by the E-Tree Manager as a scratch pad.

Defined and memory allocated in EtOpen() and EtCreate(). Don't dare alter this value under any circumstance. This buffer MUST be in DOS memory arena only because of the performance considerations involved when copying between EMS pages not simultaneously mapped into the page frame. Do not change the type buffer used in the code.

This buffer will be as large as the largest Data Record size of currently open databases or 512 bytes, whichever is larger.

EtTable$()

A one-dimensional string array that contains information used to manage each open database. Each element corresponds to a value of an E-Tree Handle.

Offset	Size (bytes)	Description
0	2	DOS file handle
2	4	Page size

6	4	Record size
10	2	Variable length flag
12	2	Reserved
14	2	Reserved
16	4	Current record address
20	4	Current record CRC of fixed-length data
24	4	Variable length data address
28	4	Variable length data length
32	2	BOF flag
34	2	EOF flag
36	4	Max fields allowed on variable length data page
40	8	Reserved

Use the routines provided in the E-Tree Library to access the data stored in this array.

Altering the data stored in this array will cause untold damage to your open databases.

EtFileNames$()

A one-dimensional array that holds the file names of all open databases.

EtKeyDesc$()

A one-dimensional string array that contains information used to manage the current index for each open database. Each element is 78 bytes in length and can be broken down as follows:

Offset	Length	Description
0	30	Index name
30	4	Magic number
34	2	Key number
36	4	Node
40	2	Node type
42	4	Previous leaf
46	4	Next leaf
50	2	Keys on leaf
52	4	Duplicate descriptor address
56	4	Offset into dupe page for section
60	4	Offset into section
64	4	Length of section
68	4	Section owner
72	4	Next section descriptor address
76	2	Section number

Use the routines provided in the E-Tree Library to access the data stored in this array.

Improperly altering the data stored in this array will cause untold damage to your open databases.

EtSoftLockedRecords&()

A two-dimensional array used to store the address of each data record that has a soft-lock (semaphore) applied. The first dimension corresponds to a handle value and the second to the data record slot as assigned using EtSoftLockDataRecord() (see below).

The value stored is the physical address of the data record that is locked. For variable-length records, this is the address of the descriptor block for the variable length data.

This array is, and should only be, accessed using the data lock routines provided in the E-Tree Library only.

Altering the data in this array could render data records in the database read-only until one of the EtReset routines is used to clean it up.

EtSoftLockedTypes%() >No Longer Used

A two-dimensional array used to store the type of each data record that has a soft-lock (semaphore) applied. The first dimension corresponds to a handle value and the second to the data record slot as assigned using EtSoftLockDataRecord() (see below).

The value stored is a zero for a fixed-length data record or a one (1) for a variable length data record.

This array is, and should only be, accessed using the data lock routines provided in the E-Tree Library only.

Altering the data in this array could render data records in the database read-only until one of the EtReset routines is used to clean it up.

EtSoftLockedCount%()

A one-dimensional array that stores the number of soft-locks applied for each open database.

This array is, and should only be, accessed using the data lock routines provided in the E-Tree Library only.

Altering the data in this array could render data records in the database read-only until one of the EtReset routines is used to clean it up.

7 (D) E-Tree Plus Database Structure

At the lowest level, almost all of the file I/O is performed on chunks of file called 'pages.' The size of a page in the file is defined when EtCreate() is used to create a new database and cannot be changed. Page sizes can range from 512 bytes to 65535 bytes. 65536 isn't used as the maximum because the standard DOS file I/O functions have a limit of 65535 bytes that can be read or written at one time.

Each page has a fixed amount of overhead required to manage its allocation and usage. This portion of the page overhead is always located in the first 6 bytes of the page:

Page Type ID (2 bytes)

-1	Free Page
0	Header Page
1	Field Information Page
2	Key Definition Page
3	Fixed-Length Data Page
4	Fixed Length data page with deleted records
5	Variable-Length Data Page
6	Index Page Internal Node
7	Index Page Leaf Node
8	Index Page Duplicates Node

Page Link (4 bytes)

Pointer to the next page of the same type

Each page type can also have other overhead required that is specific to the management of the type of information stored on the page. The format of each type of page is described below.

The Page Type ID is set whenever the page is allocated for a purpose or when the page is returned to the free page list. The only times the page type should change are when the page is being added to the free list or being taken from the free list (the free list also includes new allocations).

The header consists of one page allocated to hold information essential to the definition, control, and maintenance of the data and indexes. The header page is always the first page in the file. Most of the essential information is stored in the first 512 bytes of the header page.

The following are descriptions of the page types and their contents:

1. Header Page

OffSet	Size (bytes)	Description
0	2	Page ID (0)
2	4	Page Link (Always Zero)
6	4	Page size
10	2	File Status Semaphore 0 = No critical functions in progress >0 = Entire database is read-only
12	2	File open Count number of times the file has been opened Open increments, Close decrements Some functions should not be executed on a database that is open
14	2	Reserved
16	64	Version Information
80	2	Reserved
82	2	Reserved
84	4	Record size (fixed length portion only)
88	2	Number of fields in record
90	2	Variable length data flag 0= no variable length field in definition
92	4	First Record Definition Page
96	4	Number of Record Definition Pages If this number is zero then the Field Definitions all fit in the header <reserved> space and 'First Record Definition Page' is not used
100	4	Key definition page
104	2	Current number of indexes
106	4	Free Pages Root
110	4	Free Pages Count
114	4	Fixed Length Records First Page
118	4	Fixed Length Records Last Page
122	4	Fixed Length Records Page Count
126	4	Data-pages-with-deleted-records root
130	4	Data-pages-with-deleted-records count
134	4	Variable Length Pages Root
138	4	Variable Length Pages Count
142	4	Number of active records
146	4	Maximum Number of Variable Records Per Page This value determines the amount of space allocated for the descriptor table in a variable length data page.

		Defaults to about 5% of the page size and is determined by the formula: PageSize \ 320. This value can be changed if, and only if, there have been no data records of any kind added to the database. It is set when the record definition is added to the database.
150	42	Reserved
192	PageSize-192	Reserved for record field definitions

2. Fixed-Length Data Page

OffSet	Size (bytes)	Description
0	2	Page ID (3)
2	4	Next Page Link
6	4	Previous Page Link
10	4	Deleted-Records page link Next page with deleted records
14	4	Empty Records Root Offset of first empty record in page 0FFFFFFFFh = no more empty record space
18	2	Reserved

The following structure is repeated for each active record on the page. The offsets are relative to the start of the record slot. The record slots start at offset 20 into the page and repeat every RecordSize+16 bytes:

0	4	Address of Variable length data (0 if none)
4	2	Semaphore for Data Record (>0 = read only)
6	6	Reserved
12	4	Record CRC
16	RecordSize	Fixed Length Record Data

The following structure is repeated for each unused record slot on the page. The offsets are relative to the start of the record slot. The record slots start at offset 20 into the page and repeat every RecordSize+16 bytes:

0	4	Offset of Next Empty Record on the page 0FFFFFFFFh = no more empty record space

3. Variable-Length Data Page

OffSet	Size (bytes)	Description
0	2	Page ID (5)

2	4	Next Page Link
6	4	Reserved
10	4	Empty Space on this page
14	??	Data descriptors

The data descriptors for the variable length data define offsets into the page for the actual start of the variable length data. The fixed-length data stores the address of one of the descriptors when the record has variable length data defined. This allows the variable length data to change size and location within the page without affecting the fixed length data page.

The space set aside for the data descriptor table is determined by the value stored in the header page field Maximum Variable Records Per Page. Multiply that value by 16 to determine the total space. Each data descriptor has the following layout:

OffSet	Size (bytes)	Description
0	4	Address of Fixed-Length Record Owner
4	2	Semaphore for variable data
6	4	Next descriptor for this data
10	4	Offset in the page to data
14	4	Length of Data
18	4	CRC of data
22	4	Total length of data
26	2	Section number for this page
28	4	Reserved

Each piece of variable length data is allocated in a contiguous block starting at the offset stored in its data descriptor.

As variable length fields are deleted, added, or change size the data on the page is moved around to make sure that all the unused space on the page is at the end. The descriptors themselves never move.

4. Index Pages

Index Internal Node

OffSet	Size (bytes)	Description
0	2	Page ID (6)
2	4	Previous Page
6	4	Next Page Link
10	4	Parent node
14	2	Keys active
16	2	Pointers active

18	14	Reserved
32	Varies	Keys
Varies	Varies	Pointers

Index Leaf Node

OffSet	Size (bytes)	Description
0	2	Page ID (7)
2	4	Previous Page
6	4	Next Page
10	4	Parent node
14	2	Keys Active
16	2	Pointers Active
18	4	Previous leaf (smaller key values)
22	4	Next leaf (greater key values)
26	6	Reserved
32	Varies	Keys
Varies	Varies	Pointers
Varies	Varies	Duplicate Pointers

Duplicates Page

OffSet	Size (bytes)	Description
0	2	Page ID (8)
2	4	Previous Page
6	4	Next Page
10	4	Free Space on Page
14	Varies	Descriptors

The duplicates nodes have the same format and are managed just like variable-length data pages. The data sections hold arrays of record addresses that have identical key values.

5. Key Definition Page

OffSet	Size (bytes)	Description
0	2	Page ID (2)
2	4	Next Page Link
6	4	Number of free bytes at end of page

The following structure is repeated for each key defined. Offsets shown are relative to the beginning of the structure.

0	30	Index Name

30	4	Index Root Node Page Number
34	1	Duplicates allowed (True = yes)
35	2	Number of parts in key
37	4	Key length
41	2	Minimum number of keys for internal nodes
43	4	Index list - Last Page
47	4	Index list - Last Page
51	2	Semaphore
53	4	Magic Number
57	7	Reserved.

The following structure is repeated for each key segment defined for this index. Offsets are relative to the beginning of the structure:

0	4	Offset of first byte of segment in record
4	4	Length of segment
8	2	Data type (for comparisons)
10	2	Direction (0 = ascending, NOT 0 = descending)
12	2	Non-Modifiable (NOT 0 = non-modifiable)
14	2	Reserved

6. Field Information Page

OffSet	Size (bytes)	Description
0	2	Page ID (1)
2	4	Next Page Link
6	4	Number of free bytes at end of page
10	22	<reserved> Pad to 32 bytes
32	PageSize&-32	Definitions (48 bytes each)

The following structure is repeated for each field defined

0	30	Field name
30	4	Field length
		0 .. page size if variable length == max size
34	2	Field type
		0 = Variable Length
		1 = unsigned binary
		2 = signed binary
		3 = autoincrement (4-byte signed)
		4 = integer (2-byte signed)
		5 = long integer (4-byte signed)
		6 = IEEE single precision (4 bytes)
		7 = IEEE double precision (8 bytes)
		8 = IEEE extended precision (10 bytes)
		9 = PB BCD Fixed Point (8 bytes)
		10 = PB BCD Floating Point (10 bytes)

		11 = MBF single (4 bytes)
		12 = MBF double (8 bytes)
		13 = MS PDS Currency (8 bytes)
		14 = ASCII string case insensitive
		15 = ASCII string case sensitive
36	4	Last autoincrement value
40	8	Reserved

A variable length field must be the last field defined and there can be only one per record.

7. Free Pages

OffSet	Size (bytes)	Description
0	2	Page ID (-1)
2	4	Next Page Link

7 (E) B+Tree Indexes

This section describes the internal structure and workings of the E-Tree Plus B+Tree indexing system. The stuff here isn't necessary for using E-Tree Plus. You can ignore this section unless you are really interested or really bored. For the really bored, there are seemingly endless details on how B+Trees work. For the really interested, all the algorithms used in our indexing system are described here; nothing is left as 'an exercise for the reader'.

1. B+Trees

The indexing system chosen for E-Tree Plus is a standard B+Tree method. The B+Tree method was chosen over other methods (particularly the very similar B-Tree method) because it lends itself to the structure of the E-Tree Plus file system and offers advantages for frequently used database functions.

The definition of a B+Tree is a fully balanced tree structure containing nodes which can be either 'internal nodes' or 'leaf nodes' with all key values located on leaf nodes and only leaf nodes pointing to data records. Each node in the tree, leaf or internal, must contain at least enough keys to make it half full. After every insertion and deletion the tree is rebalanced (if necessary).

The constraints placed on a standard B+Tree (from Knuth) are:

i) Every node has <= MAX descendents
ii) Every node, except the root and leaf nodes, has => MAX\2 descendents
iii) The root must have at least 2 descendents, unless it is a leaf
iv) All leaf nodes are at the same level
v) A non-leaf node with k descendents has k-1 keys

The B+Tree and the B-Tree methods are very similar, differing only in a constraint placed on the location of data record pointers. A B-Tree allows data record pointers on any node while a B+Tree requires that data record pointers be located only on leaf nodes. This added constraint has a few side affects:

1) All tree searches must travel to the lowest level to find the key desired

2) Since all data record pointers are on leaf nodes, and all leaf nodes are on the same level (definition of balanced), sequential access of keys is simplified greatly

3) Since internal nodes do not store data record pointers, it is possible to get more keys per node, reducing the depth of the tree.

Side affect number (1) is actually a disadvantage for seeking a particular key; however, since the tree is usually very shallow it is not that bad. A search in a B-Tree would stop at the first occurrence of the desired key in the tree instead of continuing to the last level. Side affect (3) helps to make side affect (1) even less significant. Since internal nodes have less over head the leaf nodes, a B+Tree a given node size node can contain considerably more keys on internal nodes that a B-Tree with the same node size. The more keys that can be stored on a node the fewer levels the tree will require.

Side affect (2) improves the speed of sequential access by guaranteeing that, at most, only one extra node will need to be read to find the next key in sequence. In a B-Tree it is possible to require climbing down the tree all the way to the root and then back up to reach the next key in sequence.

The E-Tree plus implementation of the B+Tree requires that the tree itself store only unique key values. If an index is defined to allow duplicates, the duplicates are stored separately from the index. This makes searching and balancing the tree much easier and faster.

E-Tree Plus uses some variations on the standard B+Tree to allow some added flexibility in defining the tree parameters to optimize disk space or speed of insertions and deletions, to provide buffer space on each node, and to limit the worst case depth of the tree.

2. Complete E-Tree Constraints and Definitions:

A 'tree' structure is formed of 'nodes'. Each node in the tree, except for a special one called the 'root' has exactly one 'parent'; the root node has no parent. Each node has zero or more 'child' nodes. A node that does not have any child nodes is called a 'leaf'. A node that has one or more child nodes is called an 'internal' node. Each child node is itself the root of a sub-tree that branches from the parent node [paraphrased from Elmasri].

Each node in the tree has a 'level' or 'depth' that is defined as the number of parent nodes between it and the root node (counting the root node itself). The root node is at level zero, the root's child nodes are at level 1, and their child nodes are at level 2, etc.

Two nodes are 'siblings' is they have the same parent. Two nodes are 'cousins' if they are on the same level in the tree, but have different parents.

- A leaf node points to data records.
- An internal node points to other tree nodes.
- Each node can hold a MAX number of active keys.
- Each node has room for at least one inactive key and pointer.
- The smallest allowable value for MAX is 3 (therefore the smallest node will have room for 4 keys and pointers).
- Each leaf node can have MAX pointers.
- Each internal node can have MAX+1 pointers.
- Each internal node, except the root node, must have at least MIN keys, where MAX\2 <= MIN <= MAX.
- For leaf nodes MIN = MAX\2.
- For the root node MIN = 1.
- Each internal node must have at least MIN+1 pointers.
- Each internal node with K keys must have K+1 pointers.
- Each leaf node with K keys must have K pointers.

- Keys on leaf nodes must exist in the database.
- Keys on internal nodes need not exist in the database.
- Only unique keys are stored in the tree; duplicate keys are maintained in 'duplicate bucket' nodes separate from the actual tree.
- Leaf nodes are maintained as a doubly linked list to provide in-order traversal in both directions.

3. Insertion into the B+Tree

Inserting a key into the tree is fairly simple. You need to find the leaf node that should hold the key and then insert the key into that leaf node. If the leaf node was not full before the insertion of the new key then the process is finished.

If the leaf node was full before the insertion (it now has one too many keys) it will have to be split and the tree balanced. Splitting a node requires only that a new node be allocated and the keys (including the new key) be distributed between the old node and the new one with (MAX+1)\2 keys going to the left hand node. If there is an odd number of keys then the old node, on the left, will get the extra key.

If the leaf nodes are split, the balancing act must be carried up through all the parents that lead to that leaf until one is not split or you run out of parent nodes (the root gets split).

The rightmost key from the left-hand node (after the split) is inserted into the parent. If the split nodes were leaf nodes then the key is a copied from the leaf otherwise the key is moved from the child node. If the key inserted into this parent overflows the parent node then it too must be split. The process of splitting nodes and carrying keys up the tree is repeated until a node is not split or the root is split.

If the root node is split (whether it was a leaf or an internal node), a new node is allocated and it becomes the new root of the tree. The key carried from the split becomes the only key on the root. The only way the tree can become a level deeper is if the balancing of the tree splits the root to create a new root.

4. Deleting from the B+Tree

Since all keys are held at the leaf level, the key to be deleted must be found there before it can be deleted. Once it is found it must be deleted from the leaf node it is on. As long as the deletion does not reduce the number of keys on the node to less than MAX\2, that is all the work that needs to be done. If the leaf node holds fewer then MAX\2 keys after the deletion, the tree must be balanced.

The leaf nodes and the internal nodes require slightly different algorithms for balancing. The leaf nodes are balanced either by combining the keys on a node with the keys on one of its siblings and then evenly distributing them across the two nodes (called a 'rotation' in some books), or by merging the two nodes into a single node. Internal nodes are either combined (just like leaf nodes) or a key is stolen from the parent node to provide enough keys to support a merge. The differences are in how the merge process is carried out for the two types of nodes. Because internal nodes have an extra pointer that must be considered, and because internal nodes can have fewer than MIN keys (if MIN > MAX\2), extra steps must be taken to ensure that the keys are distributed correctly.

If one of the siblings of the node that the key was deleted from (a leaf) has > MAX\2 keys, then the two nodes can be combined and the keys evenly distributed between the two (if there are an odd number, the left node gets the extra). Since the leaf nodes are siblings, they must have a parent between them. The key in the parent node that separated the two siblings must then be updated with the largest key (the rightmost) on the left sibling. This must be done to ensure that the parent has a correct key (=> all keys to the left) for future searches to work. If neither of the siblings of the node has > MAX\2 keys then the node is merged with a sibling leaf removing one node from the tree. The left sibling is used if it exists, although either can be used. The parent node is updated by deleting the parent key that was between the two nodes and shifting all the larger keys to the left to replace it.

Once the leaf nodes have been balanced a trip must be made toward the root of the tree to check the parent nodes to be sure any that were updated are also balanced. The balancing of the leaf and internal nodes will never affect more than the two sibling nodes and the parent node used in rearranging the sub-tree. This trip is unnecessary if the balancing of the leaf nodes did not require any keys to be removed from the parent. The only way a key would be removed from the parent is if sibling leaf nodes are merged. Likewise, as the trip toward the root is made it can be stopped if an internal node is balanced without reducing the size of its parent node.

The parent node is made the current node. If is has at least MIN keys and pointers then the job is done. If not, and it is not the root, then the node is either combined or merged with one of its siblings. Its siblings are checked to see if one is > MIN. If one is found > MIN then the key separating the siblings (in the next parent node) is borrowed to help in combining the nodes. A key must be borrowed for internal nodes since they have one pointer more than leaf nodes that must be accounted for. The key from the parent is placed as the last key in the left sibling and then the two nodes can be combined, and the keys distributed between them so that the left sibling has at least MIN keys. The key in the parent node that separated the two siblings must then be updated with the largest key (the rightmost) on the left sibling. This must be done to ensure that the parent has a correct key (=> all keys to the left) for future searches to work.

If neither of the siblings are > MIN then a key is stolen (as opposed to borrowed) from the parent node. The keys to the right of the stolen key on the parent (and their pointers) are shifted left to replace the stolen key. The parent node will always have at least 1 key before the theft (MIN for the root is 1). There will always be a key to steal and the parent node will always be checked after the current node is balanced. The two siblings are merged so that the left sibling has at least MIN keys and the right sibling has whatever is left over. If the right sibling is reduced to zero keys then it is released. The parent node's pointers are already correct from the shifting done during the theft of the key. If the right sibling is not reduced to zero then the rightmost key on the left sibling must be moved to the parent to become the key between the siblings. The whole business is repeated on the next level toward the root. If the next level is the root level and the key stolen was the last, then the root node is released and the current node becomes the root node for the tree.
The index can only shrink by a level when the last key is removed from the root node. It can only grow a level when an insert propagates node splitting up to the root node and it is split to create a new root and a new level.

5. Notes

Allowing the minimum number of keys on an internal node to be something other than MAX\2 will allow the nodes to hold more keys (on average), tend to reduce the depth of the tree, and tend to make search for a key faster. It will not affect the algorithms used to balance the tree or their efficiency. However, it will tend to cause the tree to need balancing on a higher percentage of insertions and deletions, and to cause the balancing process to affect more levels each time. The larger the value of MIN the larger this affect will be. It is assumed that the advantages will be weighed against the disadvantages of increasing the value of MIN for each application. There are legitimate cases where using a MIN value as high as MAX are warranted and even recommended. Any application that has large node sizes with small keys (higher order indexes), where disk space is at a premium over insertion and deletion time, or where the index will rarely or never have keys inserted or deleted is a good candidate for a larger value of MIN. An indexed help system is a good example. The content of the help text is not likely to change once the product ships, and the space required for the index should be minimized. Using MIN=MAX will guarantee the smallest number of nodes are used by the index.

6. Pseudocode Examples

In these descriptions keys and pointers are numbered 1..MAX.

A node has q keys active.

- n.K() -> key array on a node (leaf or internal)
- n.P() -> pointer array (nodes) on an internal node
- n.Pr() -> pointer array (to records) on an leaf node
- (K,n) -> a key and pointer pair
- S() -> node stack
- SP -> node stack pointer (points at next available element)

Search for K

```
n = indexroot                'start at the root

read n
while n is not a leaf
                             'search for a leaf that does (or should)
                             '   contain the value K

                             'see if K is on this node or not
  If K <= n.K(1) then            '   is it <= all the keys in the node?
      n = h.P(1)

  Elseif K > n.K(q) then '   is it > all the keys in the node?

      n = h.P(q)

  Else                       'The trail leads through this node

      Search node n for an entry i such that n.K(i)<K<=n.K(i)
      n = h.P(i)
  End if
  s(sp)=n
  sp=sp+1
  read n
loop

'The above search will always end up at a leaf that should contain K.

Search leaf node n for an element i such that n.K(i) = K
if found then
```

```
    record = n.Pr(i)
  else
    K is not in the index
  end if
```

Insert K

```
Search tree for key K

if found then
   If duplicates allowed then
       insert on duplicate bucket node
   else
       cannot insert duplicate
   end if

Else                                'insert K at n

   if leaf node n is not full then
       insert K and pointer into n

   else                         'node is full -- must be split
       p = q
                                'nodes always have an extra space
       insert K and data record pointer into n
       p = p + 1
       allocate a new, empty, leaf node

       j = (MAX+1)\2        'the lower half goes to n (with j)
       move keys j+1 .. p to new node

       K = n.K(j)               'the original K is where it should be
                                'now we propagate the split up the tree

                                'n.K(j) will be replicated in parent

       finished = false         'a flag

       do
           If SP = 0 then'empty stack means no parents waiting
                        '  need to make a new level

                 root = a new, empty internal node

                 root.P(1) = n     'root has one key, the pointers two
                 root.K(1) = K     '    n and new for the branches
                 root.P(2) = new

                 finished = true

           Else                 'parents in the trail to be fixed

                 SP = SP-1      'pop a parent off the stack
                 n = S(SP)      'it is an internal node!

                 insert (K,new) into position

                 if n is not full then'K is now the largest value
                                      'on the left node after the
                                      'split. If the split nodes
                                      'were leaves then this is
                                      'a Copy, else it is a Move

                    finished = true

                 else                 'split internal node

                    new = a new internal node

                    p = MAX+1      'new has one more than the max
                    j = MIN
                    node n gets temp.K(1) to temp.K(j)
                         and temp.P(1) to temp.P(j+1)

                    move temp.K(j+1) to temp.K(p)
                         and temp.P(j+1) to temp.P(p+1) to new

                    K = n.K(j)        'K(j) is moved to parent

                 end if
            end if
        loop until finished
    end if
end if
```

Delete K

```
Search tree for key K

if not found then              'Key value doesn't exist -- cannot delete
  error -- key not found
  finished = true

Else                           'delete K from n at i

  If there are duplicates of this key then
      delete a duplicate

  else
      delete key from leaf node

      q = q - 1
      p = q

      Finished = true       'assume it will go quickly

  'when looking for siblings of any kind of node it is only
  'necessary to consider true _siblings_, not cousins (no
  'need to go off the further than the parent node in search
  'of siblings.  if the node is the rightmost or leftmost
  '(it will never be the only unless it is the root) just
  'consider the sibling-off-the-end to be a node with q=0.

  if p < (MAX\2) then       'leaf has too few keys
                            'remember, the root has MIN=1

      if left sibling q is > MAX\2 then
          combine with left sibling,
               re-distribute keys, and update parent key

      elseif right sibling q is > MAX\2 then
          combine with right sibling,
               re-distribute keys, and update parent key

      else
          if left sibling q > 0 then
               merge with left sibling
          else
               merge with right sibling
          end if

          reduce parent node by removing parent key (shift left)

          finished = false
      end if

  'work back to the root checking parents for underflow

  do until (node = root) or finished
      finished = true
      if Parent.q < MIN then    'remember root MIN=1
          If Parent == root then'root was emptied above
               current node becomes the root
          else
               n = parent

               if left sibling q is > MIN then
                  combine with left sibling, re-distribute,
                       and update parent key

               elseif right sibling q is > MIN then
                  combine with left sibling, re-distribute,
                       and update parent key
               else    'If the parent is not the root it will have at
                       '   least MIN
                       'if the parent is the root it will have at least 1
                       '   remember, the root has MIN=1

                  if left sibling q > 0 then
                       steal from parent to merge with left sibling
                  else
                       steal from parent to merge with right sibling
                  end if

                  finished = false

                  'if the parent was reduced to < min or
                  'the root was emptied it will be caught on the next
                  'iteration of the loop
               end if
          end if
     end if
  loop
end if
```

7. Subroutine Algorithms

Combine 2 sibling nodes, redistribute keys, and update parent

This routine must not be used unless at least one of nodes has > MIN keys and the other has => MIN-1 keys this is to make certain both nodes that result will have =>MIN keys.

```
Lq = keys in left sibling
Rq = keys in right sibling

If siblings are leaves then
  j = (Lq + Rq + 1)\2        'this many keys will go in left sibling
                             'adding one ensures that the left node
                             'will get the extra if there is an odd
                             'number of keys
else
  j = MIN
end if

j1 = j-Lq                        'this many keys to copy from right sibling
                             '   to the left

                             'L gets keys (and pointers) 1 to j from the
                             'combined group
                             'R gets keys j+1 to (Lk+Rk) from the
                             '   combined group
copy keys R.K(1..j1) to L.K(Lq+1..Lq+j1)
fix R so all remaining keys start in first position

if the siblings are leaf nodes then
  the parent key (the one between the siblings) gets a copy of L.K(j)
end if
```

Merge two leaf nodes

```
Copy all keys and pointers from right node into the left starting at the
first unused slot

Release the right node

Shift all keys in the parent node one position to the left to replace the
parent key that separated the two nodes and is no longer needed.
```

Merge (or combine and split) sibling internal nodes

```
Steal the parent key between the siblings from the parent node.

Shift the remaining keys in the parent node to the left to cover the stolen
key.

Put the key taken from the parent in the first unused key slot on the left
sibling node (could be the extra slot)

If all the keys from both nodes (and the stolen key) will fit on one node
  move all the keys from the right node to the left

ElseIf left sibling q < MIN (after parent key added) then
  move enough keys and pointers from the right sibling
  to make the left q = MIN
end if

if the right sibling q = 0 then       'a true merge
  release the right sibling
else                                  'a combine with a steal
  move the rightmost key on the left sibling up to the parent
  (move it, don't copy it!)
end if
```

References

The Art of Computer Programming Vol. 3, Knuth. Pgs. 471-479
Fundamentals of Database Systems, Elmasri, Navathe. Pgs. 120-125
Data Structures Using C, Tenebaum, et al. Pgs. 409-448

8. Internal Node

Each key is a fixed size defined by the index key description. The pointers are the page numbers of the children of the node. The keys and pointers are logically related like:

P1,K1,P2,K2,P3, ... Kn,Pn, Kx,Px

where the Ps are pointers and the Ks are keys.

Each key, Ky, has children at Py and P(y+1). There is always one extra key and pointer slot to allow for insertion during a split of a child node

9. Leaf Node

Each key is a fixed size defined by the index key description. The pointers are addresses to data records. The keys and pointers are logically related like:

K1,P1,K2,P2,... Kn,Pn, Kx,Px

where the Ps are pointers and the Ks are keys. There is always one extra key and pointer slot to allow for insertion.

The duplicate pointers is the address of a descriptor in a duplicates node. If the address is zero then there are no duplicates for that key.

10. Duplicates Node

Duplicate nodes look just like a variable length pages except for the ID and the Owner field in the descriptor blocks. The Page ID is 8 for duplicates and 5 for variable pages. The address of the beginning of the index description block on the index definition page is stored as the owner (instead of a data record address). Other than these small differences, the pages are handled exactly as if they were normal variable length data pages, some the same routines are used to manage them.

The index leaf node stores the address of one of the descriptors when duplicates of a key are inserted into the index.

Each section of data on the duplicate node contains an array of data record addresses for records with identical keys in this index. The array can be many, many sections in length and span many pages. Obviously, processing duplicates will degrade the performance of the index. If your database uses a key which could have a large number of duplicates, you may wish to append a dummy AutoIncrement field to the end of the key definition. This will reduce, if not eliminate duplicates and enhance performance.

Appendix A
Differences Between E-Tree, Btrieve and PDS ISAM

This section provides charts showing E-Tree Plus routines along with similar routines in PDS ISAM and Btrieve. In each case, the routine that provides the most similar operation to the E-Tree Plus routine is shown. Btrieve routines are shown with the name and the operation code. But first we'll discuss the differences between Btrieve's record locking strategy as compared to E-Trees.

Each E-Tree Plus routine that requires you to specify a lock type, uses a separate parameter in the CALL syntax to indicate the lock. The LockFlag% parameter can have a value ranging from 0 to 5 to indicate:

0) No locks needed
1) Lock to access record, unlock when finished (~~system~~ Inf timeout)
2) Lock to access record, leave locked (~~system~~ Inf timeout)
3) Lock to access record, unlock when finished (~~infinite~~ sys timeout)
4) Lock to access record, leave locked (~~infinite~~ sys timeout)
5) Assume the record is locked, unlock it when finished

All E-Tree Plus locks can be placed and released without requiring a explicit call to a lock or unlock function. For instance, a lock can be placed when EtRetrieve() is used to read a record (LockFlag% = 2 or 4) and then released with EtUpdate() after the record is edited (LockFlag% = 5).

Btrieve allows 'single locks' and 'multiple locks' with 'wait' and 'no wait' options to be used with all of its Get and Seek operations. The type lock to use is indicated by using a different operation code (you add 100, 200, 300, or 400 to the basic Get or Seek operation code). Any lock that is placed using one of these methods must be released using the Btrieve 'Unlock' operation (# 27).

Btrieve 'single' locks are equivalent to E-Tree Plus Lock/Unlock types (LockFlag% = 1 or 3). Btrieve 'multiple' locks are equivalent to E-Tree Plus Lock/Leave Locked types (LockFlag% = 2 or 4).

Btrieve's 'no wait' lock option is similar to the E-Tree Plus system timeout option (LockFlag% = 1 or 2), except E-Tree Plus lets you defined a short period of time for a time out delay and Btrieve simply attempts the lock one time. Btrieve's 'wait' lock option is similar to the E-Tree Plus infinite timeout option (LockFlag% = 3 or 4) in that the lock is retried until it succeeds.

User-Level Routines

E-Tree Plus	PDS ISAM	Btrieve
EtBOF	BOF	status code from Get
EtClose	Close	Close (1)
EtCreate	Open	Create (14)
EtCreateIndex	CreateIndex	Create Supplemental Index (31)
EtDelete	Delete	Delete (4)
EtDeleteIndex	DeleteIndex	Drop Supplemental Index (32)
EtEOF	EOF	status code from Get
EtInsert	Insert	Insert (2)
EtInsertVariable	n/a	Insert (2)
EtLOF	LOF	Stat (15)
EtMoveFirst	MoveFirst	Get Key First (62) Step Key First (83)
EtMoveLast	MoveLast	Get Key Last (63) Step Key Last (84)
EtMoveNext	MoveNext	Get Key Next (56) Step Key Next (74)
EtMovePrevious	MovePrevious	Get Key Previous (57) Step Key Previous (85)
EtOpen	Open	Open (0)
EtRetrieve	Retrieve	Get Direct (23)
EtRetrieveVariable	n/a	Get Direct (23)
EtSeekEQ	SeekEQ	Get Key Equal (55)
EtSeekGE	SeekGE	Get Key Greater or Equal (59)
EtSeekGT	SeekGT	Get Key Greater (58)
EtSeekLE	n/a	Get Key Less Than or Equal (61)
EtSeekLT	n/a	Get Key Less Than (60)
EtSetIndex	SetIndex	specified in parameter block
EtTextComp	TextComp$	n/a
EtUpdate	Update	Update (3)
EtUpdateVariable	n/a	Update (3)

High-Level Routines

E-Tree Plus	PDS ISAM	Btrieve
EtGetIndex$	GetIndex$	n/a
EtGetCurrentRecordAddress	n/a	Get Position (22)
EtGetCurrentRecordCRC	n/a	n/a
EtGetFileHandle *	n/a	n/a
EtGetPageSize	n/a	Stat (15)
EtGetRecordSize	LEN	Stat (15)
EtGetVariableFlag	n/a	Stat (15)
EtInitManager	Load PROISAM library	Load Btrieve TSR
EtInitManagerDefault	Load PROISAM library	Load Btrieve TSR

Middle-Level Routines

E-Tree Plus	PDS ISAM	Btrieve
EtAddFields	n/a	n/a
EtAddKeyDef	n/a	n/a
EtAddToAllIndexes	n/a	n/a
EtBuildKey	n/a	n/a
EtGetFields	n/a	n/a
EtGetKeyDef	n/a	Stat (22)
EtGetKeyNames	n/a	n/a
EtLockDataRecord	n/a	Get or Step (+100, +200, +300, or +400)
EtRebuildIndex	n/a	n/a
EtResetDatabase	n/a	Reset (28)
EtResetHeader	n/a	Reset (28)
EtResetRecord	n/a	Reset (28)
EtRetrieveByAddress	n/a	Get Direct (23)
EtRetrieveVariableByAddress	n/a	n/a
EtStat	n/a	Stat (22)
EtUnLockAllDataRecords	n/a	Unlock (27)
EtUnLockDataRecord	n/a	Unlock (27)
EtUpdateAllIndexes	n/a	n/a

Appendix B
Routine Syntax Summary

The following is a syntax summary of the user-level, high-level and some select lower-level routines included in E-Tree Plus.

```
Declare Function EtBOF% ( Handle% )
Declare Sub      EtClose( Handle% )
Declare Sub      EtCloseAll ()
Declare Sub      EtCreate  ( FileName$, Pagesize&, PreAllocation&,_
                         RecordInfo() as EtRecordInfoType,_
                         Maxkeys%, Handle%, Status% )
Declare Sub      EtCreateIndex ( Handle%, IndexName$, Unique%, _
                         Columns$(), Status% )
Declare Sub EtCreateIndex2      (   Handle%, IndexName$, Desc() AS_
                         EtKeyDescType, Segments%, Unique%, _
                         Status% )
Declare Sub      EtDelete ( Handle%, Status% )
Declare Sub      EtDeleteIndex ( Handle%, IndexName$, Status% )
Declare Function EtEOF% ( Handle% )
Declare Function EtFarAddress& ( Segment&, Offset& )
Declare Function EtFarToSegment& ( FarAddress& )
Declare Function EtFarToOffset& ( FarAddress& )
Declare Function EtFileExist% (FileName$)
Declare Function EtGetCurRecAdr& ( Handle% )
Declare Function EtGetCurrentRecordCRC& ( Handle% )
Declare Function EtGetCurVarAdr& ( Handle% )
Declare Function EtGetCurVarLen& ( Handle% )
Declare Function EtGetFileHandle% ( Handle% )
Declare Function EtGetIndex$ ( Handle% )
Declare Function EtGetLockTimeout& ()
Declare Function EtGetPageSize& ( Handle% )
Declare Function EtGetRecordSize& ( Handle% )
Declare Function EtGetVariableFlag% ( Handle% )
Declare Function EtInitManager% ( MaxOpenFiles% )
Declare Function EtInitManagerDefault% ()
Declare Sub      EtInsert ( Handle%, Buffer&, LockFlag%, Status% )
Declare Function EtLOF& ( Handle% )
Declare Sub      EtMakeRecordCurrent( Handle%, Address&, Status% )
Declare Sub      EtMoveFirst ( Handle%, Status% )
Declare Sub      EtMoveLast ( Handle%, Status% )
Declare Sub      EtMoveNext ( Handle%, Status% )
Declare Sub      EtMovePrevious ( Handle%, Status% )
Declare Sub      EtOpen ( FileName$, Handle%, Status%)
Declare Sub      EtRetrieve ( Handle%, Buffer&, LockFlag%, Status%)
Declare Sub      EtRetrieve2 ( Handle%, Address&, Buffer&, _
                         LockFlag%, Status% )
Declare Sub      EtRetrieveVariable ( Handle%, Buffer$, LockFlag%,_
                            Status% )
Declare Sub      EtRetrieveVariable2 ( Handle%, Address&, Buffer&,_
                             BufferLen&, LockFlag%, Status% )
Declare Sub      EtSeekEQ ( Handle%, KeyBuffer&, Status% )
Declare Sub      EtSeekGE ( Handle%, KeyBuffer&, Status% )
Declare Sub      EtSeekGT ( Handle%, KeyBuffer&, Status% )
Declare Sub      EtSeekLE ( Handle%, KeyBuffer&, Status% )
Declare Sub      EtSeekLT ( Handle%, KeyBuffer&, Status% )
Declare Sub      EtSetIndex ( Handle%, IndexName$, Status% )
Declare Function EtTextComp%( A$, B$ )
Declare Sub      EtSetLockTimeout ( T& )
Declare Sub      EtUnlockAllRecords ( Handle% )
Declare Sub      EtUnlockCurrentRecord ( Handle% )
Declare Sub      EtUpdate ( Handle%, Buffer&, LockFlag%, Status% )
Declare Sub      EtUpdate2 ( Handle%, Address&, Buffer&, _
                           LockFlag%, Status% )
Declare Sub      EtUpdateVariable ( Handle%, Buffer$, LockFlag%,_
                           Status% )
Declare Sub      EtUpdateVariable2 ( Handle%, Address&, Buffer&, _
                           BufferLen&, LockFlag%, Status% )
```

Appendix C
Error Codes

All routines that return a status code, either as an argument to the routine or as the value of a function, will return a zero to indicate that the routine completed successfully or a non-zero value to indicate and error code. All errors greater than zero correspond to DOS errors. DOS non-critical errors are returned as the error value from DOS plus 100. DOS critical errors are returned as the value DOS returns. E-Tree Library errors are returned as negative values. In addition some functions have specific negative return values that relate only to the specific function. Functions that do not return the complete error code (some that return other values) will return a 'failed' status and set an error code that can be read using the EtErrCode() function.

E-Tree Library Errors

Named constants are in the file ETREE.BI.

000 Success
-001 UnKnownFileError
-002 BufferTooSmall
-003 InvalidPageSize
-004 InvalidDatabaseName
-005 OverWriteAttempt
-006 InvalidCRC
-007 InvalidField
-008 LockFailed
-009 IncorrectVersion
-010 TooManyOpen
-011 InvalidPageType
-012 LockedBySemaphore
-013 NotInitialized
-014 InvalidHandle
-015 MemoryAllocFailed
-016 InvalidPage
-017 FreePageFailed
-018 LinkFailed
-019 CorruptLinkage
-020 MemoryReleaseFailed
-021 InvalidFieldLength
-022 TooManyKeys

-023 InvalidIndexName
-024 DuplicateIndexName
-025 InvalidOffset
-026 InvalidSegment
-027 LockTimedOut
-028 EmptyList
-029 DiskFull
-030 TooManyLocks
-031 LockNotFound
-032 DataRecordCorrupt
-033 DuplicateKey
-034 InvalidAddress
-035 NoModifyKeyFault
-036 IndexNeedsRepair
-037 RecordConflict
-038 SeekFailed
-254 InvalidOperation
-255 FeatureNotImplemented

Dos Critical Errors

001 Unknown unit for driver
002 Drive not ready
003 Unknown command given to driver
004 Data error (bad CRC)
005 Bad device driver request structure length
006 Seek error
007 Unknown media type
008 Sector not found
009 Printer out of paper
010 Write fault
011 Read fault
012 General failure
013 Sharing violation (DOS 3+)
014 Lock violation (DOS 3+)
015 Invalid disk change
016 FCB Unavailable (DOS 3+)
017 Sharing buffer overflow (DOS 3+)
018 Code page mismatch (DOS 4+)
019 Out of input (DOS 4+)
020 Insufficient disk space (DOS 4+)
255 Write-protect error (actually DOS critical error 0)

Dos Non-Critical Errors

These error codes can be returned from DOS as a result of a "non-critical" error. Note that the High-Level and User-Level E-Tree routines insulate you from these errors as much as possible. The only time you should encounter them is if you use any of the Library Support routines directly.

Note that we've added 100 to the actual DOS non-critical error codes to make it easy to distinguish them from DOS "critical errors."

101 Function number invalid
102 File not found
103 Path not found
104 Too many open files (no handles available)
105 Access denied
106 Invalid handle
107 Memory control block destroyed
108 Insufficient memory
109 Memory block address invalid
110 Environment invalid (usually >32K in length)
111 Format invalid
112 Access code invalid
113 Data invalid
114 Reserved
115 Invalid drive
116 Attempted to remove current directory
117 Not same device
118 No more files--DOS 3+---
119 Disk write-protected
120 Unknown unit
121 Drive not ready
122 Unknown command
123 Data error (CRC)
124 Bad request structure length
125 Seek error
126 Unknown-media type (non-DOS disk)
127 Sector not found
128 Printer out of paper
129 Write fault
130 Read fault
131 General failure
132 Sharing violation
133 Lock violation
134 disk change invalid

135 FCB unavailable
136 Sharing buffer overflow
137 (DOS 4+) Code page mismatch
138 (DOS 4+) Cannot complete file operation (out of input)
139 (DOS 4+) Insufficient disk space
140-149 Reserved
150 Network request not supported
151 Remote computer not listening
152 Duplicate name on network
153 Network name not found
154 Network busy
155 Network device no longer exists
156 Network BIOS command limit exceeded
157 Network adapter hardware error
158 Incorrect response from network
159 Unexpected network error
160 Incompatible remote adapter
161 Print queue full
162 Queue not full
163 Not enough space to print file
164 Network name was deleted
165 Network: Access denied
166 Network device type incorrect
167 Network name not found
168 Network name limit exceeded
169 Network BIOS session limit exceeded
170 Temporarily paused
171 Network request not accepted
172 Network print/disk redirection paused
173 (LANtastic) Invalid network version
174 (LANtastic) Account expired
175 (LANtastic) Password expired
176 (LANtastic) Login attempt invalid at this time
177 (LANtastic v3+) Disk limit exceeded on network node
178 (LANtastic v3+) Not logged in to network node
179 Reserved
180 File exists
181 Reserved
182 Cannot make directory
183 Fail on INT 24h
184 (DOS 3.3+) Too many redirections
185 (DOS 3.3+) Duplicate redirection
186 (DOS 3.3+) Invalid password
187 (DOS 3.3+) Invalid parameter

188 (DOS 3.3+) Network write fault
189 (DOS 4+) Function not supported on network
190 (DOS 4+) Required system component not installed

Appendix D
Using E-Tree Plus With QuickBASIC / PDS

D (A) Files Distributed with E-Tree Plus

Once you install E-Tree Plus onto your hard drive, the following files and subdirectories will be present:

ASM\	The subdirectory where the assembly source code and .OBJ files can be found.
OBJ1\	When the BASIC source code files are compiled, this is where the .OBJ files will be placed. If you are using PDS, "far string" .OBJ files will be placed here.
OBJ2\	If you are using PDS, "near string" object files will be placed here. If using QuickBASIC 4.x or BASIC 6.x, this subdirectory will not be created.
SUB\	The subdirectory containing the E-Tree BASIC source code modules. In this directory there are various *.SUB modules. Each is stored in ASCII format and has a header describing its contents.
README.DOC	If present, it contains important information such as documentation errata or other things we forgot to put in the manual.
*.BAS	BASIC example programs.
*.MAK	"Make" files used in conjunction with QB and PDS.
ETREE.BI	A file that contains the E-Tree COMMON SHARED block, User-Level and High-Level procedure declarations. It must be $Included at the top of every program module that invokes an E-Tree procedure.
*.BI	Other include files used internally by the E-Tree procedures.

BLDLIB.BAT	Batch file that constructs the LINK libraries that are used with E-Tree Plus.
LIB.DAT	A LIB.EXE response file used when building E-Tree's default .LIB file.
LIBN.DAT	A LIB.EXE response file used when building a .LIB for PDS "near string" programs.
BLDQLB.BAT	Batch file that builds a QuickLibrary for E-Tree Plus.
QLB.DAT	A LINK response file used when building the E-Tree QuickLibrary.
QLBASM.DAT	A LINK response file used when building a QuickLibrary containing only E-Tree's assembly language routines.
BLDRTM.BAT	Batch file that constructs an "extended runtime library" ("RTM") for use with BASIC 6.x and PDS. See Section (G) of this Appendix for more information.
EXPORT.DAT	Export file used in conjunction with BLDRTM.BAT.
COMP.BAT	Batch file used to compile the E-Tree BASIC modules.
EUTIL.EXE	E-Tree's database maintenance utility.
EUTIL.DOC	An ASCII text file containing documentation for EUTIL.EXE.

D (B) Procedure Declarations

Most of the E-Tree procedures receive their arguments "by reference", "by value" or by a "segmented address." We have defined the correct parameter-passing conventions for E-Tree routines using BASIC's "DECLARE" statement. The declarations for the E-Tree procedures can be found in a file called "ETREE.BI." As long as you $Include this file at the top of each program source file that invokes an E-Tree procedure, you will not have to give parameter-passing conventions another thought.

Keep in mind that including these declarations in your BASIC source files will not affect the size of your final .EXE program. However, the declarations <u>will</u> force BASIC to check the number and type of parameters passed to a procedure and will also insure that the correct parameter-passing conventions are used. This benefit alone can save you hours of debugging.

D (C) Using PDS's "Far String" Option

If you installed E-Tree Plus for the Microsoft PDS compiler, two LINK libraries were generated for you when BLDLIB.BAT was executed:

- ETREE.LIB - Contains the "far string" versions of the routines
- ETREEN.LIB - Contains the "near string" versions of the routines

LINK to ETREE.LIB when compiling with PDS's "/FS" switch and LINK to ETREEN.LIB when compiling without "/FS." Although the far string option does make more variable-length string space available, your programs' performance will suffer due to the complex nature of BASIC's far string management code.

If you installed for QuickBASIC 4.x or BASIC 6.x, only one LINK library was created: ETREE.LIB.

D (D) Building QuickLibraries

When you installed E-Tree Plus, a default QuickLibrary was created for you automatically. If for some reason you wish to build a new one, the batch file BLDQLB.BAT can be used for this purpose. From the default E-Tree directory, type "BLDQLB" to generate new ETREE.QLB and .LIB files.

D (E) Building LINK Libraries

After installing E-Tree Plus, you should have executed BLDLIB.BAT. This batch file creates the required .QLB and .LIB files for your compiler. Once executed, a default LINK library was created for you (the PDS version has two: a near string and a far string version). If you modify any of the E-Tree source files (which normally shouldn't be necessary), you will need to rebuild the default E-Tree LINK library(s). We have provided a batch file called BLDLIB.BAT which performs this task. Run this batch file from the default E-Tree directory if you wish to rebuild the default library(s).

D (F) Combining Libraries from other Products

When you installed E-Tree Plus, we automatically created a QuickLibrary called ETREE.QLB which contains all of the various E-Tree routines. At some point, you may wish to create a QuickLibrary containing code from E-Tree as well as code from another add-on product. The process is relatively simple:

- The file BLDQLB.BAT is used to build the default E-Tree QuickLibrary. It essentially contains the following LINK command:

  ```
  LINK /Q @QLB.DAT, ETREE.QLB, NUL, BQLB45 ;
  ```

  Where:

  - QLB.DAT is the response file which contains the paths and names of all of the modules that will be included in the QuickLibrary,
  - ETREE.QLB is the name of the QuickLibrary the LINK program will generate,
  - NUL directs LINK not to generate a map file, and
  - BQLB45 is the name of the QuickLibrary support file. The actual library name that appears in that position may differ depending on your version of QuickBASIC / PDS.

- Load QLB.DAT into an ASCII text editor. Do not use the QB(x) environment because it has a tendency to add unwanted trailing blank lines.

- Add your additional module or library names to the response file, keeping the same format as you see there already. Each line in the file should end with a "+" except for the last line. Be sure there are no extraneous blank lines after the last line containing module names. Once you've made the desired modifications, save the file to disk.

- Run the BLDQLB.BAT file. The resulting new ETREE.QLB will now contain all of the modules listed in the response file.

D (G) Building Extended Runtime Libraries

Extended runtime libraries ("RTMs") are available in Microsoft BASIC 6.x and PDS 7.x. They allow you to combine custom program code (written in BASIC or other languages) with Microsoft runtime code into one "extended" runtime library. This library is automatically loaded high in memory when your "BRUN" BASIC program (compiled without /O) is executed. It also allows programs that CHAIN to shared common program code as well as common data. This means that the E-Tree routines can be placed in an RTM and a group of programs can share all program code in the RTM. Because the routines in the RTM do not have to be specifically LINKed to -each- program, the overall distribution size of your programs much smaller and the programs load faster into memory.

When you installed E-Tree Plus, all of the files required to build an extended runtime library ("RTM") containing the E-Tree routines were placed in the default E-Tree directory. We didn't automatically build an RTM for you for several reasons. However, if you wish to build one now, please follow the guidelines set forth below:

- When you installed BASIC 6.x or PDS, be sure that you specified that the "component libraries" should be retained after installation. If all of the required BASIC support libraries are not present on your hard disk and in a directory indicated by your LIB environment variable, several error messages will be generated by the BUILDRTM.EXE program when it is invoked.

- Make sure that the program BUILDRTM.EXE is in the current directory or in a directory included in your DOS PATH.

- From the default E-Tree directory, type the following:

  ```
  BLDRTM RtmName [switches]
  ```

  Where: RtmName is the name of the output RTM .EXE and .LIB files; switches are either/both of "/Fpa" or "/FS." If any of these switches were used when compiling your main program(s), they must also be used when building the RTM.

Three files are actually generated when building a RTM:

- IMPORT.OBJ, which must be LINKed to the front of each program that will access the RTM,
- RtmName.EXE, which is the actual RTM and must be distributed with your programs, and
- RtmName.LIB which must be LINKed to each program that will access the RTM.

Let's take an example. Say we have two programs, "PROG1.BAS" and "PROG2.BAS." We want both to have access to a RTM that includes the E-Tree Plus routines. There are three steps involved:

- Compile the programs. Remember that you cannot use the "/O" compiler switch (stand-alone) on programs that will access a RTM. Also remember that if you use the "/FPA" (alternate math library) or the "/FS" (PDS "far strings") switches, you must also use them when building the RTM. To compile our example programs, do the following:

  ```
  BC PROG1 /FPA ;
  BC PROG2 /FPA ;
  ```

- Next, generate the RTM. Since we used the "/FPA" switch above, we must also use it when building the RTM. We'll create the RTM "CUSTOM.EXE" with the command:

  ```
  BLDRTM CUSTOM /FPA
  ```

  Assuming no errors occurred, the files CUSTOM.EXE, IMPORT.OBJ and CUSTOM.LIB were generated.

- And finally, LINK our programs. When LINKing programs that will access a RTM, you must use the "/NOD" LINK switch. The "/EX" switch listed below is optional, but will make your .EXE files smaller.

  ```
  LINK /EX /NOD IMPORT + PROG1, PROG1, NUL, CUSTOM;
  LINK /EX /NOD IMPORT + PROG2, PROG2, NUL, CUSTOM;
  ```

  Important note: If you make any changes to the RTM, you must reLINK each of your programs with the new IMPORT.OBJ and

CUSTOM.LIB file that will be generated. Failure to do so will cause the runtime error "Incompatible runtime library" to be generated when you attempt to execute your programs.

D (H) Compiling and LINKing

When "Making an EXE" from within the QB or QBX environment, the resulting .EXE file is usually larger than necessary because unneeded compiler switches are often used by QB. For this reason, we recommend that you compile and LINK your program manually, from the DOS prompt. This gives you absolute control over what goes into your .EXE file (at least all of the control that Microsoft gives you). It only involves two steps:

- Compile your program's source code files. This is accomplished by using the BC.EXE program. For example:

  ```
  BC PROGRAM [switches] ;
  ```

  "Switches" might include "/O" to make your program a stand-alone, "/FS" to take advantage of PDS's "far string" option, etc. Consult your compiler documentation for a complete list of available options. For example:

  ```
  BC PROGRAM /O ;
  ```

  This would compile PROGRAM.BAS using the "stand-alone" option, generating PROGRAM.OBJ.

- The next step is to LINK the PROGRAM.OBJ file with all of the other code required to make it an executable program. Such "support code" is usually found in libraries, such as BCOMxx.LIB (QuickBASIC), BCL71ENR.LIB (PDS), etc. For example:

  ```
  LINK [switches] PROGRAM, , NUL, [libraries] ;
  ```

  "Switches" usually include "/EX" which compresses your .EXE file by five to forty percent. PROGRAM is the name of the .OBJ file generated by the compiler in the first step. The two commas that follow the program name direct LINK to give the resulting .EXE file the same primary name as the first .OBJ listed. In this case, the .EXE file will be called PROGRAM.EXE. The "NUL" directs the LINK program not to generate a "map" file (they're of little use to BASIC programmers). Finally, you can list one or more entries in the "libraries" field. By default, the LINK program automatically searches the compiler support libraries, so there's no need to list them. If you also wish LINK to look in other libraries as well, such as E-Tree's library, you can list it here. For example:

```
LINK /EX PROGRAM, , NUL, ETREE ;
```

This would LINK PROGRAM.OBJ into an .EXE file. All routines required by PROGRAM that are present in the BASIC support library and the E-Tree library would be brought into the .EXE file automatically. Again, only code required by PROGRAM will be extracted from the library(s) and made part of the final .EXE file. Assuming there were no LINKer errors, PROGRAM.EXE would be generated and ready to execute at the completion of this step.

D (I) Using E-Tree in Programs that CHAIN and SHELL

If you are using QuickBASIC 4.x and will be writing programs that CHAIN (those compiled without the "/O" switch), be absolutely sure to close all E-Tree files prior to the CHAIN statement. The reason for this is simple. QB4 programs require that "external" routines be LINKed directly to the main program modules. When a program CHAINs to another, all of the code and data that was in use by the CHAINing program is dumped in order to make room for the CHAINed-to program, including E-Tree's internal variables and pointers. This is not a good thing! However, the E-Tree Plus code is smart enough to detect this situation and automatically perform the necessary clean-up for you. However, you can't CHAIN into the second program and expect E-Tree files to still be open. If this is a problem for you, consider upgrading to Microsoft's BASIC 6.x or PDS 7.x compiler for which we have a solution.

If you are using the Microsoft BASIC Compiler 6.x or PDS, you can put the E-Tree routines in an extended runtime library ("RTM"). The RTM will remain in memory, even across CHAINs. However (there's always a "gotcha"), be sure to close all E-Tree files if you decide to SHELL. This is because BASIC unloads the RTM from memory just before a SHELL in order to make more memory available for the operating system. For more information on building and using RTMs, see Section (G), "Building Extended Runtime Libraries" in this appendix.

D (J) Running the Example Programs

All E-Tree example programs are stored in the default E-Tree directory on your hard disk and have an extension of .BAS. To load and run one into the QB 4.5 environment, make the E-Tree directory your current directory and type the following at the DOS prompt:

```
QB Example /L ETREE
```

To load an example program into the QBX environment (PDS users), type the following at the DOS prompt:

```
QBX Example /L ETREE /ES
```

Important note: The "/ES" switch used above directs QBX to preserve the EMS memory state prior to invoking any external routines - a necessity when using E-Tree in QBX. Failure to do so on a system equipped with EMS will most certainly result in a system lock-up shortly after the first E-Tree routine is invoked.

Once in the environment, you can examine the source code, step-trace through it in order to see what's going on, or just run the program at full speed. If you wish to compile an example program, type the following at the DOS prompt:

QB4.x and BASIC 6.x users:

```
BC Program /O ;
LINK /EX Program, , NUL, ETREE ;
```

PDS users:

```
BC Program /O;
LINK /EX Program, , Nul, ETREEN ;
```

D (K) Example Program Listing

The following source code file can be found on disk in EXAMPLE.BAS. It is provided here to provide you with an easy reference to a working example of how to create and traverse an E-Tree Database file. Other example programs can be found in your default E-Tree subdirectory.

```
'This is a sample source code file that demonstrates how the E-Tree Plus
'"user-level" routines are used in QuickBASIC and PDS programs.

'If running this program in the QBX environment, be sure to use the /ES
'switch when invoking QBX if you have EMS memory.

DEFINT A-Z
```

```
'These procedures are used only by this example program. They are not
'actually part of the E-Tree package.

DECLARE SUB PrintHeader (A$)
DECLARE SUB PrintRecord (C AS ANY)
DECLARE SUB PressAKey ()            'We use this function only in this pgm

'The following $Include file contains declarations for all of E-Tree's
'"user-level" procedures. These are the procedures that you will most
'likely be using in your programs. If you require lower-level access to
'E-Tree's database structure, over 40 additional low-level routines
'are also available.

REM $INCLUDE: 'ETREE.BI'            'Contains all user-level procedure
                                    'declarations.


TYPE MyRecType                      'This is a sample record definition
    CustNumber  AS LONG
    FirstName   AS STRING * 20
    LastName    AS STRING * 20
    Addr        AS STRING * 30
    City        AS STRING * 20
    State       AS STRING * 2
    ZipCode     AS STRING * 10
    AreaCode    AS STRING * 3
    PhoneNum    AS STRING * 8
    AmountDue   AS DOUBLE
    Age         AS INTEGER
END TYPE

DIM C AS MyRecType                  'Define a variable with our structure

CLS

FileName$ = "TEST.ETR"

IF EtFileExist(FileName$) THEN              'If file exists, open it
    EtOpen FileName$, Handle%, Status%
    IF Status% THEN                         'If it's not a valid ETree file
        PRINT "Error"; Status%; " opening "; FileName$
        END
    END IF
ELSE                            'If the file doesn't exist, create it
    Pagesize& = 4096            'Automatically calculate optimum page size
    PreAllocation& = 0          'Don't preallocate any extra disk space
    READ NumberOfFields%        'Our record info is in data statements below

    'Read the field descriptions from DATA statements into the RecordInfo()
    'array.

    REDIM RecordInfo(1 TO NumberOfFields%) AS EtRecordInfoType
    FOR Ele% = 1 TO NumberOfFields%
        READ RecordInfo(Ele%).FldName, RecordInfo(Ele%).FldType
        READ RecordInfo(Ele%).FldLength
    NEXT

    Maxkeys% = 5                'Max number of keys in the database at once

    'Create the database file. The RecordInfo() array contains the
    'field descriptions (read from DATA statements).

    EtCreate FileName$, Pagesize&, PreAllocation&, RecordInfo(), Maxkeys%, Handle%, _
            Status%

    IF Status% THEN             'In case of a DOS error
        PRINT "Error"; Status%; " creating the database "; FileName$
        END
    END IF

    ERASE RecordInfo                    'We don't need this array anymore

    'Read the descriptions of the indexes from data statements and create
    'each index one at a time.

    READ Indexes%                       'Total number of indexes
    FOR IndexNum% = 1 TO Indexes%
        READ IndexName$, Unique%, NumberOfColumns%
        REDIM Columns$(1 TO NumberOfColumns%)
        FOR ColumnNum% = 1 TO NumberOfColumns%
            READ Columns$(ColumnNum%)
        NEXT

        'Columns$() array contains a list of field names (one per element)
        'that will make up this index. A "combined index" is an index which
        'is comprised of more than one field (as defined by PDS ISAM).

        EtCreateIndex Handle%, IndexName$, Unique%, Columns$(), Status%
        IF Status% THEN
            PRINT "Error"; Status%; " creating index "; IndexName$; " in "; FileName$
            EtClose Handle%
            END
        END IF
    NEXT                                    'Repeat for each index

    ERASE Columns$                          'We're finished with this
END IF

'-------------------------------------------------------------------------
' We've created (or opened) the database and established our indexes, now
' let's insert some records.
'-------------------------------------------------------------------------

RESTORE Records
READ NumberOfRecs%
LockFlag% = 1                               'Lock before, unlock after
DO
    ThisRec% = ThisRec% + 1

    'Read data into our TYPEd variable "C"

    READ C.FirstName, C.LastName, C.Addr, C.City, C.State, C.ZipCode
    READ C.AreaCode, C.PhoneNum, C.AmountDue, C.Age

    'Insert the new record into the database.

    C.CustNumber = 0            '   AutoIncrement takes care of this one
    EtInsert Handle%, VARPTR(C), LockFlag%, Status%

    IF Status% THEN
        PRINT "Error"; Status%; " inserting record"; ThisRec%
        EtClose Handle%
        KILL "test.etr"
        END
    END IF

LOOP UNTIL ThisRec% = NumberOfRecs%

'--------------------------------------------------------------------------
' Now let's retrieve the data by each of our indexes.
'--------------------------------------------------------------------------

PrintHeader "By 'Null' Index"               'Display info on the screen
EtMoveFirst Handle%, Status%                'Point at first record

DO UNTIL EtEOF(Handle%)                     'Loop until we reach EOF

    EtRetrieve Handle%, VARPTR(C), LockType%, Status%   'Retrieve it
    IF Status% THEN
        PRINT "Error"; Status%; "retrieving on null index!"
        EXIT DO
    END IF
    PrintRecord C                                       'Display it
    EtMoveNext Handle%, Status%             'Point to the next record

LOOP

PressAKey

'-------------------------------------------------------------------------
' Now, by the "Amount Due" index
'-------------------------------------------------------------------------

PrintHeader "By 'AmountDue' Index"          'Display info
EtSetIndex Handle%, "AmountDue", Status     'Set new index and first record
IF Status% THEN
    PRINT Status%; "error on SetIndex AmountDue"
    EtClose Handle%
    END
END IF

'Here's the same loop as above. Retrieve a record, display it, and move
'to the next one. Loop until the EOF (end of file) is reached.

DO UNTIL EtEOF(Handle%)
    EtRetrieve Handle%, VARPTR(C), LockType%, Status%
    IF Status% THEN
        PRINT "Error"; Status%; "retrieving on Amount Due index!"
        EXIT DO
    END IF
    PrintRecord C
    EtMoveNext Handle%, Status%
LOOP
```

```
    PressAKey

    '-----------------------------------------------------------------------
    ' Now, by the "FullName" index
    '-----------------------------------------------------------------------

    PrintHeader "By 'FullName' Index"
    EtSetIndex Handle%, "FullName", Status
    IF Status% THEN STOP

    DO UNTIL EtEOF(Handle%)
        EtRetrieve Handle%, VARPTR(C), LockType%, Status%
        IF Status% THEN
            PRINT "Error"; Status%; "retrieving on FullName index!"
            EXIT DO
        END IF
        PrintRecord C
        EtMoveNext Handle%, Status%
    LOOP

    PressAKey

    '-----------------------------------------------------------------------
    ' Now, by the "CustNumber" index
    '-----------------------------------------------------------------------

    PrintHeader "By 'CustNumber' Index"
    EtSetIndex Handle%, "CustNumber", Status
    IF Status% THEN STOP

    DO UNTIL EtEOF(Handle%)
        EtRetrieve Handle%, VARPTR(C), LockType%, Status%
        IF Status% THEN
            PRINT "Error"; Status%; "retrieving on CustNumber index!"
            EXIT DO
        END IF
        PrintRecord C
        EtMoveNext Handle%, Status%
    LOOP

    EtClose Handle%                              'That's all folks!
    END

'---------------------------------------------------------------------------
' Other "user-level" routines available in E-Tree Plus:
'---------------------------------------------------------------------------

' In all of the functions listed below, Handle% is the file handle
' returned by the EtOpen or EtCreate routine when the database was
' opened or created. Routines that have Status% in the parameter list
' use it to return an error code (a non-zero value) instead of generating a
' BASIC runtime error. A Status% code of zero indicates a successful
' operation.

'EtDelete Handle%, Status%  - Deletes the current record in the database
'   referenced by Handle%.

'EtDeleteIndex - Deletes the specified index.

'EtUpdateVariable - Inserts or updates the variable-length portion of
'   a record.

'EtMoveFirst, EtMoveLast, EtMoveNext, EtMovePrevious - Positions the
'  current record pointer.

'EtSeekEQ, EtSeekGE, EtSeekGT, EtSeekLE, EtSeekLT - Searches for a key
'  value in the current index.

'EtUpdate - Updates the fixed-length portion of the current record.

' In addition to these "user-level" routines, there are also over 70
'  other lower-level routines that can be used to manipulate a database
'  in just about any desired fashion. Full source code is provided
'  along with complete documentation for the database structure.

'---------------------------------------------------------------------------
' Definition for the file structure
'---------------------------------------------------------------------------

DATA 11:                                     'Number of fields

'FieldName, Type, Length

DATA CustNumber, 3, 0:                       'Auto-incrementing key (long int)
DATA FirstName, 15, 20:                      '20 byte string field
DATA LastName, 15, 20:                       '20 byte string field
DATA Addr, 15, 30:                           '30 byte string field
DATA City, 15, 20:                           '20 byte string field
DATA State, 15, 2:                                   '2 byte string field
DATA ZipCode, 14, 10:                                '10 byte string (case sensitive)
DATA AreaCode, 14, 3:                                '3 byte string (case sensitive)
DATA PhoneNum, 14, 8:                                '8 byte string (case sensitive)
DATA AmountDue, 7, 0:                                'Double precision
DATA Age, 4, 0:                                      'Integer

'--------------------------------------------------------------------------------
' Definitions for the indexes
'--------------------------------------------------------------------------------

DATA 4:                                              'Number of indexes

'Index name, Unique%, NumberOfColumns%, ColumnName [, ColumnName...]

DATA CustNumber, 1, 1, CustNumber
DATA FullName, 0, 2, LastName, FirstName
DATA ZipCode, 0, 1, ZipCode
DATA AmountDue, 0, 1, AmountDue


'--------------------------------------------------------------------------------
' Some dummy records we can store in the database
'--------------------------------------------------------------------------------

Records:
DATA 5
DATA Tony, Elliott, 4374 Shallowford Industrial Pkwy, Marietta, GA, 30066, 404, _
       928-8960, 10000, 28
DATA Butch, Howard, 4374 Shallowford Industrial Pkwy, Marietta, GA, 30066, 404, _
       928-8960, 1234.56, 32
DATA Bill, Gates, One Microsoft Way, Redmond, WA, 98052-6399, 206, 882-8080, 40, 0
DATA John, Smith, 1234 Main Street, Anytown, US, 12345-6789, 800, 555-1212, _
       -1520, 234
DATA Jean-Luc, Picard, c/o Star Fleet Command, San Francisco, CA, 98250, 203, _
       555-1212, 0, -425

SUB PressAKey STATIC

    PRINT
    PRINT "Press any key to continue ... "
    A$ = INPUT$(1)

END SUB

SUB PrintHeader (A$) STATIC

    CLS
    PRINT A$
    PRINT STRING$(79, "-")
    PRINT "   #  First          Last             Zip   Amount Due"
    PRINT STRING$(79, "-")

END SUB

SUB PrintRecord (C AS MyRecType) STATIC

    PRINT USING "#####"; C.CustNumber;
    PRINT TAB(7); LEFT$(RTRIM$(C.FirstName), 15);
    PRINT TAB(22); LEFT$(RTRIM$(C.LastName), 15);
    PRINT TAB(37); USING "\   \"; C.ZipCode;
    PRINT TAB(43); USING "##,###.##"; C.AmountDue

END SUB
```

Appendix E
Converting PDS ISAM Code to E-Tree

If you are experienced with PDS ISAM or have a program that you wish to convert to E-Tree Plus, the following guidelines can be used to ease your transition.

- ISAM files can contain more than one table per file. E-Tree databases can contain only one table per file. The program EUTIL.EXE included with E-Tree Plus can be used to convert existing ISAM files to E-Tree format (see Section 5). This conversion includes moving each table in an ISAM file into a separate E-Tree file, as well as creating matching indexes.

- E-Tree Plus cannot directly support embedded TYPEs (a TYPE structure within a TYPE structure) or TYPEs that contain arrays (available in PDS). This does not mean that you cannot use them; it simply means that you must use an alternate method of describing them during database creation. All that is required is to replace these as "unsigned binary" fields of the same length as the original data type. For example:

```
TYPE Embedded
      FirstName AS STRING * 20
      LastName AS STRING * 20
END TYPE

TYPE MyRec
      Name        AS Embedded     'An embedded TYPE
      Array(100)  AS INTEGER      'An array within a TYPE
END TYPE
```

  When defining this record structure with the EtCreate routine, simply define these fields as EtUnSigned ("EtUnSigned" is the name of the CONSTant to define the numeric value which actually represents the field type internally) fields of equivalent lengths. In the above example, the "Name" field in MyRec would translate as an EtUnSigned field 40 bytes long (20 + 20), and the "Array" field would translate as an EtUnSigned field 202 bytes long (Dimensions * Elements * BytesPerElement, or 1 * 101 * 2). When determining the length of an array, don't forget that the lower bound starts at element 0 unless you specifically tell the compiler otherwise. Also remember that each array element requires the following amount of memory:

- Integer arrays - 2 bytes,
- Long integer arrays - 4 bytes,
- Double-precision and Currency arrays - 8 bytes, and
- TYPEd arrays - the combined length of all its fields.

- Remember, you can still use the same TYPEd variable as you used in your PDS ISAM database; embedded TYPEs and arrays must simply be described as EtUnSigned fields. Also keep in mind that if you are using EUTIL to convert your existing ISAM databases, these concerns are handled for you automatically.

- Transaction processing is not currently supported by E-Tree Plus. This means that we have no alternatives for the following ISAM keywords: BEGINTRANS, SAVEPOINT, ROLLBACK, ROLLBACK ALL, and COMMITTRANS. In most cases we have encountered, transaction processing is not crucial for database updates. However, we do intend to include support for it in a future release of E-Tree Plus.

A few of the E-Tree equivalent statements are slightly different than those you've used in PDS ISAM. Generally speaking, a different syntax was required when:

- The BASIC statement allowed a variable number of parameters,

- The BASIC statement directly interpreted a TYPE structure (such as in BASIC's OPEN statement), or

- The E-Tree version requires a record-locking decision to be made.

Below, we'll discuss each of the PDS ISAM statements that we support along with their E-Tree equivalents. Instead of listing them in alphabetical order, we'll list them in the most likely order of usage.

OPEN

ISAM:

```
OPEN FileName$ FOR ISAM TypeName "TableName" AS #BufferNumber
```

E-Tree:

```
'When creating a new database file:
EtCreate FileName$, PageSize&, PreAllocation&, RecordInfo(), _
      MaxKeys%, Handle%, Status%
```

```
'When opening an existing database file:
EtOpen FileName$, Handle%, Status%
```

There are a couple of fundamental problems in exactly duplicating the BASIC OPEN syntax. In PDS ISAM, a table's structure is defined by a TYPE...END TYPE statement. This is used to define the field names, lengths, and types of data that will comprise an ISAM table. Because TYPE structures are interpreted at compile-time and translated into addresses and offsets, E-Tree Plus cannot use them directly to actually define the record structure of a database file. For this reason we chose to provide two routines to be used to open a database: EtCreate is used to create a new database file and has the additional parameters required by this operation; EtOpen is used to open an existing database file and has a much shorter and less complicated parameter list.

Below, we'll show you two code fragments. The first demonstrates how to open and create a database file in PDS ISAM, and the second in E-Tree Plus:

```
'Creating and opening a database in PDS ISAM:
'
TYPE MyType
  FirstName AS STRING * 20
  LastName AS STRING * 20
END TYPE

DIM MyVar AS MyType

FileName$ = "database.mdb"
OPEN FileName$ FOR ISAM MyType "MyTable" AS #BufferNumber



'Creating and opening a database in E-Tree:
'
REM $INCLUDE: 'ETREE.BI'

TYPE MyType
  FirstName AS STRING * 20
  LastName AS STRING * 20
END TYPE

DIM MyVar AS MyType

FileName$ = "database.etr"

IF EtFileExist%(FileName$) THEN                       'If file exists
  EtOpen FileName$, Handle%, Status%                  ' open it normally.
ELSE                                                  'Otherwise create it
  DIM RecordInfo(1 TO 2) AS RecordInfoType            'Needed by EtCreate

  RecordInfo(1).Name = "FirstName"                    'Field Name
  RecordInfo(1).Type = EtString                       'Field Type
  RecordInfo(1).Length = LEN(MyVar.FirstName)         'Field Length

  RecordInfo(2).Name = "LastName"                     'Field Name
  RecordInfo(2).Type = EtString                       'Field Type
  RecordInfo(2).Length = LEN(MyVar.LastName)          'Field Length

  MaxKeys% = 2                                        'Max number of indexes

  EtCreate FileName$, PageSize&, PreAllocation& RecordInfo(), _
      MaxKeys%, Handle%, Status%
END IF
```

In the EtCreate example above, there are a few parameters which may seem foreign to you:

- The RecordInfo() array is used to define the name, type and length of each field. This is necessary since it is not possible for us to interpret a TYPE...END TYPE structure directly.

- PageSize& allows you to manually define the size of each "page" used to store data and internal information about the database file. If you pass this as zero, we'll calculate an optimum page size for you.

- PreAllocation& represents the amount of disk space in bytes that you wish to preallocate for the file. PDS ISAM normally preallocates 64K bytes of disk space for a new file. At first, we suggest that you pass values of zero for both PageSize& and PreAllocation& and let the routines do most of the work for you.

- MaxKeys% represents the maximum number of indexes you wish to have maintained in the database at one time. Remember, you can add and delete indexes at any time, however, you cannot have more than MaxKey% indexes in the file at one time.

There are a couple of other differences between BASIC's OPEN and E-Tree's. that you should keep in mind. When opening a file in BASIC, you assign a "buffer number" or "handle" to the file. In E-Tree Plus, you have a choice of passing a value in Handle% (between 1 and MaxOpen% [4]), or passing a value of zero and allowing the EtOpen and EtCreate procedures to return a handle to you. The Handle% is then used with every other E-Tree routine when addressing that specific file. Also, BASIC can generate a runtime error if you attempt to open a file using an invalid filename or point to a directory that doesn't exist. Instead of generating a runtime error, E-Tree routines return a completion code in the Status% parameter. If Status% is returned as zero, the function was successful, otherwise the value returned represents an error code. See Appendix B for a complete list of E-Tree Plus error codes.

Close

PDS ISAM:

```
CLOSE #BufferNumber
```

E-Tree Plus:

```
EtClose Handle%
```

As you can see, there is no functional difference between the two implementations.

CreateIndex

PDS ISAM:

```
CREATEINDEX #BufferNumber, IndexName$, Unique%, "ColumnName" _
       [, ColumnName [. . .]]
```

E-Tree:

```
'For index types supported by PDS ISAM:
EtCreateIndex Handle%, IndexName$, Unique%, Columns$(), Status%

'For keys based on partial fields, descending, and non-modifyable indexes:
EtCreateIndex2 Handle%, IndexName$, Desc(), Unique%, Status%
```

Besides supporting the all of the key types offered by PDS, E-Tree Plus also supports true segemented (one key comprised of pieces of one or more fields), descending, and non-modifyable keys. For this reason we chose to provide two CreateIndex functions: EtCreateIndex has a very similar syntax to PDS ISAM and provides the same level of functionality; EtCreateIndex2 has extensions to support the added functionality available in E-Tree Plus.

Because add-on procedures written for BASIC cannot accept a variable number of parameters, the EtCreateIndex routine uses an array to supply the column names that will comprise the index, one column name per element. For example:

PDS ISAM:

```
CREATEINDEX #BufferNumber, "FullName", 0, "LastName", _
                 "FirstName"
```

E-Tree:

```
REDIM Columns$(1 TO 2)
Columns$(1) = "LastName"
Columns$(2) = "FirstName"
EtCreateIndex Handle%, "FullName", 0, Columns$(), Status%
```

As you can see, the E-Tree implementation requires a bit more code but isn't much more complex. Our goal was to maintain similar syntax and functionality.

DeleteIndex

PDS ISAM:

```
DELETEINDEX #BufferNumber, IndexName$
```

E-Tree Plus:

```
EtDeleteIndex Handle%, IndexName$, Status%
```

EtDeleteIndex adds only a Status% parameter. If the operation is unsuccessful (index doesn't exist or lock error), Status% will return an error code instead of triggering a runtime-error and terminating your program.

BOF, EOF, LOF

PDS ISAM:

```
BeginningOfFileFlag% = BOF(BufferNumber%)
EndOfFileFlag% = EOF(BufferNumber%)
RecordsInTable& = LOF(BufferNumber%)
```

E-Tree Plus:

```
BeginningOfFileFlag% = EtBOF(Handle%)
EndOfFileFlag% = EtEOF(Handle%)
RecordsInTable& = EtLOF(Handle%)
```

SetIndex

PDS ISAM:

```
SetIndex #BufferNumber, IndexName$
```

E-Tree Plus:

```
EtSetIndex Handle%, IndexName$, Status%
```

EtSetIndex adds only a Status% parameter. If the operation is unsuccessful (index doesn't exist), Status% will return an error code instead of triggering a runtime-error and terminating your program.

SeekEQ, SeekGT, SeekGE

PDS ISAM:

```
SEEKxx #BufferNumber, KeyValue [, KeyValue ... ]
```

E-Tree Plus:

```
EtSeekxx Handle%, KeyValueAddress&, Status%
```

There are a couple of differences worth noting here. As with many other E-Tree routines, a Status% parameter has been added. In case a DOS file I/O error occurs (the most likely time an error can occur with this routine), the DOS error code is returned in this parameter.

The PDS ISAM version of SEEKxx accepts a variable number of key values as well as a variable number of key types. This adds a great deal of complexity to our attempt at maintaining a similar syntax in E-Tree's EtSeekxx routines. Since our user-level routines are written in BASIC, we must follow BASIC's rules that apply to parameter-passing. So, instead of literally passing one or more KeyValues as you do with PDS ISAM, you must pass EtSeekxx the memory address where the key value is stored. BASIC has built-in functions that return this information to you. They are:

- **VARPTR** is used to return the offset in DGroup (BASIC's "near" data segment) where simple "scalar" variables, fixed-length strings, user-defined "TYPEd" variables, and elements of static arrays are stored. For example:

```
Address& = VARPTR(A%)      'Offset of A% (scalar variable)

DIM B AS STRING * 20
Offset% = VARPTR(B)        'Offset of fixed-length string

REM $STATIC                'Create a static array which
DIM Array%(100)            ' is stored in DGroup.
Address& = VARPTR(Array%(10))   'Offset of element 10
```

- **SADD** is used in QuickBASIC 4.x and BASIC 6.x to return the offset of a "near" variable-length string or element of a variable-length string array. For example:

```
Address& = SADD(A$)          'Or
```

```
Address& = SADD(Array$(25)) 'Address of element 25
```

- **SSEGADD** is used in PDS 7.x to return the far address of a variable-length string or string array. Because PDS supports both "near" and "far" strings, SSEGADD is the only reliable way to determine a string's address in either memory-model. For example:

```
Address& = SSEGADD(A$)          'Or
Address& = SSEGADD(Array$(8))   'Address of element 8
```

- **VARSEG** and **VARPTR** are used together to return the segment and offset (respectively) of an element of $Dynamic numeric or fixed-length string arrays. Using the **EtFarAddress** routine, these values are combined into a far address to the specified array element. Although it is unlikely that you'll ever use an array element as a key value, we just wanted to show you how just in case the need should arise:

```
REM $DYNAMIC
DIM Array%(100)             "REDIM" also makes it $Dynamic
Address& = EtFarAddress&(VARSEG(Array%(5)), _
                         VARPTR(Array%(5)))
```

As we mentioned previously, you must pass the address of the KeyValue to the EtSeekxx routines instead of the KeyValue itself. The KeyValue you are seeking must be as long as the key actually stored in the file. It can be longer, but not shorter. Let's say we have a file indexed on a "LastName" field which is a 20 byte-long string . If we want to find the first record in the database that is greater-than or equal to "Smith", we need to build a buffer that is at least 20 bytes long. You can accomplish this any of several ways:

```
'Create a variable-length string padded with spaces

KeyValue$ = SPACE$(20)
LSET KeyValue$ = "Smith"
EtSeekGE Handle%, SADD(KeyValue$), Status%


'Use a fixed-length string that is as long or longer as needed.
'All we actually need is 20 bytes, but a longer string could
'potentially be used with more than one index.

DIM KeyValue AS STRING * 100
KeyValue = "Smith"        'BASIC automatically pads with spaces
EtSeekGE Handle%, VARPTR(KeyValue), Status%
```

If you have a "combined index" consisting of more than one column, you must provide the EtSeekxx routines with a KeyValue that looks like a combination of the various columns comprising the index. For example, say you have an index called "FullName" that consists of the "LastName" and "FirstName" columns (each are 20 byte strings). You can set up a TYPEd variable that represents this combined KeyValue and load each field with the values you are seeking:

```
TYPE FullNameStuc
  LastName AS STRING * 20
  FirstName AS STRING * 20
END TYPE

DIM FullName AS FullNameStruc

'Open the file and make the "FullName" index active

'Let's search for John Smith:
FullName.LastName = "Smith"
FullName.FirstName = "John"
EtSeekEq Handle%, VARPTR(FullName), Status%
```

You could also build a variable-length string that has the same structure as the index. Granted this is much more difficult than using a TYPEd variable as described above, but it does allow a more dynamic approach (variable-length strings can be constructed at runtime and TYPEd variables must be defined at compile-time):

```
FullName$ = LEFT$("Smith"+SPACE$(20),20) + _
                 LEFT$("John"+SPACE$(20),20)
EtSeekEq Handle%, SADD(FullName$), Status%
```

MoveFirst, MoveLast, MoveNext, MovePrevious

PDS ISAM:

```
MOVExx #BufferNumber
```

E-Tree Plus:

```
EtMovexx Handle%, Status%
```

The syntax remains the same with the exception of the Status% parameter. If a DOS error occurs during an EtMovexx function, the error code will be returned in Status% (PDS ISAM would trigger a runtime error).

Retrieve, Update, Insert

All three of these statements use the same syntax. We'll use RETRIEVE and EtRetrieve in the examples listed below.

PDS ISAM:

```
'All three have the same syntax. We'll use RETRIEVE
```

```
RETRIEVE #BufferNumber, TypedVariable
```

E-Tree Plus:

```
EtRetrieve Handle%, VARPTR(TypedVariable), LockType%, Status%
```

In E-Tree's implementation, you pass the address of the TYPEd variable instead of the TYPEd variable itself (see the entry for SEEK for an explanation). LockType% indicates the type of record lock you wish to place on the record (see Section 4(D) for more information about record-locking options). If an error occurs (conflict with a unique key or a DOS error), the appropriate error code is returned in the Status% parameter (PDS ISAM would generate a runtime error).

Delete

PDS ISAM:

```
DELETE #BufferNumber
```

E-Tree Plus:

```
EtDelete Handle%, Status%
```

Status% will be returned with a non-zero value if a DOS file I/O error occurs or if the record you wish to delete is locked by another process.

GetIndex$

PDS ISAM:

```
IndexName$ = GETINDEX$(BufferNumber)
```

E-Tree Plus:

```
IndexName$ = EtGetIndex$(Handle%)
```

There is no difference in syntax or functionality.

TextComp

PDS ISAM:

```
Result% = TEXTCOMP(A$, B$)
```

E-Tree Plus:

```
Result% = EtTextComp(A$, B$)
```

There is no difference in syntax or functionality.

Index