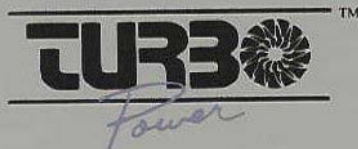
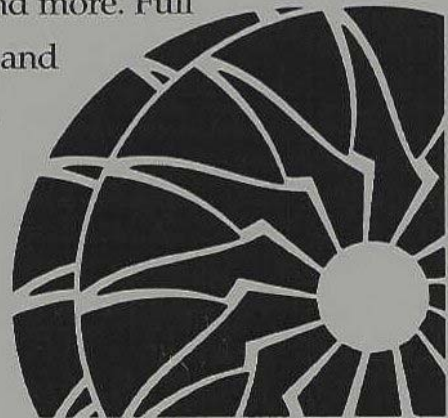


**TURBO
PASCAL**

*B-Tree Filer*TM

Write fast, compact multi-user database applications. You get ■ B+tree indexing
■ Fixed and variable length records
■ Visual database browsers ■ Network functions for messaging and printing
■ DOS, DPML, and Windows support
■ Compatibility with all PC-based networks, and more. Full source code and no royalties.



B-Tree Filer

User's Manual

Copyright (c) 1989-1995, TurboPower Software Company

All Rights Reserved

Fifth Edition April 1995

Order line (US and Canada): 800-333-4160

Elsewhere: 719-260-9136

Technical Support

TurboPower Software Company

P.O. Box 49009

Colorado Springs, CO 80949-9009

Telephone Support: 719-260-6641

9am to 5pm Mountain Time, Mon-Fri

Fax: 719-260-7151

BBS: 719-260-9726

CompuServe: 76004,2611

PCVENB Section 6

B-Tree Filer is copyright (c) 1989-1995 by TurboPower Software Company, all rights reserved.
All TurboPower product names are trademarks or registered trademarks of TurboPower Software Company. Other brand and product names are trademarks or registered trademarks of their respective holders.

Table of Contents

1. Introduction	1
A. Requirements	2
B. Installation	3
C. Database Demonstration Programs	11
1. DOS Text Mode Browser: NETDEMO	11
2. Object Professional Browser: OPISDEMO	15
3. Turbo Vision Browser: TVISDEMO	17
4. ObjectWindows Library Browser: OWDEMO	20
D. The Help System	22
E. Purchase Agreement	30
2. Configuration.....	33
A. Conditional Compilation Defines	34
B. Compiler Options	40
C. Customizing the Network Interface	42
3. B-tree Principles	45
A. Data and Keys	46
B. B-tree Organization	47
C. Managing Keys	51
4. Using B-Tree Filer	55
A. Fileblocks	55
B. Program Organization	56
C. Single User Example	57
D. Locking Techniques	66
E. Converting Single User Programs	71
F. Network Example	74
G. Questions and Answers	79
5. B-Tree Filer Identifiers	81
6. Tools	155
A. Using EMS for Heap Storage: EMSHEAP	157
B. Variable Length Records: VREC	173
C. Restructuring and Reindexing: RESTRUCT/REINDEX	186
D. Rebuilding a Fileblock: REBUILD/VREBUILD	195
E. Reorganizing a Fileblock: REORG/VREORG	197
F. Converting to Variable Length Records: FIXTOVAR	200
G. Numeric Keys: NUMKEYS	203
H. Miscellaneous Routines: ISAMTOOL	212
I. Sorting: MSORT/MSORTP	216
1. Sorting in Real Mode: MSORT	217
2. Sorting in Protected Mode: MSORTP	227
J. dBase File Conversion: DBIMPEXP	236
K. Demonstration Programs	257
1. Convert dBase file to Filer: DB2ISAM	257
2. Convert Filer file to dBase: ISAM2DB	259

7. Browsers	261
A. Displaying a Fileblock: BROWSER.....	263
B. OOP Abstract Browser	285
C. Object Professional Compatibility Layer: OPBROW.....	288
D. Turbo Vision Compatibility Layer: TVBROWS	305
E. ObjectWindows Library Compatibility Layer: WBROWSER	319
8. Network Utilities	343
A. Novell NetWare Support	344
1. Basic NetWare Access: NWBASE.....	345
2. Connection Services: NWCONN	349
3. NetWare Bindery Services: NWBIND	357
4. Basic Messaging Services: NWMSG	371
5. NetWare File Support: NWFILE.....	375
6. NetWare Semaphore Access: NWSEMA	382
7. Transaction Tracking Services: NWTTS.....	386
8. Print Capture and Print Queue Services: NWPRINT	391
9. Advanced Message Services: NWIPXSPX	406
B. NetBIOS Message Passing: NETBIOS	435
C. DOS Network Functions: SHARE	462
D. Network Demonstration Programs	475
1. Network Information: NETINFO	476
2. File Transfer Using NetBIOS: NBSEND	477
3. File Transfer Using IPX: NISEND	478
4. File Transfer Using SPX: NSSEND	479
5. Two-Way Chat Program Using SPX: SPX2WAY	480
6. Bindery Listing: BINDLIST	481
7. Using TTS in a B-Tree Filer Application: TTSFILER	482
9. Appendix.....	483
A. Error Codes.....	483
B. Converting from Borland's Database Toolbox.....	492
C. Converting from earlier versions of B-Tree Filer	499
Identifiers.....	503
Index.....	511

1. Introduction

The storage and management of data is one of the central problems of electronic data handling. It wouldn't be an exaggeration to say that, for many programmers, database work is what pays the bills. Although there are lots of database management tools available, B-Tree Filer is special in that it gives you small code size, complete control over record format and index structure, and the best performance available. It also provides tools for the networking functions that are often needed in multi-user applications--messaging, printer control, semaphores, and more.

B-Tree Filer offers you a high-quality library to manage databases in an efficient and easy-to-use manner. It includes many features that set it apart from the competition:

- integral support for multi-user, networked databases
- strict error checking and safety modes to maximize data integrity
- written in Turbo Pascal, with full source code provided
- fixed and variable length record support
- all indexes stored in one file to minimize file handle usage
- separate data and index files to maximize data integrity
- support for EMS and normal heap memory for index page buffers
- no TSRs required at runtime
- utility units for browsing, rebuilding, and sorting data files
- support for Novell, MS-NET, and all NetBIOS compatible networks
- transparent support for real mode and protected mode applications under MS-DOS and Windows
- network access units for file management, printer control, and message passing

B-Tree Filer is based on an improved version of the balanced B-tree algorithm, which has proven itself as the best method to access data in large databases. B-Tree Filer's capacity is so large that you should never need to worry about it:

- Maximum number of data records: greater than 2 billion
- Maximum number of key entries: greater than 2 billion
- Key length: 1 to 255, default value 30 characters maximum
- Maximum number of indexes per database: 254, default value 100
- Range for data record length: 21 bytes to greater than 2 gigabytes (limited by DOS and the 8086 architecture to 65,520 bytes)
- Maximum number of workstations: 65,534, default value 50

B-Tree Filer is TurboPower's version of a solid and respected product from Germany: B-Tree Isam. That product has earned a well-deserved reputation for speed and dependability. The original author was Dipl. Math. Ralf Nagel. The product was further developed and marketed by Enz EDV-Beratung GmbH. TurboPower Software Company purchased complete rights to the product in 1995. Besides translating it into English, TurboPower has improved the original package by including the new virtual sort and network utilities.

If you upgraded to the current version of B-Tree Filer from version 5.0, be sure to read Appendix C. As a result of enhancements, it is not source-compatible with earlier versions.

A. Requirements

To use B-Tree Filer, you must have the following:

1. An IBM PC, XT, AT (or close compatible), or PS/2 running DOS 3.1 or later or Windows 3.0 or later.
2. Turbo Pascal version 6.0 or 7.0, Turbo Pascal for Windows 1.5, or Borland Pascal 7.0.
3. A hard . Installation and compilation of all files will consume about 5MB of disk space.
4. The memory required is strongly influenced by the way you configure and use the supplied units. The main FILER unit typically uses 64KB to 128KB of memory for code and data.

Optional:

5. Borland's Turbo Assembler (TASM) or Microsoft's Macro Assembler (MASM). Needed only if you wish to modify the assembly language source code for certain portions of B-Tree Filer.
6. Network hardware and operating system compatible with B-Tree Filer. Examples include Novell NetWare, Artisoft LANtastic, IBM PC LAN, and Microsoft MS-NET. Any network that supports DOS 3.1 record locking calls will usually work with B-Tree Filer.

A network must provide two basic capabilities in order to work in multi-user mode with B-Tree Filer:

- the ability to define a disk storage area where files can be shared among various workstations
- the ability for a given workstation to lock all or part of a file, so that attempted access by another workstation leads to a detectable error

These requirements, and the B-Tree Filer support for different networks, are described in more detail in Chapter 2.

Although B-Tree Filer handles the details of B-tree management, the documentation assumes a basic understanding of record files and indexing techniques. Chapter 3 provides a brief explanation of B-tree data and index management. You should also have a basic understanding of DOS file access techniques and error codes.

The manual does not provide an in-depth tutorial on multi-user programming, although the discussions in Chapter 4 go some way towards explaining the issues and solutions. Neither does the documentation describe all of the background for accessing network protocols such as provided by NetBIOS and NetWare. Where possible, you should obtain the references mentioned in the appropriate chapters for further background.

B. Installation

B-Tree Filer is distributed on 1.44MB 3.5" diskettes. The number of diskettes is subject to change. They are labeled DISK3-1, DISK3-2, and so on. B-Tree Filer can be installed directly from diskettes, or you can copy the complete set of files to your hard disk and run the installation from there.

We'd like to draw your attention to several files in particular, all on the first disk:

READ.ME

Read this file before installing. It contains last minute information that may affect how you install the product.

READ.1ST

This file is in the archive FILES1.LZH, which the installation program dearchives for you. Print and read this file before using the product. It contains the latest updates to this documentation and may contain new features, clarifications, or corrections.

FASTUPD.nnn

Describes changes in this version of B-Tree Filer compared to the most recent previous one. The file extension is the current version number, for example, 550 means version 5.50.

LHA.EXE

A freeware compression/decompression utility used to compress and decompress files with extension LZH. Type LHA to get a summary of its options. LHA213.DOC contains the complete documentation of LHA.

PACKING.LST

A complete list of all files on all the diskettes, including the contents of the archives.

PRINTDOC.EXE

This file is in the archive FILES1.LZH, which the installation program dearchives for you. PRINTDOC.EXE prints text files with pagination, margins, and headers. This small utility is useful for printing files like READ.ME. Type PRINTDOC to get a list of options.

INSTALL.EXE

The installation program.

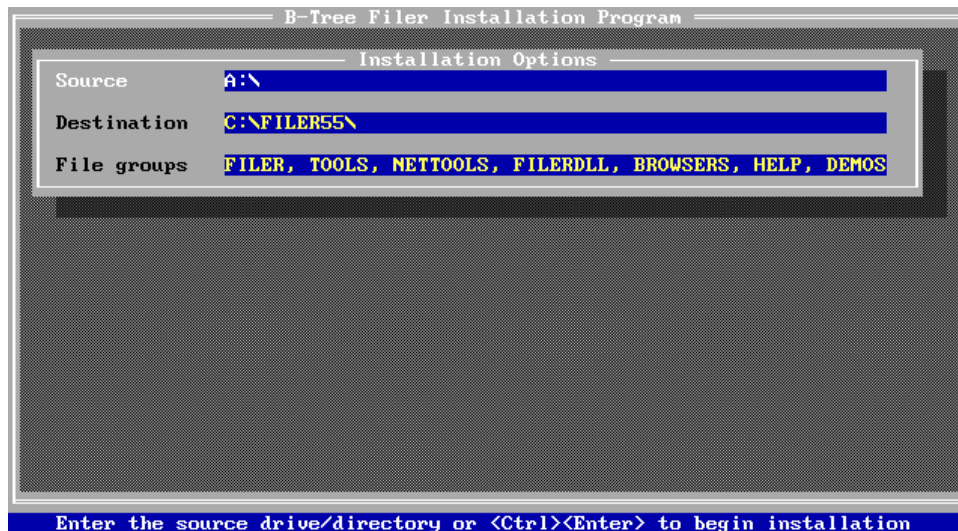
The Installation Process

INSTALL is an interactive tool that makes it easy for you to dearchive the contents of the archives on the B-Tree Filer disks. With it, you can select only as many B-Tree Filer files as you actually need.

If you're installing directly from diskettes, insert the first disk in drive A: and enter

```
A:INSTALL
```

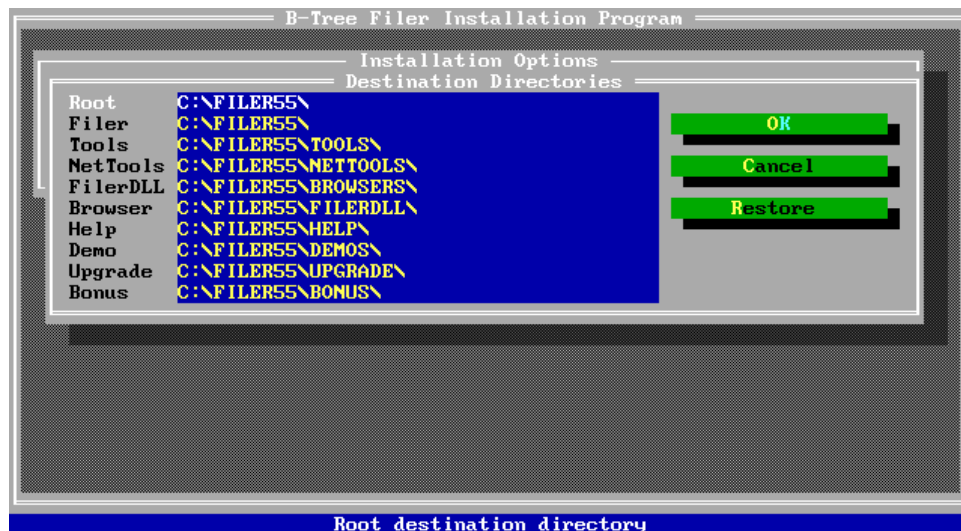
The main window is displayed:



'Source' refers to the disk you will install from (where the compressed LZH files are located). This defaults to the drive where INSTALL.EXE was located.

'Destination' specifies the directory where B-Tree Filer will be installed. This is where the decompressed files will be placed. The default is \FILER55 on the current drive. If the directory does not exist, INSTALL creates it for you.

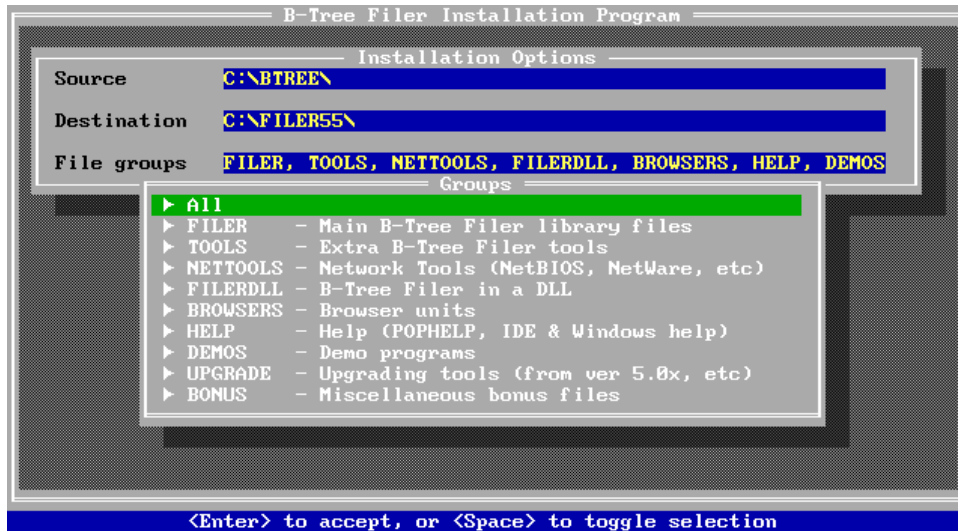
Many B-Tree Filer files are placed in subdirectories of this base directory. If you want to change the base or any of the subdirectories, use the cursor keys to position the highlight bar over Destination and press <Enter>. A new dialog is displayed:



Move the highlight bar to any of the directory names and enter the desired text. If you change the base directory, the other lines are updated relative to the new base. It is recommended, but not required, that you use a directory structure similar to the default one.

To restore the directories to their default values, select the Restore button. To accept the changes you made, select the OK button. To throw away the changes, select <Esc> or the Cancel button.

Back at the main installation screen, 'File groups' allows you to specify which parts of B-Tree Filer are installed. The current selections are displayed in the string that appears on the main screen. By default, all files are installed. To change this, move the highlight bar to the field and press <Enter>. A selection list is displayed:



Selected file groups are marked with '>'. To change the selections, cursor up or down to an item and press <Space> to toggle it. 'All' reselects all file groups. Press <Enter> to accept your choices.

When you are finished specifying install options, press <Ctrl><Enter> to start the installation. INSTALL prompts you for diskette changes if needed.

If you install all of B-Tree Filer using the default directory structure, your directory tree will look like the following:

```

L-FILER      READ.ME, READ.1ST, FASTUPD.nnn, PACKING.LST,
makefile, and all
|
|           the source code for the basic B-Tree Filer
functions
|--TOOLS     Source code for miscellaneous database tool modules
|--NETTOOLS  Source code for network-specific modules
|--FILERDLL  Dynalink Libraries (DLLs) for B-Tree Filer
|--BROWSERS  Source code for all browser units
|--HELP      POPHELP, MAKEHELP, BTFILER.HLP, BTFILER.TPH,
BTFWIN.HLP, help
|
|           text files
|--UNITS     Initially empty, the make file compiles all units
here
|--DEMOS     Demonstration programs
|--UPGRADE   UPGRADE program for converting old source files (see
Appendix C
|
|           for Borland Database Toolbox, Appendix D for B-Tree
Filer 5.0x)
L-BONUS     Bonus files, see READ.ME! in this directory

```

If you'll be working from another directory while using the B-Tree Filer units, be sure to add the required directories to the IDE's search paths for Include Directories and Unit Directories. The required paths are \FILER and \FILER\UNITS respectively. If you use the command line compiler, add the following lines to your configuration file:

```

-Ux:\FILER\UNITS
-Ix:\FILER

```

where the x: denotes the drive where the \FILER directory is located.

Rebuilding B-Tree Filer Units

A make file (FILER.MAK) is provided for use with Borland's standalone MAKE utility. There's no requirement for you to use this file; it is provided primarily as a convenient way to build all the TPU and OBJ files at once. You will need to edit the make file to specify the location and version of your Turbo Pascal compiler, to define your directory structure, to define the target environment for the rebuild, and so on. The make file contains instructions on how to perform any needed modifications.

B-Tree Filer is not shipped with precompiled units due to the large number of combinations possible (compiler version, real mode vs. protected mode vs. Windows target, compilation defines, etc.). As a result, you must build the library before you can write and compile your own applications. The easiest way is to use the supplied make file. If you did not install all of B-Tree Filer's source files, you must alter the FILER.MAK and/or BTDEFINE.INC files so that the B-Tree Filer units can be built without error.

Before running the make file, you must specify the correct compiler defines in BTDEFINE.INC (see `_2.A` for more details). You must also alter the various make macros in FILER.MAK so that you are compiling for the correct target and to define the correct directory structure. See the FILER.MAK file for more information.

FILER.MAK uses the command line compiler for your version of Turbo Pascal. You must specify the correct path for it (see the `DirCompiler` macro). The command line compiler will attempt to find a configuration file in this directory (BPC.CFG, TPC.CFG, or TPCW.CFG, depending on your version of Turbo Pascal). This is where some problems might occur:

- The CFG file is missing. The MAKE file aborts because it cannot find any standard Pascal units. You must create a CFG file with at least a `/U` directive that points to your units directory (for example, this is usually `C:\BP\UNITS` for BP7).
- The CFG file has a `/U` line that points to a directory that contains an earlier version of the B-Tree Filer units (for example, you are using BP7 and have moved all the B-Tree Filer units to `C:\BP\UNITS`). The make file will probably abort with a unit version mismatch error. You must ensure that no older B-Tree Filer units can be found.
- The command line created by the make program to compile a unit or program is too long. You must edit your CFG file to transfer some of the path information (i.e., the `/U` or `/I` paths) from the make file to the CFG file. Edit the make file to remove these paths. For example, all of the make statements that do the actual compiling have:

```
-u$(DirUnits) -i$(DirBase)
```

The `DirUnits` macro expands to `\FILER\UNITS` and the `DirBase` macro expands to `\FILER` by default. Remove these from the FILER.MAK file and add the following to your CFG file:

```
/u\FILER\UNITS  
/i\FILER
```

To run FILER.MAK, change to the \FILER directory and type the following DOS command line (note that -f must be lowercase):

```
MAKE -fFILER
```

The make file is designed so that all the general B-Tree Filer units (files with extension TPU, TPP or TPW) are placed in the \FILER\UNITS directory.

The demo program DEWDEMO is not automatically compiled by the FILER.MAK file. To recompile it you must have Data Entry Workshop and you must recompile it manually, either in the IDE or using the command line compiler.

Units Provided

Here's a brief overview of all the units provided by B-Tree Filer:

FILER

This unit is the core of the database management provided by B-Tree Filer.

EMSHEAP

A heap manager for EMS memory. Used by the FILER unit in real mode to keep index page buffers in EMS.

VREC

Variable length data record support.

RESTRUCT/REINDEX

Restructures a database, allowing fields to be added or deleted. Also provides facilities to import a database, remove deleted record space, and reindex the file.

REBUILD/VREBUILD

Regenerates a database by removing deleted record space and creating a clean index. These units are provided for compatibility with older B-Tree Filer versions. You should use RESTRUCT and REINDEX whenever possible.

REORG/VREORG

Similar to REBUILD and VREBUILD, but allow you to import a database or add/delete fields from an existing database. These units are provided for compatibility with older B-Tree Filer versions. You should use RESTRUCT and REINDEX whenever possible.

FIXTOVAR

Converts from fixed length records to variable length records.

NUMKEYS

Converts numeric values to sortable key strings.

ISAMTOOL

Miscellaneous routines that extend the FILER unit.

MSORT/MSORTP

Sorts data items using a merge sort algorithm. MSORT is optimized for real mode and MSORTP for protected mode.

DBIMPEXP

Provides import and export of dBase files.

8 Installation

BROWSER

General purpose browser that allows you to select records from a database with a scrolling list.

LOWBROWS, MEDBROWS, HIBROWS

Object-oriented abstract browser.

OPBROW

Object Professional compatible browser.

TVBROWS

Turbo Vision compatible browser.

WBROWSER

ObjectWindows Library compatible browser.

NWBASE

Provides low-level access to a NetWare server from a workstation.

NWCONN

Provides access to a workstation's NetWare connection information.

NWBIND

Provides access to the NetWare server bindery.

NWMSG

Provides access to NetWare's broadcast message facilities.

NWFILE

Provides access to NetWare's file and directory services.

NWSEMA

Provides access to NetWare's semaphore services.

NWTTS

Provides access to NetWare's Transaction Tracking Services (TTS).

NWPRINT

Provides access to NetWare's printing services.

NWIPXSPX

Provides access to NetWare's IPX and SPX services.

NETBIOS

Inter-workstation communication using the NetBIOS interface.

SHARE

Assorted network related functions, including file locking.

ISCOMPAT/VRCOMPAT

Compatibility routines that make the current FILER and VREC units look more like the FILER and VREC units of version 5.0. See Appendix C for more information.

Demonstration Programs Provided

B-Tree Filer provides a variety of demonstration programs. The following list gives a brief description of each of them.

NETDEMO

Shows a typical application using the FILER and BROWSER units. Provides an address database with commands to add and delete records, search, rebuild, print, and so on.

OPISDEMO

Demonstrates the OPBROW browser, based on TurboPower's Object Professional user interface toolkit. An EXE file is provided; in order to recompile OPISDEMO, you need Object Professional.

TVISDEMO

Demonstrates the TVBROWS browser, based on Borland's Turbo Vision. An EXE file is provided; in order to recompile TVISDEMO, you must have Turbo Vision and a Borland compiler that can compile for a DOS target (real or protected mode).

OWDEMO/BTWDemo/DEWDEMO

Demonstrate the WBROWSER browser, based on Borland's ObjectWindows Library. An EXE file is provided for OWDEMO; in order to recompile you must have a Borland compiler that can create Windows programs. For DEWDEMO, you must have TurboPower's Data Entry Workshop. BTWDemo and DEWDEMO are not documented further.

BIGSORT

Sorts text files, demonstrating the merge sort unit MSORT. This program is not documented in this manual. For details, see the source code.

DB2ISAM/ISAM2DB

Demonstrate the use of the DBIMPEXP unit. DB2ISAM converts any dBase file to a B-Tree Filer fileblock. ISAM2DB converts an example B-Tree Filer fileblock to dBase format.

NETINFO

Demonstrates many of the network-specific functions to provide information about the network where it is run.

NBSEND/NISEND/NSEND

Offer workstation to workstation file transfer for NetBIOS session services, NetWare IPX services, and NetWare SPX services, respectively.

SPX2WAY

Shows how to set up a two-way NetWare SPX connection used for an interactive chat program.

BINDLIST

Lists all the objects in the bindery for a server.

TTSFILER

Shows a method of using NetWare's Transaction Tracking Services in a B-Tree Filer application.

C. Database Demonstration Programs

To get a feel for the capabilities of B-Tree Filer, you'll probably want to try out one or more of the database demonstration programs. All of them are supplied precompiled so you can browse even if you don't have a particular library. NETDEMO.EXE provides a simple DOS text mode user interface based on only the modules supplied with B-Tree Filer and your compiler runtime libraries. OPISDEMO.EXE is another DOS text mode interface based on Object Professional. TVISDEMO.EXE is a similar program based on Borland's Turbo Vision. And OWDEMO.EXE is a Windows application based on ObjectWindows Library.

The supplied version of NETDEMO.EXE is compiled using the B-Tree Filer dynamic network capabilities; that is, it can run in single user mode or on any of the networks that B-Tree Filer supports.

NETDEMO has small data and index files ready for you to inspect or change. To do so, be sure that NETDEMO.EXE, ADDRESS.DAT and ADDRESS.IX are in the current directory. Then call NETDEMO with one of the following command line options:

```
/D      No network
/N      Novell NetWare (not NetWare Lite or Personal NetWare!)
/M      MS-Net or compatible
```

You can get a summary of the options by just typing NETDEMO at the DOS command line. Use the /M option if you are running NetWare Lite or Personal NetWare.

NETDEMO shows off various capabilities of B-Tree Filer, including:

- single- or multi-user operation in the same program
- multiple key types in one index file
- save mode to guard data integrity
- database browsing
- high-level facility to purge deleted records from data files and rebuild indexes

NETDEMO manages an address file with the following fields:

```
PersonDef = record
  FirstName : String[15];
  LastName  : String[15];
  Company   : String[25];
  Address   : String[25];
  City      : String[15];
  State     : String[02];
  Zip       : String[10];
  Telephone : String[12];
  MemoLen   : Word;
  Memo      : variable length buffer of 1-1200 bytes
end;
```

Two index keys are used to find records quickly. The key (one that does not allow duplicates) is a combination of LastName and FirstName. The second key is the zip code, and it does allow duplicates.

The NETDEMO program lets you add, modify, or delete records, browse through a list of the records sorted by either key, search on either key, print the records, and rebuild the indexes after purging deleted records. NETDEMO uses a full-screen display to make all these operations easy.

B-Tree Filer Demo Program			Main Menu		Key: Last Name	
Zip	Name	Company	Address	City		
98582	Asber, Ed	Ashton Tate	28181 Hamilton Ave.	Torrance		
11111	Bush, George	Office of the Presi	1600 Pennsylvania A	Washington		
98889	Button, Jim	ButtonWare Inc.	P.O. Box 96058	Bellevue		
77878	Canion, Rod	Compaq Computer Cor	2855 FM 149	Houston		
10816	Dvorak, John C.	PC Magazine	One Park Ave.	New York		
88949	Folks at, The	TurboPower Software	P0 Box 49889	Colo Spgs		
98873	Gates, Bill	Microsoft Corp.	16011 NE 36th Way	Redmond		
67890	Goeshere, Your Name	QuickTurboPerfect I	12345 Main St.	Anytown		
68123	Jackson, Jesse	Rainbow Coalition H	Try Again Inn #1992	Chicago		
95999	Jobs, Stephen	NeXT Inc.	123 Novel Idea Lane	Palo Alto		
95866	Kahn, Philippe	Borland Internation	1800 Green Hills Ro	Scotts Valle		
95555	Lucas, George	Industrial Light &	33 Some Kinda Place	Marin		
97520	Mace, Paul	Paul Mace Software	400 Williamson Way	Ashland		
82142	Manzi, Jim	Lotus Development C	55 Cambridge Parkwa	Cambridge		
12121	North, Oliver	Shredders R Us	1 Contra Court	Suburban		
98483	Norton, Peter	Peter Norton Comput	2210 Wilshire Blvd.	Santa Monica		
84857	Petersen, Pete	WordPerfect Corp.	1555 N. Technology	Orem		
83458	Pournelle, Jerry	BYTE Magazine	One Phoenix Mill La	Peterborough		
77871	Presley, Elvis	c/o Jerry Glanville	123 Fruitcake Road	Houston		
11111	Quayle, J. Danforth	Office of the U.P.	1600 Pennsylvania A	Washington		
F2-Add F3-Del F4-Find F5-Key F6-Filter F8-Print F9-Info F10-Rebuild Esc-Quit						

NETDEMO first prompts whether to operate in "Save Mode." In this mode, B-Tree Filer is able to recover from system or network crashes without losing data. This reliability comes at the expense of speed, but the difference will be negligible for interactive data entry.

NETDEMO attempts to open an existing data set in the files ADDRESS.DAT and ADDRESS.IX. If these are found, NETDEMO proceeds. Otherwise, it asks if you want to create a new (empty) data set. A small set of addresses is supplied to allow easy exploration of NETDEMO.

If you are creating a new data set, choose F2 to create a new record. Enter the appropriate record fields, and then press <Ctrl><Enter> to exit the record editor. You are prompted to either save the new record or discard it. Remember that the key formed by combining LastName and FirstName must be unique among all the records in the data set. (Using a name as a primary key isn't common because of the relatively high probability of duplication. NETDEMO does it to avoid having to create an artificial field like a "customer number." If you encounter a duplicate key problem with NETDEMO, you must modify one of the duplicate names.)

When the data set has at least one record, NETDEMO displays a full screen browse window. You can use the <Up> and <Down> arrow keys to browse among the records, the <Left> and <Right> arrows to scroll horizontally to see all the fields, and other cursor keys to reposition the highlight bar, which indicates the currently selected record. You can modify or delete the selected record using other NETDEMO commands.

While the browser is active, there is a menu bar at the bottom of the screen. This indicates many of the function keys you can press to activate various NETDEMO functions (see the menu bar on the screen above). Here is a brief description of each command:

- <F1> Modify
Edit the contents of the currently selected record. NETDEMO will reindex the record when you finish editing. Remember that the record must retain a unique primary key. When you're done, press <Ctrl><Enter> to accept any changes, or <Esc> to discard them and retain the previous version. You can also modify a record by pressing <Enter> while the highlight bar is over it.
- <F2> Add
Add a new record. Press <Ctrl><Enter> to accept the new entry, or <Esc> to discard it.
- <F3> Delete
Delete the currently selected record. NETDEMO requires you to confirm your choice.
- <F4> Find
Search for a record. NETDEMO presents a full-screen search template where you can enter search parameters in any desired fields. NETDEMO's searching is not case sensitive, and matches do not have to be complete. For example, entering 'IB' in the company name field matches records like 'IBM' or 'Ibsen Baking Company'.
- If the search fields you enter are part of NETDEMO's index fields (LastName and ZipCode), then NETDEMO will do a fast indexed search to find the initial match, possibly followed by a sequential search to qualify any other match fields. If NETDEMO does not find a complete match, it positions the browse cursor on the closest approximate match.
- <F5> Select Key
By default, NETDEMO presents records sorted by the primary key (LastName and FirstName). Use <F5> to switch to the other key, which will display the records in zip code order. The currently active index is displayed on the top line of the screen.
- <F6> Filter
Display only selected records. NETDEMO presents a filter template where you can enter search parameters for any desired fields. Only the records that match the non-empty fields are displayed in the browse screen. For example, if you enter '9' in the zip code field, only those records whose zip code start with 9 are displayed. Press <F6> again to disable filtering.
- When filtering is active, the scroll bar disappears. That is because the records displayed when filtering is enabled might be widely separated in the key sequence, or might cover a very narrow range of the entire sequence. Therefore the scroll bar doesn't have much meaning. Disabling the scroll bar in this situation is your choice. The BROWSER unit will continue to display the scroll bar until you explicitly disable it.
- <F8> Print
Print all the records. When you press <F8>, NETDEMO first displays a small menu. Here you can choose to print the records in LastName or ZipCode order, or to cancel the print request. If you proceed, NETDEMO writes all records (one line per record) to the default printer PRN. While NETDEMO is printing, you can press any key to abort the print job.

<F9> Info

Information about the current address data set appears on the bottom line of the screen. This includes the total number of records in the data file, the number that have been deleted, the database mode (Normal or Save) and the current workstation number. Press any key to continue.

Note that when variable length records are used, the record count can be a little misleading. Instead of the number of logical records in the database it is the number of fixed length record "sections" (see _6.B for more information).

<F10> Rebuild

Rebuild the data and index files using just the data file. The files are first closed, and then the data file is read in sequential order. Previously deleted records are purged from the data file, and the indexes are rebuilt from scratch. Rebuild is possible only if no other workstation is currently using the fileblock.

<Esc> Exit

Use this command to exit back to DOS. NETDEMO offers an opportunity for you to change your mind.

NETDEMO Design

NETDEMO uses a simple approach to data locking in a multi-user environment. When it needs to guarantee access to a particular record, NETDEMO simply locks the entire database (with BTLockAllOpenFileBlocks) for the shortest period of time possible. While reading records, it first checks for other locks on them. If a record is locked, NETDEMO prompts whether to try again, and aborts the operation if you request. (The actual locking process is more complicated than this superficial introduction. See _4.D for the full story.)

NETDEMO buffers the currently selected record in memory. This minimizes the amount of additional reading required when you modify a record. However, additional care is required when a modified record is to be written back to disk. Has the record been deleted in the meantime? Has someone else changed it? See _4.E and NETDEMO's ChangePerson function for the techniques used to handle these possibilities.

NETDEMO also demonstrates the VREC, BROWSER, and VREBUILD units. These high level modules make it easy to provide full-screen record displays and automatic rebuild functions.

The OPISDEMO program demonstrates how B-Tree Filer can be used with Object Professional. OPISDEMO is very similar to NETDEMO--it performs the same functions, but works in the window hierarchy of Object Professional. It can run in single user mode or on any of the networks that B-Tree Filer supports.

OPISDEMO uses similar data and index files to NETDEMO (see NETDEMO above for a description of ADDRESS). Be sure that OPISDEMO.EXE, ADRESSEN.DAT and ADRESSEN.IX are in the current directory. Then call OPISDEMO with one of the following command line options:

```

/O      No network
/N      Novell NetWare (not NetWare Lite or Personal NetWare!)
/M      MS-Net or compatible

```

You can get a summary of these options by just typing OPISDEMO at the DOS command line.

OPISDEMO attempts to open the existing data set ADRESSEN.DAT and ADRESSEN.IX. If these files are found, OPISDEMO proceeds. Otherwise, it asks if you want to create a new (empty) data set. A small set of addresses is supplied to allow easy exploration of OPISDEMO.

If you are creating a new data set, choose F2 to create a new record. Enter the appropriate record fields, and then press <Ctrl><Enter> to save the new record. Remember that the key formed by combining LastName and FirstName must be unique among all the records in the data set. If you encounter a duplicate key problem with OPISDEMO, you must modify one of the duplicate names.

When the data set has at least one record, OPISDEMO displays a full screen browse window:

LastName	FirstName	Telephone	Country/Zip/City
Abel	Oswald	06206-8889	D-68623 Lamp
Andersen	Bob	0431-9958	D-24149 Kiel
Aumann	Silvia	04441-9938	D-49377 Vech
Bach	Margot	06251-9016	D-64625 Bens
Bach	Marion	06251-887968	D-64625 Bens
Bader	Kim	07431-556654	D-72461 Albs
Bauer	Immanuel		D-44139 Dort
Becker	Steffi	0511-5809	D-30627 Hann
Bein	Toni	040-13368	D-22587 Hamb
Brenner	Robert	030-2560	D-12203 Berl
Bresser	Peter		D-27389 Stem
Bugatti	Enzo	06183-31584	D-63543 Neub
Bungert	Carla	06423-6142	D-58300 Wett
Clausewitz	Rüdiger	00352-7691	L-2213 Luxe
Columna	Isabelle	05634-2039	D-34513 Wald

Sorted by Name

^F1 more | F2 New F3 Delete F4 Search F5 Edit F7 Key F8 Status ESC Quit

You can use the <Up> and <Down> arrow keys to browse among the records, the <Left> and <Right> arrows to scroll horizontally to see all the fields, and other cursor keys to reposition the

highlight bar, which indicates the currently selected record. You can modify or delete the selected record using other OPISDEMO commands.

While the browser is active, there is a menu bar at the bottom of the screen. This indicates many of the function keys you can press to activate various OPISDEMO functions. All commands can also be selected by positioning the mouse cursor on the command and clicking the left mouse button. Here is a brief description of each command:

<Ctrl><F1> More

Display more commands. Additional commands are displayed on the menu bar. Press <Ctrl><F1> again to see the original set of commands.

<F2> New

Add a new record. Press <Ctrl><Enter> to save the new record, or <Esc> to cancel.

<F3> Delete

Delete the currently selected record. OPISDEMO requires you to confirm your choice.

<F4> Search

Search for a record. OPISDEMO presents a search template and allows you to enter either the last name and first name or the zip code. OPISDEMO's searching is not case-sensitive, and matches do not have to be complete. For example, entering 'IB' in the company name field matches records like 'IBM' or 'Ibsen Baking Company'. If OPISDEMO does not find a complete match, it attempts to position the browse cursor on the closest approximate match.

<F5> Edit

Edit the contents of the currently selected record. OPISDEMO will reindex the record when you finish editing. Remember that the record must retain a unique primary key. When you're done, press <Ctrl><Enter> to save the changes or <Esc> to discard them.

<F7> Key

By default, OPISDEMO presents records sorted by the primary key (LastName and FirstName). Use <F7> to switch to the other key, which will display the records in zip code order.

<F8> Status

Information about the current address data set is displayed. This includes the total number of records in the data file, the number of records that have been deleted, and the network type. Press <Enter> to continue.

<Esc> Quit

Use this command to exit back to DOS. OPISDEMO offers an opportunity for you to change your mind.

<F9> Reindex

Rebuild the data and index files using just the data file. The files are first closed, and then the data file is read in sequential order. Previously deleted records are purged from the data file, and the indexes are rebuilt from scratch. Reindex is possible only if no other workstation is currently using the fileblock.

<Alt><F10> Info

Display copyright information about OPISDEMO. Press <Enter> to continue.

The TVISDEMO program demonstrates how B-Tree Filer can be used with Turbo Vision. TVISDEMO is very similar to NETDEMO. It can run in single user mode or on any of the networks that B-Tree Filer supports.

TVISDEMO uses the same data and index files as OPISDEMO (see OPISDEMO above for a description of ADRESSEN). Be sure that TVISDEMO.EXE, ADRESSEN.DAT and ADRESSEN.IX are in the current directory. Then call TVISDEMO with one of the following command line options:

```
/O      No network
/N      Novell NetWare (not NetWare Lite or Personal NetWare!)
/M      MS-Net or compatible
```

You can get a summary of these options by just typing TVISDEMO at the DOS command line.

TVISDEMO first prompts whether to operate in "Save Mode." In this mode, B-Tree Filer is able to recover from system or network crashes without losing data. This reliability comes at the expense of speed, but the difference will be negligible for interactive data entry.

Press <Alt> to start the browser. TVISDEMO attempts to open an existing data set in the files ADRESSEN.DAT and ADRESSEN.IX. If these are found, TVISDEMO proceeds. Otherwise, it asks if you want to create a new (empty) data set. A small set of addresses is supplied to allow easy exploration of TVISDEMO.

If you are creating a new data set, choose <Alt><N> to create a new record. Enter the record fields, and then press <Enter> or select OK to save the new record. Remember that the key formed by combining LastName and FirstName must be unique among all the records in the data set. If you encounter a duplicate key problem with TVISDEMO, you must modify one of the duplicate names.

When the data set has a least one record, TVISDEMO displays a browse window:

LastName	Telephone	Zip	City
Abel, Oswald	06206-8889	D-68623	Lampertheim
Andersen, Bob	0431-9958	D-24149	Kiel
Aumann, Silvia	04441-9938	D-49377	Vechta
Bach, Margot	06251-9816	D-64625	Bensheim
Bach, Marion	06251-887968	D-64625	Bensheim
Bader, Kim	07431-556654	D-72461	Albstadt
Bauer, Immanuel		D-44139	Dortmund
Becker, Steffi	0511-5809	D-30627	Hannover
Bein, Toni	040-13368	D-22587	Hamburg
Brenner, Robert	030-2560	D-12203	Berlin
Bresser, Peter		D-27389	Stemmen
Bugatti, Enzo	06183-31584	D-63543	Neuberg
Bungert, Carla	06423-6142	D-58300	Wetter
Clausewitz, Rüdiger	00352-7691	L-2213	Luxemburg
Columna, Isabelle	05634-2039	D-34513	Waldeck

Alt-X Exit Alt-B Browser Alt-F3 Close

You can use the <Up> and <Down> arrow keys to browse among the records, the <Left> and <Right> arrows to scroll horizontally to see all the fields, and other cursor keys to reposition the highlight bar, which indicates the currently selected record. You can modify or delete the selected record using other TVISDEMO commands.

While the browser is active, there is a status line at the bottom of the browse window. This indicates the various TVISDEMO functions. Commands are selected by pressing Alt and the highlighted letter in the command. All commands can also be selected by positioning the mouse cursor on the command and clicking the left mouse button. The commands are:

<Alt><N> New

Add a new record. Press <Enter> to save the new record, or <Esc> to cancel.

<Alt><D> Delete

Delete the currently selected record. TVISDEMO requires you to confirm your choice.

<Alt><S> Search

Search for a record. TVISDEMO presents a search template and allows you to enter either the last name and first name or the zip code. TVISDEMO's searching is not case-sensitive, and matches do not have to be complete. For example, entering 'IB' in the company name field matches records like 'IBM' or 'Ibsen Baking Company'. If TVISDEMO does not find a complete match, it attempts to position the browse cursor on the closest approximate match.

<Alt><E> Edit

Edit the contents of the currently selected record. TVISDEMO will reindex the record when you finish editing. Remember that the record must retain a unique primary key. When you're done, press <Enter> to save the changes or <Esc> to discard them.

<Alt><O> Sort

By default, TVISDEMO presents records sorted by the primary key (LastName and FirstName). Use <Alt><O> to switch to the other key, which will display the records in zip code order.

<Alt><T>

Status

Information about the current address data set is displayed. This includes the total number of records in the data file, the number of records that have been deleted, the record size, the save mode, and the network type. Press <Enter> to continue.

<Alt><X>

Exit

Use this command to exit back to DOS.

The OWDEMO program demonstrates how B-Tree Filer can be used in the Windows environment with ObjectWindows Library. OWDEMO is very similar to NETDEMO. It can run in single user mode or on any of the networks that B-Tree Filer supports.

OWDEMO uses the same data and index files as OPISDEMO (see OPISDEMO above for a description of ADRESSEN.DAT and ADRESSEN.IX). Execute OWDEMO.EXE and then choose the mode: single user, MS-Net/Share, or Novell.

OWDEMO attempts to open an existing data set in the files ADRESSEN.DAT and ADRESSEN.IX. If the files are found, OWDEMO proceeds. Otherwise, it asks if you want to create a new (empty) data set. A small set of addresses is supplied to allow easy exploration of OWDEMO.

If you are creating a new data set, choose <Alt><N> to create a new record. Enter the appropriate record fields, and then press <Enter> to save the new record. Remember that the key formed by combining LastName and FirstName must be unique among all the records in the data set. If you encounter a duplicate key problem with OWDEMO, you must modify one of the duplicate names.

When the data set has at least one record, OWDEMO displays a browse window.

You can use the <Up> and <Down> arrow keys to browse among the records, the <Left> and <Right> arrows to scroll horizontally to see all the fields, and other cursor keys to reposition the highlight bar, which indicates the currently selected record. You can modify or delete the selected record using other OWDEMO commands.

While the browser is active, there is a menu bar at the top of the browse window. This indicates the various OWDEMO functions. Commands are selected by pressing Alt and the letter that is highlighted in the command. All commands can also be selected by positioning the mouse cursor on the command and clicking the left mouse button. Here is a brief description of each command:

<Alt><A> About

Display copyright information about OWDEMO. Press <Enter> to continue.

<Alt><N> New

Add a new record. Press <Enter> to save the new record, or <Esc> to cancel.

<Alt><D> Delete

Delete the currently selected record. OWDEMO requires you to confirm your choice.

<Alt><S> Search

Search for a record. OWDEMO presents a search template and allows you to enter either the last name and first name or the zip code. OWDEMO's searching is not case-sensitive, and matches do not have to be complete. For example, entering 'IB' in the company name field matches records like 'IBM' or 'Ibsen Baking Company'. If OWDEMO does not find a complete match, it attempts to position the browse cursor on the closest approximate match.

<Alt><E> Edit

Edit the contents of the currently selected record. OWDEMO will reindex the record when you finish editing. Remember that the record must retain a unique primary key. When you're done, press <Enter> to save the changes or <Esc> to discard them.

<Alt><F> Filter on

Display only those records that fit the filter.

<Alt><T> Status

Information about the current address data set is displayed. This includes the total number of records in the data file and the workstation ID number. Press <Enter> to continue.

D. The Help System

POPHELP is a memory-resident program designed to provide easy-to-access, online help for the B-Tree Filer library. Since it is based on the TSR swapping capabilities of Object Professional, it occupies as little as 8KB of DOS memory. POPHELP uses help files generated by the MAKEHELP utility, also provided with B-Tree Filer, so you can merge help files from different TurboPower products or to extend the help yourself.

If you are using the Borland Pascal 7.0 DOS IDE, you can choose to integrate B-Tree Filer help text directly into the IDE help system instead of using POPHELP. The section entitled "Installing the BP7 Help File" below describes how to include the B-Tree Filer BP7 help file into the IDE.

If you are using B-Tree Filer in a Windows environment you have another option: using the B-Tree Filer Windows help file. This is installed in your help directory by B-Tree Filer's install program. The Windows help file is called BTFWIN.HLP and by default is installed in the \FILER\HELP directory. See your Microsoft documentation on how to load the help file into the Windows Help program.

Installing the BP7 Help File

A Borland Pascal 7.0 IDE help file (BTFILER.TPH) is provided with B-Tree Filer. You can add it to the IDE help system as follows:

1. Start the IDE.
2. Pull down the help menu.
3. Select Files... from the help menu. The Install Help Files dialog box is displayed.
4. Select New. The Help Files dialog box is displayed.
5. Enter the path and file name for BTFILER.TPH (the default is \FILER\HELP\BTFILER.TPH).
6. Select OK to add the help file.
7. Select OK to store the help system configuration.

The B-Tree Filer help file is now integrated into the IDE help system and available for your use.

Installing POPHELP

POPHELP can be installed in several ways:

- as a normal TSR using 150-200K of RAM
- as a TSR that swaps itself to EMS (expanded) or XMS (extended) memory, leaving only an 8K kernel behind in normal RAM
- as a TSR that swaps itself to disk, leaving an 8K kernel in RAM
- as a shell that execs another program and then automatically removes itself from memory.

You select among these options by using POPHELP command line parameters.

A help file for B-Tree Filer is supplied with this version of POPHELP. This help file contains information about all interfaced constants, types, variables, procedures, and functions in the library. The help file is supplied in source format so that you can add to it as necessary. Compiling it is a simple one-step operation.

The installation procedure described in _1.B stored the following files in your \FILER\HELP directory:

```
POPHELP.EXE
MAKEHELP.EXE
BTFILER.TXT
*.TXT
BTFILER.HLP
```

BTFILER.TXT is the main help file, which references all of the other files with extension TXT. If you need to make changes or additions to the help text, you must recompile it by entering the following at the DOS command line:

```
MAKEHELP /Q BTFILER
```

Once you've created the new BTFILER.HLP file, you can delete the *.TXT files from your disk, unless of course you intend to modify them further. You can also delete MAKEHELP.EXE. This leaves two files related to help, POPHELP.EXE and BTFILER.HLP.

This version of POPHELP is set up to load the BTFILER.HLP file by default. Hence, in the simplest case, you make POPHELP memory resident by entering

```
POPHELP
```

at the DOS command line while BTFILER.HLP is in the current directory. This installs POPHELP in swapping mode; it uses EMS memory (about 250K), XMS memory (about 600K, unless /1 is used) or disk space (two files, unless /1 is used, each about 250K) for swapping and retains about 8K of RAM (10K if /1 is used).

POPHELP provides a number of options to control how it goes resident. It uses the following DOS command line syntax:

```
POPHELP [HelpFile] [Options]
```

HelpFile is an optional pathname to a help file. POPHELP automatically loads the BTFILER.HLP file by searching in the current directory, in the directory of POPHELP.EXE, and on the DOS PATH. Specify a different directory or a different help file by putting it on the POPHELP command line. POPHELP applies a default extension of HLP to the specified file name.

The following options can be specified on the command line:

```
/1          Single swap file (for RAM disks or XMS).
/A          Use normal file attribute (rather than hidden) for
hidden swap files.
/B          Force use of black and white video attributes.
/D          Force disk swapping.
/E ProgToExec CommandLine
            Execute a program instead of going resident. No
swapping. This
            option must be the last one on the command line.
/H Height   Specify initial help window height other than 12 rows.
/I HotKey   Specify alternate for help index.
/L HotKey   Specify alternate hotkey for screen lookup.
/M          Disable swap message.
/N          No swapping.
/P HotKey   Specify alternate hotkey for previous topic.
```

/S Path	Specify drive and directory for swap files.
/T	Display help text on second video monitor.
/U	Unload POPHELP.
/V	Leave help text visible on second monitor after popping down.
/X	Use XMS memory for swap.
/?	Display help screen.

Note that '-' can be used instead of '/' when specifying command line options.

When disk space is used for swapping, POPHELP normally uses two swap files. When the /1 option is specified, POPHELP uses a small intermediate buffer that eliminates the need for one of the files and thus cuts its disk space requirement in half. The /1 option has an analogous effect on the XMS swapping option. When /X /1 is specified, POPHELP uses half as much XMS memory, swaps somewhat slower, and consumes 2KB more of normal RAM.

The /D option forces POPHELP to use disk space for swapping even if sufficient EMS or XMS space is available. Specify this option if you have other applications that need the EMS or XMS space, or if you have an extended memory RAM disk that you prefer to use for swapping.

When the /E option is used, POPHELP is resident only for the time the specified program is running. When the program halts, POPHELP removes itself from memory automatically. POPHELP uses COMMAND.COM to run ProgToExec, so the program can be anywhere on the DOS PATH. You can also specify any command line parameters normally used by the program, except redirection of input or output.

Since the help window is resizeable after POPHELP pops up, the /H option is of minor use.

/M disables the swapping message that DESKPOP normally displays when it is swapping to disk. /M is particularly useful when you are swapping to a RAM disk.

/N tells POPHELP to run in non-swapping mode. The program uses more memory this way.

If you have a RAM disk and you want POPHELP to take advantage of it, use the /S option to give POPHELP the drive letter of the RAM disk, and it will swap to and from that area.

The /T and /V options relate to using two monitors with POPHELP, one for the source code and one for the help text. If /T is used, POPHELP displays its output on the monitor that was not active when POPHELP went resident. If /V is used, help text is left visible on the second display after popping down.

The /X option tells POPHELP to use XMS memory for swapping even if EMS is available. If /X is not specified and both EMS and XMS are present, preference is given to EMS.

The following command lines provide common examples of using the options.

```
POPHELP C:\BTREE\BTFILER
```

Load BTFILER.HLP from the specified directory.

```
POPHELP MYHELP /H 15
```

Load MYHELP.HLP using a default help window height of 15 rows.

```
POPHELP /D /M /S F:\ /1
```

Force POPHELP to swap to disk, using the root directory of drive F: for a single swap file. Disable swap messages. The single swap file is most often appropriate on RAM disks, which are very fast but have relatively limited storage space.

```
POPHELP /X /1
```

Enable use of (extended) memory. Note that in order to use XMS you must load an XMS-compatible driver such as HIMEM.SYS, 386MAX.SYS, or QEMM386.SYS. The /1 option causes POPHELP to use half as much extended memory space, with slight degradation in performance.

```
POPHELP /E TURBO MYPROG.PAS
```

Load POPHELP temporarily and run the TURBO integrated environment.

```
POPHELP /N
```

Make POPHELP resident without swapping (keeps lots of DOS memory).

```
POPHELP /U
```

Unload the resident copy of POPHELP.

```
POPHELP /B
```

Force POPHELP to use only black and white (\$07, \$0F, \$70) video attributes.

```
POPHELP /T /V
```

Display help text on the second video adapter (the one that is currently inactive). When POPHELP exits, the help text will remain visible on the second display.

Keep in mind one important restriction when installing POPHELP in swapping mode. When POPHELP pops up, it disables any TSRs loaded chronologically after it. This is necessary because it probably overwrites their memory while it swaps in. Some TSRs, especially network shells and communication programs, won't accept being disabled in this way: the network will disconnect a station that doesn't respond after a time, or a communication program will lose incoming characters. It's possible to avoid these problems by loading POPHELP after such TSRs. To avoid one common problem, POPHELP automatically detects when it is loaded before the Novell NetWare shell and refuses to pop up in this situation.

Hotkeys

POPHELP uses three different hotkeys. By default, the hotkeys are:

<LeftShift><F1>

Look up a help topic based on the word at the cursor

<LeftShift><F2>

Redisplay the previous help topic

<LeftShift><F3>

Display the help index

The following options control the hotkeys:

```
/L    Specify alternate hotkey for screen lookup
/P    Specify alternate hotkey for previous topic
/I    Specify alternate hotkey for help index
```

You can specify alternate hotkeys when POPHELP is installed. On the command line, a hotkey is specified as a hexadecimal word. The top byte specifies the shift key(s) to be pressed and may be zero. The bottom byte specifies the scan code for the hotkey.

The shift key codes are:

```
No shifts - 00
RightShift - 01
LeftShift - 02
Ctrl - 04
Alt - 08
```

Valid scan codes (in hexadecimal) are:

A - 1E	N - 31	0 - 0B	F1 - 3B	[- 1A
B - 30	O - 18	1 - 02	F2 - 3C	; - 27
C - 2E	P - 19	2 - 03	F3 - 3D	, - 33
D - 20	Q - 10	3 - 04	F4 - 3E	/ - 35
E - 12	R - 13	4 - 05	F5 - 3F	\ - 2B
F - 21	S - 1F	5 - 06	F6 - 40] - 1B
G - 22	T - 14	6 - 07	F7 - 41	' - 28
H - 23	U - 16	7 - 08	F8 - 42	. - 34
I - 17	V - 2F	8 - 09	F9 - 43	` - 29
J - 24	W - 11	9 - 0A	F10 - 44	
K - 25	X - 2D		F11 - 57	
L - 26	Y - 15		F12 - 58	
M - 32	Z - 2C			

For example:

```
/L 0244 performs lookup when <LeftShift><F10> is pressed
/P 0819 shows previous topic when <Alt><P> is pressed
/I 0517 displays help index when <Ctrl><RightShift><I> is
pressed
```

Two hotkeys cannot be based on the same scan code, even if the shift keys differ. For example, you cannot use <Alt><F1> for one hotkey and <Ctrl><F1> for another.

Using POPHELP

Once POPHELP is installed you can go about your business. The most likely place for you to start using it is within your text editor or within the TURBO integrated environment.

If you want to use a particular B-Tree Filer routine and you can remember its name but not its parameters, position the editor cursor within or immediately after the procedure name you typed. Then press the Lookup hot key, <LeftShift><F1> by default. POPHELP searches the list of topics looking for an exact match (disregarding case). If that fails, it looks for a topic whose name starts with the search string.

If POPHELP can find a matching topic with one of these two search strategies, it pops up a help window showing you the details about that routine. If it can't find a match, it pops up a window with the names of all the B-Tree Filer units.

If you're not sure of the name of a routine, but you remember which unit it's in, press the Index hotkey, <LeftShift><F3> by default, and you'll get a pick list showing all the B-Tree Filer units. Select the unit you want and a list of topics within that unit appears.

In the help window, you can use the arrow keys to move the cursor to highlighted areas which indicate links to other topics. Once the cursor is positioned in a highlighted region of interest, press <Enter> to bring up that topic. This allows you to browse through the help database. The <Tab> and <ShiftTab> keys make the cursor jump immediately from link to link.

Once you've seen the information you want, you can return to the underlying application by pressing <Esc>. If you want to see the same help topic again, press the Previous Topic hotkey, <LeftShift><F2> by default.

Pasting Help Text into your Editor

Among its other capabilities, POPHELP can also paste marked blocks of help text into your editor. Cut/paste works much the same way that block copying works in the TURBO integrated environment.

The first step is to position your editor's cursor at the exact location where you want to paste the help text. Next, pop up the help screen and select your topic. Move the cursor through the text until it is positioned on the first character that you want to paste. Mark the beginning of the block by pressing <F7> (or <CtrlK>). Scroll through the text again until the cursor is positioned just beyond the last character that you want to paste. Press <F8> (or <CtrlK><K>) to mark the end of the block. POPHELP highlights the block. Finally, press <F4> (or <CtrlK><C>) to copy the marked block into your editor. POPHELP immediately pops down and starts feeding keystrokes to the underlying application. The next time you invoke POPHELP, the block is hidden. If you want to toggle the block's visibility, press <CtrlK><H>.

If the pasted text causes your editor to scroll, the display may temporarily look strange since POPHELP can type much faster than the editor was probably designed to accept. However, your editor will "catch up" when POPHELP finishes pasting characters.

If your editor has an automatic indentation feature, you will probably want to disable it--e.g., <CtrlQ><I> toggles autoindent in the TURBO IDE--during the paste. If autoindent is on during pasting, the pasted text may gradually move to the right of the screen.

If the pasted block is larger than POPHELP's internal paste buffer (512 keystrokes), POPHELP must swap itself into memory to refill the buffer each time it is exhausted. When this occurs the pasting is temporarily delayed until the buffer is refilled and POPHELP swaps out again. Depending on the speed of your system and whether POPHELP swaps to EMS, XMS or disk, this delay can range from a fraction of a second to a few seconds. If POPHELP cannot safely swap itself in when the buffer empties, the paste is aborted prematurely. This should not occur when POPHELP is used over modern programmer's editors; EDLIN may have trouble.

POPHelp Command Summary

The following table summarizes the POPHELP commands.

<F1>

Display the master topic index.

<AltF1>

Display the topic that was displayed just prior to the current one. The cursor is positioned to its last location in the help window.

<F2>, <CtrlQ><F>

Search for a text string. If a non-empty string is entered, POPHELP searches the complete help text for the string. Searching is case-insensitive. POPHELP searches from the position of the cursor until it finds a match or reaches the current topic again. If a match is found, the cursor is moved to the beginning of the match.

<F3>

Prompt for a new help file. A default extension of HLP is added to the name you enter. POPHELP reads any help file compiled using the MAKEHELP utility provided with Async Professional or Object Professional. Note that each help file needs memory space that is proportional to the help file's size and number of topics. POPHELP reserves a certain amount of memory when it goes resident and it cannot increase that amount later. Therefore, if you expect to load multiple help files during one POPHELP session, you should load the largest of them when POPHELP first goes resident.

<F4>, <CtrlK><C>

If a block is marked and visible, paste it from the help system to the underlying editor.

<F5>, <AltZ>

Toggle the zoom mode of the window.

<F6>, <AltM>

Enter move/resize mode. Use the cursor keys to move the window around the screen. This moves the window in small jumps rather than one row or column at a time. Use the Shift-cursor keys to resize the window one row or column at a time. You can also use "speed resizing" to resize the window in small jumps. Use <Ctrl><LeftArrow> and <Ctrl><RightArrow> for horizontal speed resizing; <PageUp> and <PageDown> for vertical speed resizing. The window cannot be resized while it is zoomed.

<F7>, <CtrlK>

Mark the beginning of a block.

<F8>, <CtrlK><K>

Mark the end of a block.

<F9>, <CtrlL>

Perform the last search again. If the last search performed was a topic name search (activated by a <ShiftF1> popup), this command repeats the topic name search. Searching starts with the next topic beyond the current one and is circular, wrapping from the last topic to topic 1 as many times as you like. This feature is especially useful when you're searching for a member function that has a common beginning, such as 'Get'.

If the last search performed was a text search (activated by <F2> or <CtrlQ><F>), this command repeats the text search. Searching starts just after the last match. Previous matches are stored on the topic stack and can be reviewed by pressing <AltF1>. A subsequent <F9>, however, continues from the position of the last match rather than from the current screen.

<CtrlK><H>

Hide or unhide the currently marked block. <F4> pastes only if a block is visible.

<PgUp>, <PgDn>

Move to the next or previous page of the current help topic.

<Left>, <Right>, <Up>, <Down>

Move the cursor around the help screen. Attempting to move the cursor beyond the edge of the window causes more text to scroll into the window.

<Tab>

Jump to the next cross reference topic, if any.

<ShiftTab>

Jump to the previous cross reference topic, if any.

<Enter>

Select the cross-reference topic at the cursor location, if any. The new topic is loaded into the window and the cursor is positioned on the topic's first character.

<Esc>, <AltX>

Exit POPHELP. The current topic and page are retained and can be redisplayed by pressing the Previous Topic hotkey, <LeftShift><F2> by default.

E. Purchase Agreement

This software and accompanying documentation are protected by United States copyright law and also by International Treaty provisions. Any use of this software in violation of copyright law or the terms of this agreement will be prosecuted to the best of our ability.

B-Tree Filer is copyright(c) 1989-1995 by TurboPower Software Company, all rights reserved.

TurboPower Software Company authorizes you to make archival copies of this software for the sole purpose of back-up and protecting your investment from loss. Under no circumstances may you copy this software or documentation for the purposes of distribution to others. Under no conditions may you remove the copyright notices made part of the software or documentation.

You may distribute, without runtime fees or further licenses, your own compiled programs based on any of the source code of B-Tree Filer. You may not distribute any of the B-Tree Filer source code, compiled units, or compiled example programs without written permission from TurboPower Software Company.

Note that the previous restrictions do not prohibit you from distributing your own source code or units that depend upon B-Tree Filer. However, others who receive your source code or units need to purchase their own copies of B-Tree Filer in order to compile the source code or to write programs that use your units.

The supplied software may be used by one person on as many computer systems as that person uses. We expect that group programming projects making use of this software will purchase a copy of the software and documentation for each member of the group. Contact TurboPower Software Company for volume discounts and site licensing agreements.

With respect to the physical diskettes and documentation provided with B-Tree Filer, TurboPower Software Company warrants the same to be free of defects in materials and workmanship for a period of 60 days from the date of receipt. If you notify us of such a defect within the warranty period, TurboPower Software Company will replace the defective diskette(s) or documentation at no cost to you.

TurboPower Software Company warrants that the software will function as described in this documentation for a period of 60 days from receipt. If you encounter a bug or deficiency, we will require a problem report detailed enough to allow us to find and fix the problem. If you properly notify us of such a software problem within the warranty period, TurboPower Software Company will update the defective software at no cost to you.

TurboPower Software Company further warrants that the purchaser will remain fully satisfied with the product for a period of 60 days from receipt. If you are dissatisfied for any reason, and TurboPower Software Company cannot correct the problem, contact the party from whom the software was purchased for a return authorization. If you purchased the product directly from TurboPower Software Company, we will refund the full purchase price of the software (not including shipping costs) upon receipt of the original program diskette(s) and documentation in undamaged condition. TurboPower Software Company honors returns from authorized dealers, but cannot offer refunds directly to anyone who did not purchase a product directly from us.

TURBOPOWER SOFTWARE COMPANY DOES NOT ASSUME ANY LIABILITY FOR THE USE OF B-TREE FILER BEYOND THE ORIGINAL PURCHASE PRICE OF THE SOFTWARE. IN NO EVENT WILL TURBOPOWER SOFTWARE COMPANY BE LIABLE TO YOU FOR ADDITIONAL DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS, OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE THESE PROGRAMS, EVEN IF TURBOPOWER SOFTWARE COMPANY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

By using this software, you agree to the terms of this section. If you do not agree, you should immediately return the entire B-Tree Filer package for a refund.

TurboPower SoftWare Company provides electronic mail and telephone support for B-Tree Filer on an as-available basis at no extra charge. We encourage you to use electronic mail for support questions. When you call for technical support, the person who happens to answer the phone will try to answer your question on the spot. When you ask a question by electronic mail, the person most qualified to answer your question will respond. Please have your serial number handy (it is on the front cover of this manual) when you call. Electronic access information and the technical support telephone number are listed on the title page of this manual.

2. Configuration

Before using B-Tree Filer's units, you need to understand and perhaps modify various settings in the following categories:

- conditional compilation defines
- compiler options
- the network interface

These configuration options primarily control the operation of the FILER unit, which is the core of B-Tree Filer's data management capabilities. This chapter describes these topics in detail.

A. Conditional Compilation Defines

Conditional defines are used to control the operation of B-Tree Filer. These defines are in a Pascal include file, BTDEFINE.INC, that is included into all B-Tree Filer units. To modify it, load it in an editor and find the appropriate conditional symbol. TurboPower uses the convention of inserting a period between the '{' and the '\$' of a conditional define to deactivate it. For example, the following define is active

```
{ $DEFINE NoNet }
```

The deactivated form of the same define is

```
{ . $DEFINE NoNet }
```

With the period in place, the compiler sees the line as a comment which has no effect on compilation.

After modifying BTDEFINE.INC, save it to disk and use the compiler or the supplied make file to rebuild your application for the changes to take effect.

Network Definition

If you want to access files on a network file server, you must reconfigure the FILER unit.

Conditional defines in BTDEFINE.INC specify which networks are supported and you must activate the desired defines for proper network support. The default settings in BTDEFINE.INC are:

```
{ $DEFINE NoNet }  
{ . $DEFINE Novell }  
{ . $DEFINE MsNet }
```

Since NoNet is defined by default, no network support code is compiled into the FILER unit. To activate one or more network interfaces, insert a period in front of the NoNet \$DEFINE, and remove the period from the networks you want to support. Be sure that the compiler rebuilds FILER.TPU so that the changes take effect. Once one or more network interfaces are available, the BTInitIsam function is used to select which one to use for a given program run.

Selecting both the MsNet and Novell defines links in only an extra 2KB of code, which is a small penalty to pay for allowing the run time selection of the network. You can use a run time configuration or INI file or a command line option to optimize the behavior of your network application for the network on which it is running.

The following provides more details about each of the network options.

NoNet;

Locking and file sharing routines do nothing when FILER is compiled with this define. Be very careful not to run a NoNet application in a multi-user setting. Many networks will let you proceed, but the data and index files are likely to become corrupted if two or more workstations run the same application.

Novell;

Supports all networks compatible with Novell NetWare 2.x, 3.x or 4.x. Uses Novell-specific record locking APIs for optimized performance.

MsNet;

Supports any network compatible with Microsoft's locking calls, which were first implemented in the SHARE program distributed with MS-DOS 3.1. The only basic requirement is that the network correctly implement DOS file sharing when a file is opened (using DOS INT \$21 function \$3D) and DOS file locking (using DOS INT \$21 function \$5C). The use of this interface usually requires that the MS-DOS SHARE or equivalent be loaded and configured for the network. When running under Windows either SHARE must be loaded prior to running Windows, or the VSHARE driver must be loaded in Windows.

Most networks (including Novell) offer support for Microsoft record locking, so MsNet is a good value to try if you're not sure which define is correct for your situation. Here is a partial list of the networks and operating systems supported by the MsNet network interface:

- Microsoft Windows (true Windows applications and DOS programs run in a DOS box)
- Microsoft Windows for Workgroups
- Microsoft LAN Manager
- IBM PC-LAN
- NetWare Lite and Personal NetWare
- Artisoft LANtastic
- CBIS Network-OS version 6.20 or later
- PC_MOS/386 version 2.01 or later (The Software Link, Inc)
- Vines version 3.01(0) or later (Banyan Systems, Inc)
- Alloy NTNX
- DESQview (Quarterdeck Office Systems)

Despite their names (and indeed the manufacturer's name), NetWare Lite and Personal NetWare are not NetWare compatible; they are instead peer-to-peer networks and do not contain NetWare 2.x, 3.x or 4.x server compatible software. You must use the MsNet network interface for applications intended to run on these networks.

Using EMS in Real Mode

Two defines in BTDEFINE.INC control whether the FILER unit can use EMS memory for storing index page buffers in real mode. Index buffers consume a minimum of 20KB of heap space using the default constants.

Larger buffers can significantly improve the performance of index-intensive operations, especially for single-user and lightly loaded multi-user applications. By placing index buffers in EMS memory instead of on the normal heap, you can reduce normal heap usage to a negligible amount and, in many cases, arrange to load the entire index into EMS RAM for the ultimate in performance.

To enable EMS support in Filer, activate the UseEMSHeap; define in BTDEFINE.INC (it will only take effect if you are compiling for real mode). This define is activated by default and pulls in about 5.5KB of code from the EMSHEAP unit. As described in _6.A, the EMSHEAP unit automatically detects EMS and initializes itself accordingly. The BTInitIsam procedure described in Chapter 5 is used to specify the amount of EMS to use, if any, for a particular run of a program.

The EMSDisturbance; symbol, also in BTDEFINE.INC, can be defined when UseEMSHeap is defined. EMSDisturbance, which is off by default, controls whether the FILER unit restores its own EMS page frame context before it accesses the EMS page frame (see _6.A for more information). This step is needed only if another portion of the application also uses EMS and modifies the page frame without saving and restoring its state. Note that Borland's OVERLAY unit, which is probably the most likely additional user of EMS, does save the page frame and therefore does not require EMSDisturbance to be defined. On the other hand, the EMS large arrays in Turbo Professional and

Object Professional assume that they "own the page frame" and modify it without regard for any other code. Hence, for these units to coexist with the FILER unit, EMSDisturbance must be defined. Note that FILER always saves and restores the page frame context for the benefit of other users of EMS, regardless of the setting of EMSDisturbance.

There are several other compiler defines that control the EMSHEAP unit, and therefore they indirectly affect FILER if UseEMSHeap is defined. These defines, UseTPEMS, UseOPEMS, DebugEMSHeap, NoErrorCheckEMSHeap, and ManualInitEMSHeap, are described in _6.A.

Index File Key Strings;

All keys in a B-Tree Filer index are stored as strings. The FILER unit can store either of two kinds of strings. If the LengthByteKeys; define in BTDEFINE.INC is active (as it is by default), Turbo Pascal style strings are used. Each string is composed of a length byte followed by the characters that comprise the string. If the ASCIIZeroKeys; define is active, C style strings are used. The characters of the string come first and are terminated by a single null character (#0). Only one of the two defines can be active at one time.

ASCIIZeroKeys controls only the format of the strings stored in the index file. All strings passed to and returned from FILER routines continue to use the Pascal format. The purpose of this define is to create an index file that can also be read and written by B-Tree Filer for C.

If you change these defines, you must rebuild the index file of all affected fileblocks.

Using B-Tree Filer in a DLL

Starting with version 5.50, many B-Tree Filer functions can be accessed via a DLL (DynaLink Library). The DLL was coded so that getting your application to use the DLL is just a matter of changing a compiler define in BTDEFINE.INC and recompiling. The syntax of the B-Tree Filer calls did not change. To use the DLL, your application must be compiled for protected mode or Windows, and hence it is not available for Turbo Pascal 6.0 or 7.0, or Borland Pascal 7.0 in real mode. For information about DLLs, see the Borland documentation.

No source code is provided for the DLL with this Pascal version of B-Tree Filer. The reason is simple: it is written in C and compiled with B-Tree Filer for C (and comprises most of that product). The source code to the DLL and MAKE files to recreate it are provided with B-Tree Filer for C.

To use the B-Tree Filer DLL at run-time instead of linking the code statically into your application, activate the UseFilerDLL compiler define in BTDEFINE.INC:

```
{.$DEFINE UseFilerDLL}
```

By default the define is not activated. Remove the period to activate it and then rebuild the B-Tree Filer units for a protected mode or Windows target. You should use the supplied MAKE file to do this (see _1.B for information on rebuilding the units).

Recreating the B-Tree Filer units with UseFilerDLL; active replaces the code in the following units with calls to exported routines in the DLL.

- FILER - basic B-Tree Filer unit
- VREC - variable length record support
- ISAMTOOL - extra tools
- NUMKEYS - number-to-key conversions

- DBIMPEXP - dBase file import/export
- REORG/VREORG - fileblock reorganization
- REBUILD/VREBUILD - rebuilding fileblocks
- FIXTOVAR - fixed length to variable length fileblock conversion

When you distribute your application, you must also distribute the B-Tree Filer DLL. The file name of the DLL is CBTLWDS.DLL (which stands for C B-Tree, Large memory model, Windows target, Dynamic linking, Single threaded). If your application runs under Windows, the DLL must be in the current directory, in the same directory as the application, in the Windows directory (usually \WINDOWS), in the Windows system directory (usually \WINDOWS\SYSTEM), or on the DOS PATH. If your application is a protected mode application, the DLL must be in the current directory, in the same directory as the application, or on the DOS PATH.

There are a few points to be aware of when using the B-Tree Filer DLL:

- The DLL can only be used by a protected mode program or a Windows program.
- Because the DLL is likely to be used by a number of different applications with differing requirements, it is compiled to provide functionality to a common denominator. In particular you should note that the DLL is compiled with both the MsNet and Novell network support included, to support both ASCIIZ and Pascal-style strings (both in the index file and for the application), and with full numeric coprocessor support (in Pascal this would be equivalent to the \$N+ and \$E+ compiler defines).
- The maximum length of a key string is limited to 127 characters. You cannot access fileblocks that have an index with a larger key length. There are two reasons for this. First, the B-Tree Filer DLL was compiled with this setting for MaxKeyLen (see Chapter 5 for the definition of MaxKeyLen). Second, this value was chosen as a trade off between maximizing the key length and minimizing the memory consumed by the index page buffers.

Protected Mode and WIN87EM.DLL.

There is a small problem that may affect you if you use the B-Tree Filer DLL in protected mode. The DLL was compiled with the C/C++ equivalent of Pascal's \$N+ and \$E+ compiler defines. Under Windows, the numeric coprocessor is emulated by the Windows DynaLink Library WIN87EM.DLL. Although the B-Tree Filer DLL has no floating point code in it (it just requires the type definitions of single, double, extended and so on), the C run-time library makes a specific initialization call to WIN87EM.DLL. This causes the problem: the Windows WIN87EM.DLL is not callable in protected mode, and RTM.EXE (the program provided with Borland Pascal that provides some of the Windows operating system calls in protected mode) causes your application to fail with an error message. Later versions of RTM.EXE trap this initialization call to WIN87EM and ignore it.

The best solution is to download the latest version of RTM.EXE from the BPASCAL forum on CompuServe (in fact you'll find the latest DPMI16BI.OVL there as well). If you cannot, then we provide a replacement WIN87EM.DLL for use in protected mode. This DLL exports just one entry point, the initialization routine that the C/C++ run-time library attempts to call. However, its use does come with a strong warning. You must *not* put this replacement WIN87EM.DLL anywhere where Windows might find it. It could get loaded instead of the real coprocessor emulator, causing Windows and Windows programs to behave erratically. You must ensure that it is in the same directory as your application, and not in the Windows system directory.

Other Considerations for the DLL

The number of compiler defines in BTDEFINE.INC are reduced when you select to use the DLL. Because the DLL already has the Novell and MsNet network interfaces compiled into it, the network selection in BTDEFINE.INC is much simpler. You can either include network support or not:

```
{ $DEFINE NoNet }
```

is on by default. Deactivate this compiler define if you want to use B-Tree Filer under a network.

UseEMSHeap (see "Using EMS" earlier in this section) is not used when you select to use the DLL. This define is only for real mode and since the DLL can only be used by a protected mode or Windows program, UseEMSHeap no longer has a meaning.

The fileblock reorganization routines (those found in the REORG, VREORG, REBUILD, VREBUILD, and FIXTOVAR) do not use fast record buffering schemes when using the DLL. If you would prefer speed over final EXE file size, use the RESTRUCT and REINDEX routines which are statically linked to your program (see _6.C for more information).

You cannot modify the network interface if you are using the DLL. _2.C does not apply in this case.

Initialization Blocks

InitAllUnits;, which is not defined by default, controls whether each B-Tree Filer unit has an initialization block, even if only an empty one. Defining this symbol works around a bug in some very early versions of Borland's Turbo Debugger. Defining it also generates 16 bytes of extra code for units that wouldn't otherwise have an initialization block, and generates a tiny delay at program startup for calls to the empty initialization blocks. If you run Turbo Debugger and the source code for units that you compiled with debug information does not appear in the module window, you may need to activate this define.

CRT and Mouse Functions

UseTPCRT; and UseOPCRT; determine whether the BROWSER unit takes advantage of the CRT and mouse functions of the units from Turbo Professional and Object Professional. If neither define is active, BROWSER uses Borland's unit and mouse operations are not available in the browser. If your program already uses TPCRT and/or TPMOUSE, define UseTPCRT; if it already uses OPCRT and/or OPMOUSE, define UseOPCRT. Failure to do so will lead to a "CRT/TPCRT conflict" or at least the presence of wasted code in the application's EXE file. If you enable UseTPCRT or UseOPCRT, you'll need to have a copy of TPDEFINE.INC or OPDEFINE.INC available if you compile BROWSER.PAS, SIMPDemo.PAS, or NETDemo.PAS. These include files are provided with Turbo Professional and Object Professional, respectively.

Abnormal Locking Behavior

LockBeforeRead;, which is not defined by default, offers an automatic workaround for a bug that appears in some versions of Novell's NETX shell. Under these versions, if one station places a record lock and a second workstation attempts to read the locked record, the second station will report an erroneous B-Tree Filer 10070 error. The reason for this is that NETX is attempting to signal a DOS error code 5, but is forgetting to set the carry flag (which DOS uses to signal an error to an application). B-Tree Filer takes this to mean that the read request succeeded but that only 5 bytes were read. Generally more bytes were requested, and hence B-Tree Filer takes this to mean a disk problem and hence generates error 10070.

To avoid this problem, LockBeforeRead causes Filer to place a lock on any region of a file it is about to read. If another station already has a lock within that region, the lock attempt fails immediately. If it succeeds, the station performs the read and then automatically removes the lock (if the lock wasn't placed independently for another reason). This behavior reduces performance but always avoids the associated problem.

An alternative to defining LockBeforeRead is to upgrade users to a version of the NetWare shell without the problem. NETX versions 3.22 through 3.26 do not seem to have the problem. Versions 3.31 and 3.32 (shipped for MS-DOS 6.0 compatibility) have the problem again. The special "patched" version 3.32PTF does not exhibit the problem. The VLM Requester does not have the problem.

Older Versions of B-Tree Filer

The symbol BTree52; is always defined in BTDEFINE.INC to identify the FILER unit naming conventions new to versions 5.21 and later. This define allows units that extend B-Tree Filer to be written to work with both the old and new calling sequences.

Turbo Pascal Compiler Versions

You should never modify the Heap6; define in BTDEFINE.INC, but you should understand its purpose. Heap6 is automatically defined when BTDEFINE.INC is compiled using the Turbo Pascal 6 or 7 compilers (and possibly will be for future compiler versions as well). Because Borland rewrote the compiler's heap manager for version 6, it's necessary to adjust various internal functions of B-Tree Filer to work with the differing heap managers. Alternate code is controlled via the Heap6 define.

Because B-Tree Filer no longer supports Turbo Pascal 4.0, a test in BTDEFINE.INC (known as a compile stopper) halts compilation if this version of the compiler is detected. Please note that officially B-Tree Filer no longer supports Turbo Pascal versions 5.0 or 5.5, although they aren't specifically excluded with a compile stopper.

B. Compiler Options

Each unit provided with B-Tree Filer begins with source code similar to the following:

```
{ $I BTDEFINE.INC }
{ $F-,V-,B-,S-,I-,R-,A- }
```

BTDEFINE.INC is included to allow specification of compilation directives. Note that you can modify BTDEFINE.INC to specify compiler options as well as just the conditional symbols described already. Finally, the unit source code continues with a group of directives that must be used for the unit; these settings override any that you put in BTDEFINE.INC.

In real mode, the majority of B-Tree Filer units can be overlaid given the default compiler directives in their source files. The exceptions to this rule are:

```
NWIPXSPX
NETBIOS
```

The following overlayable units have initialization blocks. Overlaying such a unit requires that Turbo Pascal's overlay manager be initialized before the initialization block is executed. See the Turbo Pascal manual for more information.

```
Basic B-Tree Filer:  BASESUPP, EMSHEAP, FILER, VREC
Browsers:           BROWSER, OPBROW
Network tools:      NWBASE
Other tools:        SHARE, DBIMPEXP, MSORT
```

Although almost any B-Tree Filer unit can be overlaid, performance considerations will often make it undesirable to do so.

When using library packages such as B-Tree Filer, remember that the Turbo Pascal compiler has a finite capacity. When you write a short program that uses units provided by others, you may forget that the compiler is actually processing lots of code provided by the units listed in the uses statement. To maximize the compiler's capacity, you should apply the following advice. (Because Borland Pascal has a much higher capacity, the following does not apply and you can usually turn debug information on everywhere.)

1. In the TURBO.EXE integrated environment, apply the following settings:

Compile Destination	Disk
Debug Information	Off
Local Symbols	Off
Link Buffer	Disk
Integrated Debugger debug info	Off
Standalone Debugger debug info	Off

Although these settings may lead you to believe that you can't use a debugger, that's not true. The important goal is to add debug information only to those units where you need to trace, not in every unit.

2. If you outgrow the capacity of the integrated environment, use the command line compiler TPC.EXE instead. When using TPC, make the following options part of your TPC.CFG file (see the Turbo Pascal manual) or part of your command line:

```
/$D- /$L- /L /M
```

These options turn off debug information, force link buffering to disk, and perform a make by default. Again, the key to compiling with debug information is to add such information to only those units where you need it and to strip the information after you're finished with it. To compile a single unit with debug information, use the following command line:

```
TPC /$D+ /$L+ /L UnitName
```

To compile a program with debug information for Turbo Debugger, use the following command line:

```
TPC /$D+ /$L+ /L /V ProgName
```

Do not use the /V option regularly, since it causes the compiler to append debug information to your program's EXE file, which will bloat it significantly.

3. If you outgrow even the capacity of TPC.EXE, consider using the TPCX.EXE protected mode compiler of Turbo Pascal 6.0 or BPC.EXE of Borland Pascal 7.0. These compilers use up to 15MB of extended memory to provide almost unlimited capacity.

There are various other tricks for improving compile capacity, although in most cases they offer second-order improvements compared to the items already listed. Keep the following in mind:

- Unload as many TSRs and device drivers as possible.
- Turn off range and stack checking with {\$R-,S-}.
- Extract all units from TURBO.TPL by using TPUMOVER, and instruct the IDE not to look for TURBO.TPL at all. In this case, the compiler will load other units such as SYSTEM.TPU and DOS.TPU only when the program calls for them.
- Keep the main program as small as possible and instead put all code into units. After following this advice, your main program will use a single main unit and contain a single call to the one "Main" routine interfaced by that unit.
- Minimize the number of identifiers interfaced by every unit.
- Carefully set the defines in files such as BTDEFINE.INC to activate only the library features that you need.

C. Customizing the Network Interface

This section provides more internal details about how B-Tree Filer supports networks. It also describes the steps you need to take if you are writing for a network that is not supported by the built-in interfaces of B-Tree Filer. It is unlikely that you'll need to do this, since almost every current PC-based network supports the DOS function calls used by the MsNet interface.

Note that if you are using B-Tree Filer in a DLL, you cannot modify the network interface.

To work with B-Tree Filer, the network must support the extended file sharing attributes supported by MS-DOS since version 3.0. The network interface does not allow easy customization of the "file open" function. The network must also provide a convenient mechanism for multiple stations to refer to disk files shared among them. This is typically done by drive mapping letters, or by server naming conventions.

The network must also offer a function that provides for physically locking and unlocking regions of a file, given its handle. This function must support "virtual locks," i.e., locks that are located at offsets beyond the current size of the file. B-Tree Filer uses such locks to determine an individual "number" when it opens a given fileblock. It places single byte locks at file offsets starting just below \$7FFFFFFF of the fileblock's dialog file. The locks are used to reserve a workstation number for each station opening a fileblock. B-Tree Filer then refers to this unique workstation number to regulate control of file modification and index buffer flushing for that fileblock. The internal workstation number can be obtained by calling the BTGetInternalDialogID function of the FILER unit.

When a station attempts to read a section of a file that has been locked by another station, the network must immediately fail and return an error code either to the DOS file read call, or via a critical error (INT \$24) interrupt handler. Similarly, if a station attempts to lock a region of a file that is already locked by another station, the locking call must fail immediately or be entered into a server-maintained queue of requested locks that will timeout after a specified number of milliseconds.

If you have a network that meets all of these requirements but does not function correctly with one of the predefined network interfaces, follow these instructions for adding a new interface.

All network interface code is located in the source file ISNETSUP.INC (located in the \FILER directory). The available interfaces are determined at compile time by compiler defines found in BTDEFINE.INC. The selected interface is determined at run time by a parameter passed to BTInitIsam.

To add a new network interface, choose an unused network name, for example MyNet. Insert your network routines before the double line above the function IsamInitNet in ISNETSUP.INC. Surround your routines with {\$IFDEF MyNet} and {\$ENDIF}. See the existing network interfaces for examples.

Required Network Interface Functions

Each network interface must contain the following functions (MyNet stands for the network name):

```
function MyNetLockRecord(Start      : LongInt;  
                        Len         : LongInt;  
                        Handle      : Word;
```



```

        TimeOut      : Word;
        DelayTime : Word) : Boolean; far;

```

This function must lock the bytes from Start to Start+Len-1 inclusive in the file designated by Handle. If successful, the function should return True, otherwise False. TimeOut is the time period (in milliseconds) that the lock should be retried if the initial attempt fails. If TimeOut is exceeded before a successful lock, the function must return False. DelayTime is the pause time, in milliseconds, between successive locking attempts. This pause helps to prevent deadlocks when multiple workstations are trying to lock the same file region. When possible, TimeOut should be passed on to a network specific function that queues lock requests on the file server. In this way, the server can service the requests in the order in which they were received rather than based on random requests from various stations. Notice that the function *must* be declared far, either with the far keyword, or by ensuring that {F+} define is set.

```

function MyNetUnLockRecord(Start      : LongInt;
                           Len        : LongInt;
                           Handle     : Word) : Boolean; far;

```

This function must unlock the bytes from Start to Start+Len-1 inclusive in the file specified by Handle. If successful, the function must return True, otherwise False. Notice that the function *must* be declared far, either with the far keyword, or by ensuring that {F+} define is set.

```

function MyNetInitNet : Boolean;

```

This function must assign a value to IsamNrOfWS (declared in FILER.INC). This number specifies an upper limit on the actual number of workstations that can access B-Tree Filer fileblocks simultaneously. IsamNrOfWS must have the same value on all machines in the network and must be in the range of 1 to MaxNrOfWorkstations (50 by default). If the network does not provide a function that indicates the maximum number of workstations supported, it's always safe to assign MaxNrOfWorkstations to IsamNrOfWS.

This function must also perform any network-specific initialization, such as installing an int \$24 handler to detect sharing violations. If it does need to install an int \$24 handler, it can install B-Tree Filer's standard one by calling IsamInstallInt24Handler; (a procedure with no parameters).

```

function MyNetExitNet : Boolean; far;

```

This function must reverse all actions performed by MyNetInitNet. This is especially important for removing an interrupt \$24 service routine. If successful, the function must return True, otherwise False. The standard \$24 handler can be removed by calling IsamRemoveInt24Handler; (another procedure with no parameters). Notice that the function *must* be declared far, either with the far keyword, or by ensuring that {F+} define is set.

In addition, three internal function pointers (defined in FILER.INC) must be initialized:

```

IsamLockRecord
IsamUnLockRecord
IsamExitNet

```

The MyNet functions must be assigned to the function pointers. The assignments are:

```

IsamLockRecord = MyNetLockRecord;
IsamUnLockRecord = MyNetUnLockRecord;
IsamExitNet = MyNetExitNet;

```

Binding the Network Interface

To bind your new network interface correctly, you must perform a few more steps. At the end of ISNETSUP.INC, modify the function IsamInitNet. At the end of the case statement (just before the else clause), add the following new label:

```
{ $IFDEF MyNet }  
  MyNet :  
    IsamInitNet := MyNetInitNet;  
{ $ENDIF }
```

Insert the following lines at the beginning of BTDEFINE.INC (directly with the Novell and MsNet compiler defines):

```
{ . $DEFINE MyNet }
```

Removing the period between the brace and the \$DEFINE enables your network interface.

The enumerated type NetSupportType is defined in the file FILER.PAS. Expand it as follows:

```
NetSupportType = (NoNet, Novell, MsNet, MyNet);
```

Finally, you must expand the initialization code of IsamCompiledNets in FILER.PAS as follows:

```
{ $IFDEF MyNet }  
  + [MyNet]  
{ $ENDIF }
```

3. B-tree Principles

This chapter offers some theoretical underpinning for B-Tree Filer, defining the terminology and describing how the fundamental B-tree operations work. If you've already had a course covering B-trees, or if you're not really interested in the internals, you may want to skip directly to Chapter 4.

Software developers use the term "" so often that its meaning in any particular context is often hazy. In this manual, the term database is used to refer loosely to a collection of data stored for an application, without really specifying what form that collection takes. The term "" means the combination of a data file, an index file containing keys, and other information used to manage access to these files. With this terminology in mind, we can proceed with the general topic of data and keys.

A. Data and Keys

There are two basic methods of searching for particular information in a database:

- data access
- indexed sequential data access, also known as random access

Searching by sequential access entails reading the data file from beginning to end while comparing every data record to the desired one. This is slow and inefficient with large data sets. To access the data directly, you must know the position of the desired data record in the sequential data file. Given the position, one data access will read the correct record.

The information needed to find the data record is available with the help of the index key information stored by B-Tree Filer. Data records and keys are stored in two independent files, the data file and the index file.

The Database: data file consists of fixed size data records stored one after another. Every data record corresponds to a number, the data record reference, defining the position of the data record in the data file. For example, reference 0 refers to the very first record in the data file (reserved by B-Tree Filer for internal use), and reference 1 is the first user record in the file. When a record is entered into the data file with the B-Tree Filer routine `BTAddRec`, the data record reference is returned, and can then be added to the index file with its corresponding key. The file contains the keys and their corresponding record reference values. (The terms "data record reference" and "number" are used interchangeably in this manual.)

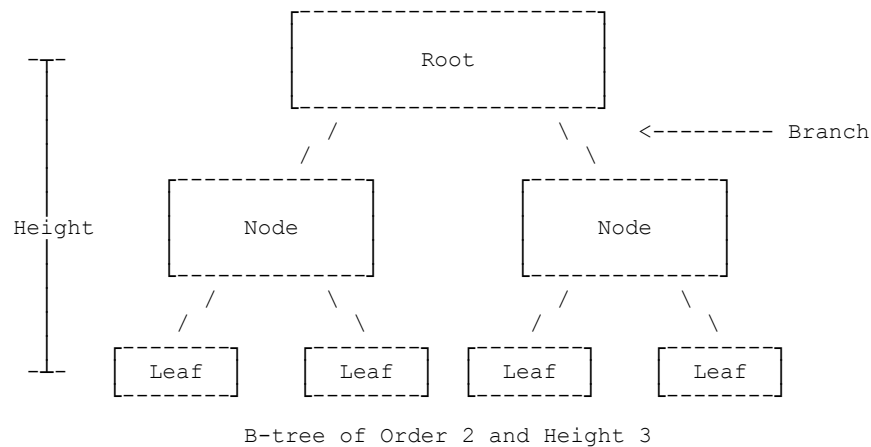
To find a certain data record, you must search in the index file for the key by using, for example, `BTFindKey`. If the key is found, the data record can be read by `BTGetRec`, using the reference number returned by the search procedure. If the key is not found, the B-tree search method guarantees that the data record doesn't exist.

B. B-tree Organization

Trees are one of the most-studied data structures in computer science, and the B-tree is among the most popular trees. Many people think that "B-tree" means binary tree, but that's not so. The "B" in B-tree stands for "Bayer", who is one of the inventors of this data structure. The B-tree is a much richer data structure than the binary tree, and one that was designed specifically to optimize searching for records in large databases. The remainder of this section defines the terminology of B-trees in general, then describes how B-trees are applied to database management. The following section covers the basic B-tree algorithms for searching the tree, and for adding and removing elements.

B-tree Terminology

The term "tree structure" comes from the similarity of this general data structure to an upside-down tree. A "node" in a B-tree represents a collection of one or more keys. The "branches" connecting the nodes denote a search order to be used while finding a particular key value. The branches of the tree point downward, starting at the "root," continuing until nodes with no outgoing branches are encountered. These nodes are called "leaves."



The height of the tree is defined by the maximum number of branches traversed while going from the root to any leaf. The root is at level 1, its direct branches at level 2, with its direct descendants at level 3, etc. All leaf nodes, i.e., all nodes that don't have any descendants, end at the same level in the B-tree structure.

The number of branches leaving any given node is called the "degree" of the node, with the largest degree giving the degree of the tree. For example, a binary tree has a degree of two, since each node has two branches. Multi-branched trees, e.g., the B-tree, have a degree greater than two. In B-Tree Filer, the degree of the tree is set implicitly with the constant `CreatePageSize` to the value `CreatePageSize+1`.

The "order" of a tree is defined as the minimum number of branches on any node. Every node in a B-tree, not including the root, must be at least half filled; therefore every node, with the exception of the root, must have at least $\text{CreatePageSize}/2+1$ branches. This leads to efficient memory usage.

A "balanced" tree is one in which the levels of the leaf nodes vary by at most one. The B-tree optimizes this even further by requiring all leaf nodes to be at the same level. This implies that no matter which path from the root is taken, the number of steps to the leaf node is always the same.

The construction of a B-tree, in contrast to a binary tree, uses more memory but guarantees fewer accesses to the disk and therefore faster data access times.

B-trees for s

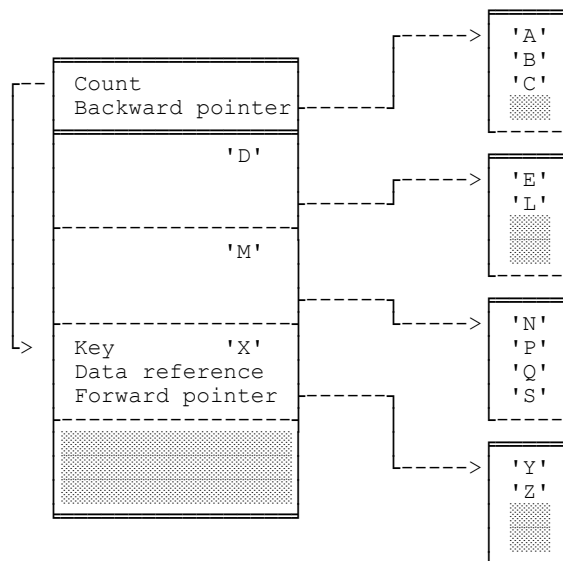
The B-tree is organized as a "tree," i.e., the keys are stored in a certain order in the tree. The keys used to access the records in the data file are stored in the nodes of the tree.

The index entries are stored in increasing order within each node. The first index entry contains the smallest key of the node; the last index entry contains the largest key of the node.

The entries are defined as records containing the following fields:

- the associated key
- the data record reference
- forward node reference

The node reference (forward pointer) points to the node containing keys larger than the one stored in the current index entry.



Internal Structure of a B-tree

The node is itself declared as a record and contains:

- an integer holding the actual number of index entries stored on this node
- a backward node reference (backward pointer) pointing to the node that contains keys smaller than any keys in this node
- an array of the index entries themselves

The B-tree can be traversed from node to node through the use of the backwards pointer, which is used to find smaller keys, and the forward pointer, used to find larger keys.

The smallest key of the B-tree is the first element within the leaf node at the leftmost edge of the tree (which has no backwards pointer, since there are no keys smaller), and the largest key is the last element within the rightmost leaf node (with no forward pointer, since there are no larger keys). The search path (starting from the root) to find either the smallest or largest key is of equal length since all leaf nodes of a B-tree must lie at the same level.

Every node contains at least $\text{CreatePageSize}/2$ and at most CreatePageSize index entries. The root is the sole exception in that it may contain less than $\text{CreatePageSize}/2$ index entries.

Choosing a value for CreatePageSize involves tradeoffs. The search for keys in larger pages requires fewer accesses to the disk, and is therefore faster than having smaller pages. The search within a page is considerably faster than loading a page from disk. However, a larger CreatePageSize increases the memory requirements of B-Tree Filer, and it also increases the time to read a page. The default value of 62 for CreatePageSize was selected based on thorough experimentation, and guarantees optimal access times and appropriate memory usage.

B-Tree Filer also defines the maximum height of the B-tree with the constant MaxHeight . With the default value of 8, the B-tree will not fill until more than 10^{10} keys are added to it. Since 2^{32} keys is less than 10^{10} , there is no danger of the B-tree overflowing!

Keys in the Index File

As already described, the keys in the file provide the connection to the data records in the data file. Every time a data record is entered, at least one corresponding key must be added so that the data record can be found through an indexed access.

Since the index and data files are stored separately, either file can be manipulated independently. If a key is deleted, the corresponding data record is not automatically deleted. In this case, the data record cannot be found by an indexed access. An indexed access is also not possible if a data record is entered in the data file without simultaneously adding a key entry to the index file. Parallel manipulation of the index and data files is therefore essential for proper functioning of B-Tree Filer.

All keys are of type `IsamKeyStr` (a String type). If a number, e.g., an invoice number, is to be used as a key, it must first be converted to a string. (Routines to perform such conversions are in the `NUMKEYS` unit, discussed in 6.G.) The value of each element in the string is ASCII based. Care must be used when building the string (e.g., use of upper and lower case letters) to insure that keys can be sensibly compared with one another.

There are two types of keys, distinguished as follows:

- primary keys: keys that are unique within the database
- secondary keys: keys that can be entered multiple times

Primary keys cannot be duplicated within the index file (this requirement is enforced by B-Tree Filer's indexing routines). Customer numbers, invoice numbers, etc., can be used as primary keys.

Secondary keys are used for non-unique items: names, addresses, weights, etc. B-Tree Filer keeps these possibly duplicated secondary keys separate internally by combining the unique data record references with each key. (The process of combination is done logically, and requires no additional space for key storage.)

The B-tree does not require a primary index since internally all keys are made unique anyway.

The maximum length of a key is determined by MaxKeyLen, which can be between 1 and 255 (default value is 30). The length and type of each key are determined by the values of the fields KeyL and AllowDupK, specified within a variable of type IsamIndDescr.

Every key corresponds to exactly one data record, though many keys can point to a data record. In an address data base, the name, the address, as well as any other item can be made into a key. The number of keys that can point to a data record is determined by the parameter NumberOfKeys that is passed to the procedure BTCreateFileBlock. In B-Tree Filer, no matter how many keys are used to manage a given database, all those keys are stored in a single index file.

C. Managing Keys

Adding or deleting a single key can require a major reorganization of the B-tree in order for it to retain its required properties. This section describes how B-Tree Filer transparently manages the B-tree.

Searching for a Key

The search for a key in the B-tree always begins at the root node. The root and all other nodes are searched using a binary search. The number of index entries currently stored on the node is halved and the middle index entry obtained. Its key entry is then compared to the desired key. If they are identical, the search is ended successfully.

The search must proceed further if the key entry does not match the desired key. Since the keys on a node are always stored in increasing order, the comparison of the present key with the desired key provides the direction for further search. If the compared key entry is smaller than the desired key, the search proceeds to the right. If it is larger than the desired key, the search goes left. If the key is found anywhere on the present node, the search is ended.

Otherwise, the backwards and forwards pointers can be used to locate another node. The forward pointer of the last index entry whose key was smaller than the desired key leads to a new node with keys larger than the last entry and smaller than the next entry of the original node. The search then continues on the new node, which may already be in memory or may require a disk access to obtain. Only when the search proceeds to a node where the smallest index entry is larger than the search key is the backwards pointer used. The search then continues on this node, where the keys are all smaller than the one just searched. The search ends successfully when the corresponding key is found or in failure when a leaf node of the tree is reached without a match.

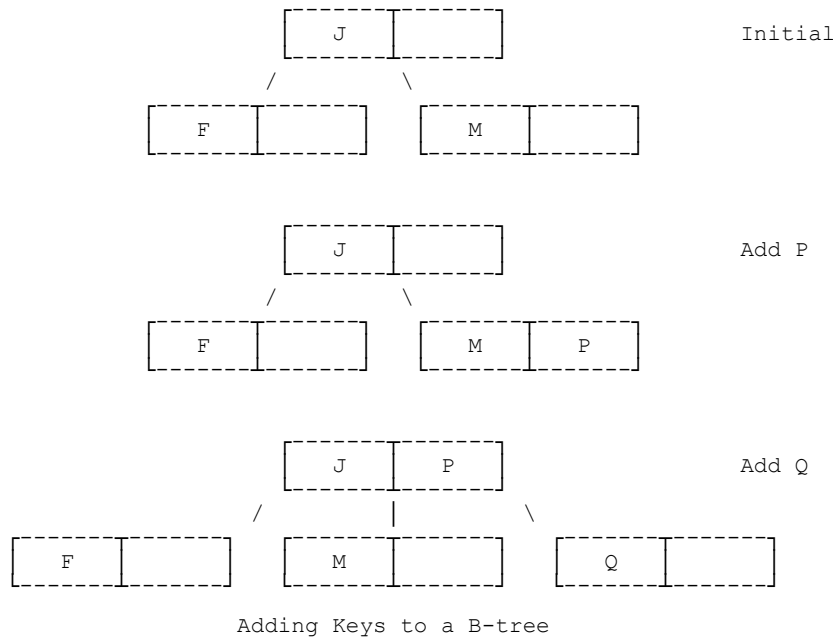
The B-tree can also access the records of a database in sorted sequential order. From a given starting point, the forward pointer of the current index entry leads to the next key in sorted order. When a leaf node is reached, the algorithm backs up to the previous node level and continues with the next larger index entry. This is equivalent to a depth-first traversal of the B-tree. To denote a particular location within the B-tree, the term "pointer" is used. In reality the sequential pointer is an array of records, each record containing one number that defines the node and another that indicates the active element within the node. The sequential pointer can be thought of as a path leading from the root to the node containing the current entry. Certain B-Tree Filer routines initialize the sequential pointer to a known value. Other routines overwrite the pointer with a temporary value. Descriptions of the B-Tree Filer index functions point out their effect on the sequential pointer.

Adding a Key

If a new key is to be added to the index file, the B-tree is first searched to assure that the key doesn't already exist. If the key does exist, an error is generated. (Remember that duplicate keys are made unique by incorporating their data record reference numbers.) If the key does not exist, the search ends at a leaf node with negative results. The new key is added at this location. If there is still room for another key on that leaf node, the addition is performed by adding the new key in sorted position within the node's index array. However, if the leaf node is full--there are already CreatePageSize keys stored there--the B-tree must be modified to make room for the new key.

The required space is created by splitting this node in two. The CreatePageSize/2 smaller index entries remain on the old node, while the CreatePageSize/2 larger index entries are entered on a

new node. The middle index entry, which is smaller than the index entries of the new node, is used to point to this new node, by entering it in a node of a previous level of the tree.



If the node that this middle index entry is to be entered into is also full, another node split occurs. By recursively applying this method there may be node "halvings" all the way to the root level. Even the root can then be split as described. In this case, the middle index entry from the final split becomes the new root and the tree grows by one level. This is the only case where the height of the tree grows. This method of adding keys preserves the distinctive characteristic of a B-tree, that all leaf nodes must lie at the same level.

In implementing splitting, B-Tree Filer offers a further enhancement of basic B-tree theory. Before a node is split, an attempt is made to perform a "balance," which will empty some of the node. Here, the number of index entries on a neighboring node is checked. If there are fewer than CreatePageSize entries on the neighboring node, index entries from the overflowing node are moved over to the neighboring node. A node split is then no longer necessary.

This method is better in that it allows for filling all nodes with keys, and therefore it conserves memory. This leads to the following benefits:

- the search time is decreased
- the size of the index file is decreased
- more keys can be kept in memory

When a continuous, sequential key (a customer number, for example) is used, the index file size may be reduced by 30% to 50% as a result of page balancing.

Deleting a Key

If a key is to be deleted, the location of the key in the tree must first be determined. This is accomplished by a search. As with key addition, it is easy to delete a key found on a leaf node. This is done by simply removing the entry from the node's index array.

If the node now has less than $\text{CreatePageSize}/2$ index entries ("underflow"), either of two actions can be taken. Index entries can be obtained from a neighbor node ("page balancing"), as described in the addition of keys. Or in the case where the neighbor would also suffer from underflow, the two pages are made into one ("merging," exactly the opposite of node splitting). The parent of the two pages, now merged into one, also shrinks by one index entry. If it also suffers from underflow, and page balancing is not possible, it is also combined into one node with a neighbor. Under certain conditions this will occur all the way to the root. If this should happen to remove the single index entry that made up the root, then the tree would shrink by one level. This is the only way that a B-tree can lose height.

Remember that the index and data files are stored separately. If an index entry is deleted without deleting the corresponding data record, the data record remains as a "data orphan" in the data file, i.e. it can no longer be found in the data file through an indexed reference.

If a key is to be deleted from a location other than a leaf node in the tree, the forward pointer that points to the node with the larger keys would also be lost by the deletion of the key, and the structure of the tree would be corrupted. In order to prevent this, the next largest key in the tree is obtained and moved to the location that the deleted key occupied. Because of the key ordering this will always be located on a leaf node. In this manner the forward pointer of the deleted key remains valid. After the new key is copied to the location occupied by the deleted key, it must also be removed from its previously occupied position. This is done with the same scheme by which a key is deleted from a leaf node. In case the node from which the deleted key was removed now has less than $\text{CreatePageSize}/2$ entries, it is corrected by either node balancing or node merging.

4. Using B-Tree Filer

Compared to Chapter 3's dense conceptual descriptions, this chapter offers a welcome breeze of practicality. It describes how to put B-Tree Filer to use in your applications. The programming examples of sections C and F provide a quick start for those who learn best by example.

A. Fileblocks

A fileblock is the combination of a data file, an index file, and other information kept in memory and on disk to describe the state of the data and index files. For single user databases, the "other information" is kept in memory and in the first record of the data file, which is always reserved for B-Tree Filer's use. For databases opened in save mode and for multi-user databases, the fileblock incorporates a third file, called the dialog file. In save mode, this file temporarily holds records or keys until they are successfully written to the actual data or index file. In multi-user applications, the dialog file holds additional information required to manage the interactions among workstations.

B-Tree Filer can manage any number of fileblocks, constrained only by the number of open files allowed by the operating system. There is only one index file for every data file, independent of the number of keys defined. Therefore, if five single user fileblocks are to be opened at one time, the operating system must be able to keep ten different files open. If three multi-user fileblocks are to be opened, nine DOS files are required.

(MS-DOS controls the maximum files per process with the FILES= entry in the CONFIG.SYS file. Since the first five files are always used by DOS, be sure to account for those in the total request. Without extensions, DOS allows at most 20 simultaneously open files in a given program. B-Tree Filer offers the ExtendHandles procedure to increase the number of files under DOS 3.3 or later. See _6.H for more information.)

B-Tree Filer databases are referenced through a pointer to a fileblock. This minimizes static data allocation by using the heap to hold the fileblock information that is kept in memory. An application declares a pointer variable of type IsamFileBlockPtr for each fileblock. This consumes only four bytes in the data segment.

When a fileblock is opened, several boolean parameters are used to specify properties of the fileblock. These parameters are described fully in BOpenFileBlock in Chapter 5, but two of them are worth mentioning here. The first one, Net, specifies whether the fileblock is stored on a file server and is to be used by more than one workstation simultaneously. The second one, Save, determines whether the fileblock is opened in "save mode," a special mode that B-Tree Filer provides to increase data integrity.

In save mode, B-Tree Filer uses the dialog file to keep a backup of each modified portion of a fileblock while changes are being made. In that way, if an error occurs during the operation, changes can be undone and the fileblock restored to a known condition. If the operation is successful, all changes are flushed to disk and the backup information in the dialog file is removed. Even though the dialog file holds backup information for only a single atomic operation (adding a key or adding a record, for example), save mode can increase data integrity enormously. Reliability comes at the expense of performance, however, so save mode is not always desirable. See _4.G for a discussion of the pros and cons.

B. Program Organization

To access B-Tree Filer's routines, Use the FILER unit in your main program and in any other units where needed. The program must be organized with the following sequence of actions.

1. Initialize the FILER unit and create a buffer by calling `BTInitIsam`.
2. Optionally create one or more fileblocks by calling `BTCreateFileBlock`.
3. Open one or more fileblocks by calling `BTOpenFileBlock`.
4. Use the B-Tree Filer procedures and functions on the opened fileblocks.
5. Close opened fileblocks with `BTCloseFileBlock`.
6. Dispose of the page buffer and shut down the FILER unit by calling `BTExitIsam`.

This sequence can be repeated as often as desired. In addition, files can be opened and closed as desired between steps two and five, inclusive.

Another important program action should be considered at the highest level, and that is handling. B-Tree Filer exports the boolean variable `IsamOK`, which is set `True` for success, and `False` when an error occurs. An integer variable, `IsamError`, is then available for classification of the error. Almost every call to a B-Tree Filer routine should be followed by a check of `IsamOK`.

The value of the parameter `ExpectedNet` passed to `BTInitIsam` determines whether B-Tree Filer is dealing with a real network or is emulating a network on a single PC. By passing `NoNet` in this parameter when the FILER unit has been compiled with any network define other than `NoNet` (in `BTDEFINE.INC`), you can develop a network application on a PC not currently connected to the network, or you can develop an application that runs in single user mode today but is ready for a multi-user upgrade in the future. Of course, you can't be sure your locking logic is correct until the application is running in the real environment.

Many single user applications must eventually be ported to a network environment. To simplify this task, all B-Tree Filer procedures and function calls are available when compiled for single user mode as well as for the multiuser (network) mode. If you call a routine such as locking a fileblock that has no relevant action in single user mode, it does nothing but return a successful status code.

C. Single User Example

The following programming examples illustrate operations on B-Tree Filer fileblocks. Each major action--defining the record structure, opening the database, adding and deleting records and keys, and searching for information--is shown. Many of the routines are simply shells around Filer's lower level routines, adding error handling and specific information about the particular application. This approach is recommended for all applications based on B-Tree Filer.

The following code examples can be found in the EXAMPLE.PAS file. EXAMPLE.PAS doesn't actually do anything, but you may find it useful for cutting and pasting. It was stored in the \FILER directory by the installation program.

Data Definition

You must first define a record type to hold the information stored by the database. It could be like the following:

```
type
  PersonDef =
    record
      Del      : LongInt;
      FirstName : String[20];
      LastName  : String[25];
      Street    : String[30];
      City      : String[30];
      State     : String[2];
      ZipCode   : String[9];
      Telephone : String[15];
      Age       : Integer;
    end;
```

The first field, Del, of type LongInt, must be initialized to zero by the application. All s should begin with such a field. B-Tree Filer overwrites these first four bytes when it deletes the record, in order to maintain a linked list of reusable space within the data file. Many of B-Tree Filer's facilities, particularly those for rebuilding and reorganizing fileblocks, determine whether a given record is valid by checking whether the first four bytes of the record are zero or not. (Deleted records always have a non-zero value, valid records should have zero.) These facilities will not work if a field such as Del isn't defined.

The other fields here are given for purposes of example. B-Tree Filer's only restriction is that the record be 21 bytes or larger in size. This restriction is imposed by the fact that the first record of any data file holds information describing the database as a whole, allowing B-Tree Filer to open it without additional files.

To access the corresponding fileblock, a variable of type IsamFileBlockPtr must be declared. Typically, this variable will be global to an entire program.

```
var
  PF : IsamFileBlockPtr;           {Symbolic access to the database}
```

Although the following routines access this global variable directly, they could just as well pass an IsamFileBlockPtr as a parameter. When an application is managing multiple fileblocks, it's recommended to pass the fileblock pointers as parameters.

Allocating Page Buffers

B-Tree Filer uses buffers to hold as many B-tree nodes in memory as will fit, thereby optimizing search times. Its algorithm requires that at least MaxHeight nodes be held in memory at once. The BTInitIsam routine is used to allocate these page buffers. It allows you to reserve heap space for other uses, then allocates the remaining space for buffers.

```
procedure AllocatePageBuffer (HeapToRemain : LongInt);
var
  NumberOfPages : LongInt;
begin
  NumberOfPages := BTInitIsam(NoNet, HeapToRemain, 0);
  if not IsamOK then begin
    {Insufficient memory}
    Halt;
  end;
end;
```

In this case BTInitIsam returns an error only if it was unable to allocate at least MaxHeight pages. If IsamOK is False, the returned value contains the number of pages that it could have allocated, but didn't. The typical response to an insufficient memory error is simply to write an error message and halt. Note that in real mode BTInitIsam can also use EMS memory for the page buffers; see Chapter 5 for more details.

For protected mode and Windows targets, you potentially have a very large heap, in the order of megabytes. In this case, you no longer need to ask how much memory the program needs elsewhere. Instead you should ask how much memory B-Tree Filer can profitably use. Giving B-Tree Filer too much memory in a network environment is wasteful (the majority of the time the index pages must be reloaded due to changes by other workstations). Another factor to consider is that generally Filer's buffers take one selector each, and selectors can be rare commodities in protected mode. The best advice is to pass a value of MemAvail-HeapSizeForFiler for the HeapToRemain parameter, where HeapSizeForFiler is some value that you determined elsewhere (for example, a fixed 512K, or 100*size of an index page buffer, etc). For the formula to calculate the size of an index page buffer, see the definition of BTInitIsam in Chapter 5.

Creating a Fileblock

It is to your advantage to declare global constants describing the length of each key in a fileblock. These are used in more than one place and when the inevitable day comes when you want to adjust the key definitions, you'll appreciate having the constants.

```
const
  Key1Len = 30;           {First and last name}
  Key2Len = 5;            {ZipCode}
  Key3Len = 15;           {Telephone}

function CreateFile : Boolean;
var
  IID : IsamIndDescr;
begin
  IID[1].KeyL := Key1Len; IID[1].AllowDupK := False;
  IID[2].KeyL := Key2Len; IID[2].AllowDupK := True;
  IID[3].KeyL := Key3Len; IID[3].AllowDupK := True;
  BCreateFileBlock('ADDRESS', SizeOf(PersonDef), 3, IID);
  CreateFile := IsamOK;
end;
```


CreateFile creates the necessary data and index files for a new fileblock. In this example, BTreeCreateFileBlock creates the data file 'ADDRESS.DAT' and the index file 'ADDRESS.IX'. The Pascal function SizeOf provides the length of the data record, i.e., the record size.

The parameter that describes the keys is defined in the variable IID of type IsamIndDescr. IID must be initialized before calling BTreeCreateFileBlock. In the example, there are three keys for every data record. (B-Tree Filer allows up to 100 keys for every record, with each key up to 30 characters long. These values can be adjusted by changing constants in FILER.CFG. See Chapter 5 for more details.) The IsamIndDescr describes each key, providing the maximum length of each string and whether the key is a (unique) or (possibly duplicate) key. Keys can be created from any data desired; they are not restricted to data from any one field. In this example, the primary key is the concatenation of the last and first name. The zip code and telephone number are entered as secondary keys. Note that B-Tree Filer does not require a primary key for each fileblock.

After a fileblock is created, it must still be opened in order to use it.

Opening a Fileblock

```
function OpenFile : Boolean;
begin
  BTreeOpenFileBlock(PF, 'ADDRESS', False, False, False, False);
  if not IsamOK then begin
    OpenFile := False;
    {Error reporting code that examines
     IsamError can go here. Corrective action may
     be taken, for example by reconstructing a defective
     index file as described in section 6.D.}
    Exit;
  end else
    OpenFile := True;
end;
```

OpenFile shows how a function to open an existing fileblock might look. It contains a call to the Filer procedure BTreeOpenFileBlock that opens the fileblock in normal mode for single user access. The four boolean parameters passed to BTreeOpenFileBlock specify alternate open modes and are described in Chapter 5. By using the global variables IsamOK and IsamError, it is possible to detect errors that occur while opening the files. Notice that no information is needed about the record length or the keys to open an existing fileblock. B-Tree Filer reads this information from the data file and keeps the information in the corresponding fileblock in memory.

Closing a Fileblock

```
function CloseFile : Boolean;
begin
  BTreeCloseFileBlock(PF);
  if not IsamOK then begin
    CloseFile := False;
    {Error handling}
    Exit;
  end else
    CloseFile := True;
end;
```

It is essential that every open fileblock be closed before an application halts. Otherwise buffered index information may not be written to disk and an error will occur during the next use of the fileblock. The function CloseFile can handle this task. It closes the index and data files by calling

BTCloseFileBlock. Error handling can take place here as well, although at this point there are few corrective actions to take, short of rebuilding the fileblock.

Data and Index File Operations

It is advantageous to write a single function that returns the keys with which the data file can be accessed. This ensures that keys associated with a given data record are consistent throughout an entire application. CreateKey is an example of such a function, creating any of the three different key strings from the record fields.

```
{ $F+ } {Routine should be global}
function CreateKey(var P; KeyNr : Integer) : IsamKeyStr;
begin
  with PersonDef(P) do
    case KeyNr of
      1 : CreateKey := StUppcase(Pad(LastName, 20)+Pad(FirstName,
10));
      2 : CreateKey := Copy(ZipCode, 1, 5);
      3 : CreateKey := Copy(Telephone, 1, 15);
      else
        CreateKey := '';
      end;
    end;
  end;
end;
```

This function is a little more complicated than it needs to be now, so that it will also work with other B-Tree Filer units to be described later. The REINDEX unit calls a routine like CreateKey indirectly when it is rebuilding an index file. The routine must meet specific requirements in order for the indirect call to work correctly.

CreateKey is declared far and global by using the { \$F+ } directive (or the Far keyword). The routine's first parameter is an untyped Var parameter and the second parameter is the KeyNr of type integer. The Var parameter passes the data record into the routine. Because it's untyped it will work regardless of the record type; however, the routine must typecast the parameter into the actual data type in order to refer to the fields of the record.

StUppcase is a function that converts the argument string to upper case. Pad is a function that either truncates a string to the specified length or pads it with blanks to reach that length. It's important that the CreateKey routine never return a key string longer than the allowed length for a given index; otherwise routines such as BTAddKey will generate an error. It's not essential for CreateKey to pad all key strings to the same length. However, when two fields are concatenated to create a key string, as with first and last names here, it is important to pad the first field to its maximum length before adding the second field. Otherwise, the index sorting sequence will not be correct.

Now that files exist and keys can be created, you need routines to add, delete, change, and search for data.

Adding Data to the Fileblock

Inserting a new record requires two steps. BTAddRec is called first to add the record to the data file. It returns a RefNr (data record reference) to describe the record's location within the file. The second step requires adding every key to the index file with BTAddKey. The following example should clarify the situation:

```
procedure UndoAdd(P : PersonDef; RefNr : LongInt; LastKey :
Integer);
var
```

```

    KeyNr : Integer;
    Key : IsamKeyStr;
begin
    for KeyNr := 1 to LastKey do begin
        Key := CreateKey(P, KeyNr);
        BTDeleteKey(PF, KeyNr, RefNr, Key);
        if not IsamOK then
            {Abort: too many errors}
            Halt;
        end;
    end;
end;

function AddRecord(P : PersonDef) : Boolean;
var
    KeyNr : Integer;
    RefNr : LongInt;
    Key : IsamKeyStr;
begin
    AddRecord := False;
    BTAddRec(PF, RefNr, P);
    if not IsamOK then begin
        {Error handling}
        Exit;
    end;
    for KeyNr := 1 to BTNrOfKeys(PF) do begin
        Key := CreateKey(P, KeyNr);
        BTAddKey(PF, KeyNr, RefNr, Key);
        if not IsamOK then begin
            {Remove keys added so far}
            UndoAdd(P, RefNr, KeyNr-1);
            {Remove the new record}
            BTDeleteRec(PF, RefNr);
            {Error handling}
            Exit;
        end;
    end;
    AddRecord := True;
end;

```

After the new data record is successfully entered into the file with BTAddRec, the keys are created one by one within a for loop, and are added to the index file using BTAddKey. The variable RefNr contains the location to which the new data record was written.

If errors are encountered during the key operations (e.g. IsamError = 10230, duplicate key), the function UndoAdd attempts to delete all keys added so far, so that AddRecord can return gracefully by not corrupting the fileblock. However, if more errors are encountered during the deletion, attempting further recovery will probably cause more problems than it solves, so the program should be aborted. AddRecord returns True if all operations are successful.

Removing Data from a Fileblock

The function DeleteRecord is analogous to AddRecord.

```

procedure UndoDel(P : PersonDef; RefNr : LongInt; LastKey :
Integer);
var
    KeyNr : Integer;
    Key : IsamKeyStr;
begin

```

```

    for KeyNr := 1 to LastKey do begin
        Key := CreateKey(P, KeyNr);
        BTAddKey(PF, KeyNr, RefNr, Key);
        if not IsamOK then
            Halt;
        end;
    end;
end;
function DeleteRecord(P : PersonDef; RefNr : LongInt) : Boolean;
var
    KeyNr : Integer;
    Key : IsamKeyStr;
begin
    DeleteRecord := False;
    {Assure record not already deleted}
    if P.Del <> 0 then
        Exit;
    for KeyNr := 1 to BTNrOfKeys(PF) do begin
        Key := CreateKey(P, KeyNr);
        BTDeleteKey(PF, KeyNr, RefNr, Key);
        if not IsamOK then begin
            {Add back keys that have been deleted so far}
            UndoDel(P, RefNr, KeyNr-1);
            {Error handling}
            Exit;
        end;
    end;
    end;
    BTDeleteRec(PF, RefNr);
    if IsamOK then
        DeleteRecord := True;
    end;
end;

```

To avoid an attempt to delete an already deleted record, the P.Del field is examined to confirm that the data record has not yet been deleted. Then you can proceed in just the opposite manner compared to adding a record. First the keys are deleted, and then the data record. It's debatable whether to attempt to undo a deletion in case of an error. UndoDel is shown here, but you may decide it makes sense to continue deleting keys even if one DeleteKey fails.

Modifying Records

When an existing record is to be modified, not only must the data file be updated, but also the old keys must be deleted and new keys added. This makes for a fairly complicated procedure, as shown in ModifyRecord.

```

function CheckRecord(P, POld : PersonDef) : Boolean;
begin
    {Verify that: new record has valid keys,
     new record differs from old}
    CheckRecord := True;
end;

function ModifyRecord(P, POld : PersonDef; RefNr : LongInt) :
Boolean;
var
    KeyNr : Integer;
begin
    ModifyRecord := False;
    if not CheckRecord(P, POld) then
        Exit;
    for KeyNr := 1 to BTNrOfKeys(PF) do begin

```

```

{Update modified keys}
if CreateKey(P, KeyNr) <> CreateKey(POld, KeyNr) then begin
  BTDeleteKey(PF, KeyNr, RefNr, CreateKey(POld, KeyNr));
  if not IsamOK then
    if IsamError = 10220 then
      {Key already deleted, ignore the error}
    else begin
      UndoAdd(P, RefNr, KeyNr-1);
      UndoDel(POld, RefNr, KeyNr);
      Exit;
    end;
  BTAddKey(PF, KeyNr, RefNr, CreateKey(P, KeyNr));
  if not IsamOK then begin
    UndoAdd(P, RefNr, KeyNr-1);
    UndoDel(POld, RefNr, KeyNr);
    Exit;
  end;
end;
end;
BTPutRec(PF, RefNr, P, False);
if not IsamOK then begin
  UndoAdd(P, RefNr, BTNrOfKeys(PF));
  UndoDel(POld, RefNr, BTNrOfKeys(PF));
  Exit;
end;
ModifyRecord := True;
end;

```

The CheckRecord routine should verify that the new record, P, generates valid keys (e.g., its primary keys must not be empty). CheckRecord should also check that the new and old records aren't identical, since that would lead to a lot of wasted effort.

ModifyRecord removes each changed key from the index file and adds the new key. If an error occurs, it calls UndoAdd and UndoDel to recover as gracefully as possible from the error. Then it calls BTPutRec to overwrite the old data record with the new.

Next or Previous Data Record

NextPrevRecord shows how a function can be implemented to scan through a data file in index order. The "access pointer" must be positioned over a known key before NextPrevRecord can be called for the first time. This can be done by calling any of the functions BTFindKey, BTSearchKey, BTFindKeyAndRef, BTSearchKeyandRef, or BTClearKey.

```

function NextPrevRecord(var P : PersonDef; var RefNr : LongInt;
                        KeyNr : Integer; var Key : IsamKeyStr;
                        Next : Boolean) : Boolean;
begin
  NextPrevRecord := False;
  if Next then begin
    BTNextKey(PF, KeyNr, RefNr, Key);
    if not IsamOK and (IsamError = 10250) then
      {There was no next key. Move to first key in the file}
      NextKey(PF, KeyNr, RefNr, Key);
    end else begin
      BTPrevKey(PF, KeyNr, RefNr, Key);
      if not IsamOK and (IsamError = 10260) then
        {There was no previous key. Move to last key in file}
        BTPrevKey(PF, KeyNr, RefNr, Key);
    end;
  end;
end;

```

```

end;
if not IsamOK then
  Exit;
BTGetRec(PF, RefNr, P, False);
if not IsamOK then begin
  {Error handling}
  Exit;
end;
NextPrevRecord := True;
end;

```

Searching for Data Records

To locate a data record by indexed access, the search key string must be specified along with the corresponding index number. The following routine demonstrates an indexed search.

```

function FindRecord(var P : PersonDef; var RefNr : LongInt; KeyNr :
Integer;
                    var Key : IsamKeyStr) : Boolean;
begin
  FindRecord := False;
  BTSearchKey(PF, KeyNr, RefNr, Key);
  if not IsamOK then begin
    {Determine why SearchKey failed, for example:
     IsamError = 10210 Neither the key nor any larger was found.}
    Exit;
  end;
  BTGetRec(PF, RefNr, P, False);
  if not IsamOK then begin
    {Error handling}
    Exit;
  end;
  FindRecord := True;
end;

```

The subroutine is aborted if IsamOK is False after calling BTSearchKey. SearchKey returns True even if no exact match was found, as long as another record with a larger key is available. The actual key is returned to FindRecord's caller through FindRecord's Var parameter, Key. If any matching key is found, BTSearchKey also returns the associated data record reference number in RefNr. Using this number, the data record is then read by BTGetRec and returned in P. The procedure BTFindKey should be used instead of BTSearchKey if an exact key match is required.

Searching for a record based on a non-indexed field is a little trickier, and can be handled in more than one way. One method ignores index order altogether and simply reads each record from the data file, skipping any records marked as deleted. Another method is to use the BTNextKey routine to search the records in sorted order by key. This method is advantageous when a two part search is required: the first part an indexed search on a possibly duplicate key, and the second part a non-indexed search on a different field. Using BTNextKey allows the routine to start on the first possible match and to quit as soon as no more matches are possible. ScanForRecord exemplifies this method:

```

function MatchedRecord(P, Q : PersonDef) : Boolean;
begin
  {Return True if P and Q match based on some criteria, for
  example...}
  MatchedRecord := (StUppcase(P.City) = StUppcase(Q.City));
end;

```

```

function ScanForRecord(var P : PersonDef; KeyNr : Integer;
                      var RefNr : LongInt) : Boolean;
var
  Done : Boolean;
  Goal : PersonDef;
  Key : IsamKeyStr;
begin
  ScanForRecord := False;
  Goal := P;
  Done := False;
  repeat
    BTNextKey(PF, KeyNr, RefNr, Key);
    if not IsamOK then
      {Probably reached the largest key}
      Done := True
    else begin
      BTGetRec(PF, RefNr, P, False);
      if not IsamOK then begin
        {Error handling}
        Done := True;
      end else if MatchedRecord(P, Goal) then begin
        {Found a match}
        Done := True;
        ScanForRecord := True;
      end;
    end;
  until Done;
end;

```

This example searches for a record that "matches" the initial record passed in the P parameter. The search begins with the record one beyond that associated with the current sequential pointer, and continues on to the record having the largest key of type KeyNr. The MatchedRecord routine compares the current record to the search goal and, if a match is found, ScanForRecord returns the record number (in RefNr) and its contents (in P).

The most important operations of B-Tree Filer were presented in these small examples. These routines were kept simple in order to illustrate the concepts clearly. The NETDEMO example program contains more robust examples.

D. Locking Techniques

B-Tree Filer supports six levels of locking, as explained in this section. In the following description of the locking levels, the routines that activate and deactivate such locks are listed after the level number. In every case, it is important for a workstation to minimize the time that it retains a lock.

Level 1: **BTLockAllOpenFileblocks** and **BTUnlockAllOpenFileblocks**

These procedures act on all network fileblocks that are open at the time of the call. After **BTLockAllOpenFileBlocks** returns successfully, all network fileblocks of the calling workstation are available for its exclusive use. No other workstation can read or write the locked fileblocks. With rare exceptions, a network fileblock must be locked before it can be modified by using **BTAddRec**, **BTAddKey**, **BTDeleteRec**, **BTDeleteKey**, or **BTPutRec**.

No "deadlock" problems can occur with level 1 locking. (A deadlock occurs when two workstations are simultaneously trying to lock the same two files, and one of them locks the first file while the other one locks the second file. Then both could possibly wait forever waiting for the other file to become free.) **BTLockAllOpenFileBlocks** either locks all open network fileblocks, or it locks none of them. The application should check the value of **IsamOK** to determine whether **BTLockAllOpenFileBlocks** was successful.

It is especially critical when using level 1 locking to minimize the amount of time the fileblocks remain locked. One specific situation to avoid is waiting for interactive keyboard input before unlocking again.

Level 2: **BTLockFileBlock** and **BTUnlockFileBlock**

These procedures affect a single network fileblock. After **BTLockFileBlock** returns successfully, the calling workstation has exclusive access to the specified fileblock and can modify it using the procedures mentioned above.

A deadlock may also occur as described under level 1 locks. The calling program must take steps to prevent this situation. Methods to avoid deadlocks are described in [_4.F](#).

Level 3: **BTLockRec** and **BTUnlockRec**

BTLockRec locks a single record of a specific network fileblock. This locking step can be combined with the locking steps 1 and 2. Possible deadlocks must be avoided by the calling program.

Level 4: Not directly implemented by B-Tree Filer

You can supply routines to use any other locking methods that pertain to a data record (e.g., a "shareable lock" under Novell). B-Tree Filer supports this through the **BTGetRecordInfo** procedure. Given an open fileblock and a record number, **BTGetRecordInfo** returns the handle of the data file that contains the record, and the record's file offset and length. This is sufficient for a custom locking routine to access the record.

Level 5: **BTReadLockAllOpenFileBlocks** and **BTUnlockAllOpenFileBlocks**

These procedures act upon all network fileblocks that are open at the time of the call. After **BTReadLockAllOpenFileBlocks** returns successfully, no workstation can write to the fileblocks. Other workstations can continue to read from the fileblocks and they can also call **BTReadLockAllOpenFileBlocks** themselves. **BTUnlockAllOpenFileBlocks** removes read locks from all the fileblocks just as it would remove level 1 or 2 locks.

There are no deadlock problems possible with read locks. Note that it is possible to call `BTLockAllOpenFileBlocks` after calling `BTReadLockAllOpenFileBlocks` without calling `BTUnlockAllOpenFileBlocks` in between.

Level 6: `BTReadLockFileBlock` and `BTUnlockFileBlock`

This locking level is just like level 5 except that it affects only a specified network fileblock. B-Tree Filer uses temporary level 6 locks frequently to guarantee that the fileblock remains stable while it completes a particular operation.

General Comments About Locking

Almost all methods of writing to a network fileblock require that it first be locked at level 1 or 2. The following routines will fail unless a degree 1 or 2 lock is in force:

```
BTPutRec  
BTAddRec  
BTDeleteRec  
BTAddKey  
BTDeleteKey
```

(`BTPutRec` may in some circumstances be used to update a fileblock without a level 1 or 2 lock. See Chapter 5 for details.)

The mechanism used to lock a fileblock can be described as follows:

- for a level 1 or 2 write lock, a section of the dialog file is physically locked. The size of this section is proportional to the maximum numbers of workstations and indexes for which B-Tree Filer is configured. This lock fails if another workstation already has the fileblock locked with a level 5 or 6 read lock.
- for a level 5 or 6 read lock, one workstation's worth of the dialog file section is locked. This lock fails if another workstation already has the fileblock locked for writing, but it does not fail if another workstation also has the fileblock locked for reading.
- for a level 3 record lock, the first four bytes of the physical record in the data file are locked.

Procedures that access a network fileblock (with three exceptions, see below) always read the dialog file and thereby recognize the locks.

Since the data file is not directly affected by locks of degree 1, 2, 5, or 6, it is possible for another workstation to circumvent the lock and read or even modify a record. `BTGetRec` with a parameter `ISOLock` of `True` will read a fileblock locked at level 1 or 2. `BTPutRec` with a parameter `ISOLock` of `True` will modify a record even if the fileblock is locked at level 1 or 2. Of course, these routines must be used with great care.

`BTGetRecReadOnly` will read from a locked fileblock even if the record itself has been locked at level 3. It cannot read the first four bytes of the record if those are locked. However, if the data record reserves the first four bytes for a deletion marker, no real information is lost.

It is important to note that if record locks of level 3 or 4 are used by one workstation, another station will fail to access the locked record even if it previously locked the entire fileblock using level 1 or 2. In most cases you should avoid record locks altogether since they complicate application design.

Locking Retries

The lock routines BTLockFileBlock, BTReadLockFileBlock, and BTLockRec automatically retry locking attempts within a configurable time period. This time period is defined in the global variable IsamLockTimeOut and has a default value of 768 milliseconds (ms).

To give BTLockFileBlock a better chance to place a full fileblock lock in the face of many read lock attempts by other workstations, its timeout value is higher. The factor by which it is higher is determined by the global variable IsamFBlockTimeOutFactor. The default value of 4 gives a timeout of approximately 3 seconds.

The implementation of a lock with timeout is different in Novell networks than in MsNet compatible networks. In a Novell network, the timeout value is passed with the locking request. If the lock cannot be performed immediately, the server places it in a wait queue. The lock fails only if the wait time elapses before the request is filled. The Novell approach is "fair" in the sense that the first station requesting a lock is guaranteed to get it at the next safe opportunity.

An MsNet network has no timeout value for a lock. Therefore B-Tree Filer reattempts a failed lock after a short pause. If the pause times exceed the timeout value and the lock is still not successful, the locking attempt fails. The pause time is specified in the global variable IsamDelayBetwLocks (the default value is 64 ms). This approach is less fair than Novell because another station might, by chance, gain control of the fileblock even though your station asked for it first.

You can change the timeout value during execution of the program by changing IsamLockTimeOut, IsamFBlockTimeOutFactor, or IsamDelayBetwLocks.

In a heavily used network where many workstations are placing and releasing read locks continuously, you may find that BTLockFileBlock never has a chance to succeed. In this kind of situation it might be better to institute some kind of semaphore system so that the updates to the fileblock can continue with minimum disruption. When a workstation needs to place a level 1 or 2 lock it raises a semaphore and then tries to place the lock with BTLockAllFileBlocks or BTLockFileBlock. The other workstations which are placing and releasing read locks monitor this semaphore, and if it goes high, wait for it to be lowered. Once the workstation that needed the full lock removes it, it lowers the semaphore, releasing the other workstations. The semaphore could be either a semaphore provided by the network operating system (see _8.A.6 for details on Novell NetWare's implementation) or it might be the presence/absence of a file in a shared directory.

TRAFFIC: Determining Optimal Timeout Values

TRAFFIC is a utility for optimizing the lock parameters IsamFBlockTimeOutFactor, IsamLockTimeOut, and IsamDelayBetwLocks in a B-Tree Filer application.

TRAFFIC uses a fileblock named 'TRAFFIC'. The fileblock has a record length of 353 bytes and uses 4 indexes, similar to many applications. The TRAFFIC source code can be customized to bring it closer to a particular application if necessary.

TRAFFIC executes two elementary operations:

1. Write: the fileblock is locked, a random record and the corresponding indexes are added, and then the fileblock lock is removed.
2. Read: the fileblock is read locked, then after reading a random key of a random index, the matching record is read, and finally the read lock is removed.

After every elementary operation, a short delay is executed. The delay time is an adjustable parameter and has a default of 100 ms.

To give you a quick impression of the operations as they occur on each workstation on the network, a character is displayed for each operation:

- : Write was successfully executed
- . Read was successfully executed
- X Fileblock lock before write attempt failed
- x Read lock before read attempt failed
- O Random record for write attempt already exists
- o No record to read was found (only occurs for an initially empty file)

The first four characters are the most relevant during testing. "O" and "o" serve only for status notification and can generally be ignored.

Starting the Tests

Assure that the appropriate conditional symbol for your network is defined in BTDEFINE.INC and compile TRAFFIC. You may want to define both MsNet and Novell in order to test the effect of both locking methods on a Novell network. Then copy TRAFFIC.EXE to a directory on your file server where all workstations can access it. Run TRAFFIC without any parameters to get a summary of the command line options.

The following example assumes that the test is executed on a Novell network and a total of 5 workstations (A to E) participate.

To start the first test, call the program from workstation A in the following manner:

```
TRAFFIC /N /W
```

These options inform TRAFFIC that it is running on a Novell network and that the write test should be performed.

TRAFFIC begins writing random records and displays a series of ":" characters, possibly interspersed with "O" characters which indicate a random duplication of records.

Now start TRAFFIC on workstations B through E with the following command line:

```
TRAFFIC /N /R
```

Each of these workstations now performs random reads, indicated on screen by a series of "." characters.

Press <CtrlC> to stop the tests.

Evaluation of the Tests

The ideal case is when no "X" or "x" characters are displayed. For the test described, this is unlikely since this test constitutes a high network load. Therefore a few "X" or "x" characters are no reason for concern.

What is to be done if "X" or "x" appear frequently or in blocks? TRAFFIC allows you to adjust the B-Tree Filer locking parameters in order to minimize the frequency of the locking failures.

TRAFFIC Command Line Parameters

The general form of the TRAFFIC command line is:

TRAFFIC NetworkType TestType [Options]

NetworkType and TestType must always be specified in the order shown. The following options are available.

NetworkType options:

```
/N      Novell network (uses NetWare-specific lock calls).
/M      MsNet or compatible (uses generic DOS locking).
/O      no network (no locking, and therefore usable only on a
single workstation. Useful for speed comparisons)
```

TestType options:

```
/R      Read test. The TRAFFIC fileblock must already exist or the
test stops with error 9903.

/W      Write test. If the TRAFFIC fileblock doesn't already exist,
it is created in the current directory.

/Bn     Mixed Read and Write Test. "n" is a number between 1 and
255 that indicates how many read operations occur before a write is
done. For example /B10 indicates that TRAFFIC performs 10 reads
followed by one write. If no fileblock exists, the program stops
with the error code 9903.
```

Other Options:

```
/Dn     "n" specifies the delay in milliseconds after each
operation. This value controls the load on the network. Although
the default value of 100 ms on 5 workstations already causes a very
high network load, the load can be raised still further by reducing
the delay time. The value should be chosen to emulate the operation
of your intended application. Typical values for a writing
workstation are between 50 ms and 10000 ms. Typical values for a
reading workstation are between 10 ms and 500 ms.

/Tn     "n" specifies the timeout value in milliseconds for a
locking attempt. The default value is 768 ms. This value is the
retry time for a read lock. For a write lock this value is
multiplied by the factor supplied with the /F option. Increase the
/T value if a reading workstation displays the "x" character too
frequently.

/Fn     "n" specifies the factor that multiplies the /T timeout
when a fileblock write lock is placed. The default factor is 4.
Hence, by default a write lock is retried for about 3 seconds.
Increase the /F value if the /T timeout has been adjusted already
and the writing workstation displays "X" too frequently. The total
timeout for a write lock should not exceed about 5 seconds or your
end user will think that the program has hung.

/Ln     "n" specifies the delay in milliseconds after a failed
locking attempt. The default value is 64 ms. This value is used
only for the MsNet network type. (Under Novell, the file server
queues lock requests and grants the request as soon as possible.)
Generally you won't need to change the default. However, for some
networks based on serial ports, you may need to increase the value
to avoid overloading the network communication channels.
```

E. Converting Single User Programs

It is easy to convert an existing single user program for network capabilities by using level 1 locking exclusively.

- Call `BTLockAllOpenFileBlocks` before every write access, and call `BTUnlockAllOpenFileBlocks` afterwards. Check that the locks were successfully placed and retry if they were not.
- Add checks after every B-Tree Filer call to handle the additional errors that may occur (lock violations, etc.) and retry when appropriate.
- Follow two rules for performance optimization: do not perform any keyboard input while a lock is active; and perform as many B-Tree Filer calls as possible within each lock.

While this approach is straightforward, it does not provide optimum performance when a program has several unrelated fileblocks open simultaneously. For this reason individual fileblock locks (level 2) are often a better choice, but not without added complication. Additional steps must be taken to prevent deadlock if level 2 locks are used. Consider the following scenario:

- Workstation 1 successfully places a lock on fileblock A.
- Station 2 successfully places a lock on fileblock B.
- Station 1 now attempts to lock fileblock B. Since that is not possible, station 1 continually requests the lock.
- Station 2 now attempts to lock fileblock A. Since that is not possible, station 2 continually requests the lock.
- Both stations will wait indefinitely for the other to give up its lock on the other fileblock.

The best solution is to limit the number of retries. When a station reaches the limit, it should give up and release all related locks it has already been granted. It should then wait for a quasi-random amount of time before trying again. The delay may be provided by requiring an operator to request a transaction again, or by using the `Random` function to generate a non-interactive program delay. In this way, one of the workstations will gain a lead on the other and beat the deadlock.

Programs using locks of level 3 or 4 must account for additional locking conflicts:

- The same deadlock just described for entire fileblocks may occur with single records. This can be solved the same way.
- A level 1 or 2 lock of an entire fileblock no longer guarantees that a workstation can access any particular existing record, since another station may have a lock of degree 3 or 4 on that record. This can be solved by setting the `ISOLock` parameter of `BTGetRec` and `BTPutRec` to `True`.

Further difficulties not directly related to locking may arise in the network environment. Consider the following three scenarios:

1. A record that was edited by workstation 1 is deleted or changed by station 2 in the meantime.
2. A record that is to be deleted by workstation 1 is deleted or changed by station 2 in the meantime.
3. A listing being printed by workstation 1 is made inaccurate when station 2 modifies the index during the listing. (For example, if station 2 modifies the index of a record already printed, and reindexes the record to a position beyond the current end of the listing, that record will be printed twice.)

The simplest solution to these problems is obtained simply by locking the entire fileblock throughout the process of editing, deletion, or listing. This solution is rarely adequate, however, since other workstations are so frequently unable to access the data.

A better idea is to lock just the single record being edited or deleted. Even this will interrupt a listing being performed by another station, unless the listing routine uses BTGetRecReadOnly to read the records.

By using the following algorithms, you can avoid locking even the individual record while interactive entry occurs.

Editing

```
Read the record
Interactively perform the edit
Lock the record
Reread the record
Compare with the original record
If identical
    Lock the fileblock
    Write back the modified record
    If a key was modified
        Delete the old keys
        Add the new keys
    Unlock the fileblock
    Unlock the record
If not identical (another station modified it)
    Unlock the record
    Present the new data to the user and perform the edit again
```

Deleting

```
Read the record
Interactively obtain confirmation to delete
Lock the record
Reread the record
Compare with the original record
If identical
    Lock the fileblock
    Delete the record and all keys
    Unlock the fileblock
    Unlock the record
If not identical (another station modified it)
    Unlock the record
    Start from the beginning
```

If record locks are not used elsewhere in an application, it's easy to avoid them here too. The simplest way is to move the fileblock lock up a few steps to the position of the record lock. A read lock can also be used to reduce the time while other stations can't read from the fileblock. Consider the following slightly modified versions of the algorithms (modified lines are indicated by asterisks).

Editing

```
Read the record
Interactively perform the edit
Read lock the fileblock **
Reread the record
Compare with the original record
```

```

If identical
  Lock the fileblock
  Write back the modified record
  If a key was modified
    Delete the old keys
    Add the new keys
  Unlock the fileblock
**
If not identical (another station modified it)
  Unlock the fileblock **
  Present the new data to the user and perform the edit again

```

Deleting

```

Read the record
Interactively obtain confirmation to delete
Read lock the fileblock **
Reread the record
Compare with the original record
If identical
  Lock the fileblock
  Delete the record and all keys
  Unlock the fileblock
**
If not identical (another station modified it)
  Unlock the fileblock **
  Start from the beginning

```

These algorithms introduce a small window of opportunity for another workstation to place a full lock on a fileblock. Before B-Tree Filer can place a full lock on the fileblock it must first remove the read lock, thereby allowing another workstation a small amount of time to place a full lock. If this might be a problem in your environment, use the first pair of algorithms for editing and deleting records.

Read locks are also valuable in solving difficulty 3 described above. By placing a read lock during the generation of a critical report, an application allows other users to continue reading from the fileblock, but prevents anyone from corrupting the reporting by modifying the fileblock.

Examples in the following section (as well as in the NETDEMO program) demonstrate working implementations of these algorithms.

F. Network Example

The examples already provided in _4.C are largely applicable here as well. Please refer to that section if you haven't already read it.

This section contains code examples for dealing with some of the thorny issues discussed in the previous section. The source code can be found in NETEXAMP.PAS, which was stored in the \FILER directory by the installation program.

Reacting to Locks

Consider the function FindRecord from _4.C. There are several ways to make this routine network-aware. You can simply add appropriate error handling after calling the existing routine:

```
if not FindRecord(P, RefNr, KeyNr, Key) then
  if IsamErrorClass = 2 then begin
    {Lock error}
    Writeln('File or record locked');
    {Abort operation}
  end;
```

This is simple, but not good for much. There is not even an attempt to retry the operation. In some cases, the operating system can be programmed to retry automatically. (See BtSetDosRetry in Chapter 5.) Unfortunately, this procedure is not functional for all networks. Even when it is, the application must still handle the case when a lock error occurs after the retries run out.

Therefore it's clear that you need a new routine that will retry in case of a lock, and eventually either give up or prompt the user for further direction. The following code demonstrates this concept:

```
const
  MaxRetries = 10;
  RetryCount : Integer = 0;

function IsLockError(Ask : Boolean) : Boolean;
begin
  if IsamOK or (IsamErrorClass <> 2) then begin
    {No error, or non-locking error}
    IsLockError := False;
    {Reset retry count}
    RetryCount := 0;
  end else begin
    {Lock error}
    IsLockError := True;
    inc(RetryCount);
    if RetryCount > MaxRetries then begin
      {Out of retries}
      if not Ask then
        {Abort the operation without even asking}
        IsLockError := False
      else if not YesNo('Lock error. Try again?') then
        {Abort the operation if the user says to do so}
        IsLockError := False;
      {Reset retry count}
      RetryCount := 0;
    end;
```



```

    end;
end;

```

YesNo is a routine that prompts the user for a yes/no response.

IsLockError can be called instead of polling IsamOK after each B-Tree Net operation. If IsLockError returns False, and IsamOK is also False, then a non-locking error occurred (or perhaps the retry limit was exceeded and the user chose to abort the operation).

With IsLockError in place, FindRecord can now be implemented as follows:

```

function FindRecord(var P : PersonDef;
                    var RefNr : LongInt;
                    KeyNr : Integer;
                    var Key : IsamKeyStr) : Boolean;
begin
    FindRecord := False;
    repeat
        BTSearchKey(PF, KeyNr, RefNr, Key);
    until not IsLockError(True);
    if not IsamOK then begin
        {Key not found, program error, or abortive locking error}
        Exit;
    end;
    repeat
        BTGetRec(PF, RefNr, P, False);
    until not IsLockError(True);
    if not IsamOK then begin
        {Error handling}
        Exit;
    end;
    FindRecord := True;
end;

```

This example can be extended to all other reading functions from _4.C.

Reacting to Modified Data

In a single user application, you can be sure that once a record or key is entered, it is always available until the application itself modifies or deletes it. This is not the case in a network application. For example:

```

workstation 1 reads record number 10 and displays a "delete?"
prompt
workstation 2 deletes record number 10
workstation 1's user accepts the prompt
workstation 1 deletes record number 10

```

Record 10 is deleted twice--an action that should not be performed on the data file. A single user solution was already presented in _4.C. A corresponding network implementation of the function DeleteRecord can be written as follows. As in the previous examples, the first field of each data record is used for a deletion tag.

```

var
    MaxError : Integer;

procedure SetMaxError;
var
    ErrorClass : Integer;
begin

```

```

    ErrorClass := IsamErrorClass;
    if ErrorClass > MaxError then
        MaxError := ErrorClass;
    end;

procedure UpdateUnlock(RefNr : LongInt);
begin
    if BTFFileBlockIsLocked(PF) then begin
        BTUnlockFileBlock(PF);
        if not IsamOK then
            {Hardware failure? shouldn't happen}
            SetMaxError;
        end;
    if BTRecIsLocked(PF, RefNr) then begin
        BTUnlockRec(PF, RefNr);
        if not IsamOK then
            {Hardware failure? shouldn't happen}
            SetMaxError;
        end;
    end;
end;

function UpdateLock(RefNr : LongInt) : Boolean;
begin
    UpdateLock := False;
    {Record lock is needed only if the application uses record locks
elsewhere}
    BTLockRec(PF, RefNr);
    if not IsamOK then begin
        {Record could not be locked}
        SetMaxError;
        Exit;
    end;
    {Lock the fileblock}
    LockFileBlock(PF);
    if not IsamOK then begin
        {FileBlock could not be locked}
        SetMaxError;
        UpdateUnlock(RefNr);
        Exit;
    end;
    UpdateLock := True;
end;

function MatchRecord(P1, P2 : PersonDef) : Boolean;
begin
    {Return True if P1 and P2 are the same}
    MatchRecord := True;
end;

function DeleteRecord(P : PersonDef; RefNr : LongInt) : Integer;
var
    KeyNr : Integer;
    TempP : PersonDef;
begin
    {At this point, the record to be deleted has already been read
into
    record P, and the user has verified that a deletion is to
occur.}

```

```

{MaxError is the highest error class encountered}
MaxError := 0;

{Lock the record and the fileblock}
if not UpdateLock(RefNr) then begin
    {Couldn't lock}
    DeleteRecord := MaxError;
    Exit;
end;

{Get the record again to see if it still matches the original}
BTGetRec(PF, RefNr, TempP, False);
if not IsamOK then begin
    {Shouldn't happen}
    SetMaxError;
    UpdateUnlock(RefNr);
    DeleteRecord := MaxError;
    Exit;
end;

if TempP.Del <> 0 then begin
    {Record was already deleted. That's ok since we wanted to
delete it also}
    UpdateUnlock(RefNr);
    DeleteRecord := MaxError;
    Exit;
end;

if not MatchRecord(P, TempP) then begin
    {Record was modified in the meantime. Return a "warning" error
class}
    MaxError := 1;
    UpdateUnlock(RefNr);
    DeleteRecord := MaxError;
    Exit;
end;

{Finally perform the deletion}
for KeyNr := 1 to BTROfKeys(PF) do begin
    BTDeleteKey(PF, KeyNr, RefNr, CreateKey(P, KeyNr));
    if not IsamOK then
        if IsamError = 10220 then
            {Key already deleted. Shouldn't happen, but it's still ok
if so}
        else begin
            {Error handling}
            SetMaxError;
            UpdateUnlock(RefNr);
            DeleteRecord := MaxError;
            Exit;
        end;
    end;
end;
BTDeleteRec(PF, RefNr);
if not IsamOK then
    SetMaxError;

{Unlock the record and the fileblock}
UpdateUnlock(RefNr);
DeleteRecord := MaxError;
end;

```

The UpdateLock routine locks the record to be updated, then locks the entire fileblock so that keys can be removed safely. The record lock is necessary only if other portions of the same application are using record locks for independent reasons. UpdateUnlock unlocks the record and fileblock. MatchRecord compares two data records and returns True if they are considered to be the same. An actual implementation of this routine will typically compare each field of the two records and return True only if all of them are the same.

The global variable MaxError keeps track of the worst error class encountered and DeleteRecord returns that error class. A return value of 0 means that the routine succeeded. Error class 1, normally a warning for a key not found, is used in DeleteRecord to indicate that another workstation has modified the record to be deleted. If DeleteRecord returns 1, the application should warn the user, redisplay the record, and determine whether the record should still be deleted. Error class 2 is a locking error, as usual, and means that another station has the fileblock or the individual record locked. An appropriate response is to delay for a short time and retry for a specified number of times. Error classes 3 and 4 mean severe errors.

The function ModifyRecord can be implemented similarly. All of the example routines, suitably modified for network use, are provided in NETEXAMP.PAS.

G. Questions and Answers

This section discusses some of the most common application design questions that arise for B-Tree Filer.

Save Mode or Normal Mode?

When a fileblock is opened with `BTOpenFileBlock` a boolean parameter specifies whether "save mode" is to be used. In save mode, data integrity is increased by writing portions of the fileblock to be modified to the dialog file before the changes are made. Then, if an error occurs, the fileblock can be restored to its known state prior to the operation. If no error occurs, all new data is flushed to the disk before the backup information is removed from the dialog file. If the system crashes before an operation is complete, B-Tree Filer automatically repairs the fileblock by using the information in the dialog file the next time the fileblock is opened.

As you can imagine, save mode reduces performance when adding, deleting, or modifying information in the data and index files. Therefore you should consider the following factors before choosing to use save mode.

- You can call `BTFlushFileBlock` at specific times to guarantee that all information is flushed from memory onto the disk storage device.
- The routines in the REINDEX unit will reconstruct a corrupted index file from scratch. Reconstruction of 5000 records with four keys per record takes 5 to 20 minutes.
- B-Tree Filer maintains an internal flag that indicates when a fileblock wasn't correctly closed after being modified (see `BTForceWritingMark` in Chapter 5). When an application attempts to open such a fileblock, an error is generated and the application can trigger an automatic rebuild.
- The performance loss due to save mode is usually reduced on a network because of smart caching on the file server. (However, see procedure `BTForceNetBufferWriteThrough` in Chapter 5.)
- On Novell networks, B-Tree Filer can work with the transaction tracking services (TTS) of the operating system. This provides another approach to data integrity. See [_8.A.7](#) for more information on TTS.

The decision between save mode and normal mode for a single user application is usually made by comparing the day to day performance loss for save mode against the time to reconstruct the database if a severe error does occur. For a network application, the data sets are usually larger and more critical, leading to higher costs if the data is lost for any period of time. As a result, save mode is a good idea for network applications when Novell's TTS is not available. If save mode seems too slow, use normal mode and provide for a regular backup (perhaps every few hours) of the files. Some networks even support automatic backup to streaming tape so that data saving can take place in the background without disrupting the system.

Another point to consider is that save mode applies to one atomic operation only, in other words a single `BTAddKey`, a single `BTAddRec`, and so on. In a great number of applications adding a record usually results in adding several keys, or on a higher level for example adding an order involves adding several order lines. Would save mode make sense in this kind of database environment? For example, would you gain anything from being able to recover 6 of the 10 lines of an order? You presumably would prefer having none of the order present or all of it, in which case save mode does not gain you anything.

The final choice between save mode and normal mode will be based on a combination of measurable and subjective factors. Are updates done in batch or interactive mode? How does it "feel" to add and delete records interactively under heavy network load? How disruptive is a rebuild if a crash occurs? What backup strategy is already in place for the network? Fortunately, just by toggling one boolean parameter to `BTOpenFileBlock`, you can easily experiment with your application.

Network Emulation

There are three ways to control how a fileblock is opened for network use:

1. Define `NoNet` or one of the true network interfaces in `BTDEFINE.INC`.
2. Pass `NoNet` or one of the true network types to `BTInitIsam`.
3. Pass `False` or `True` for the `Net` parameter when opening a fileblock with `BTOpenFileBlock`.

If `BTDEFINE.INC` is set for `NoNet`, then no fileblock can be opened for network use and no network emulation is available. If `BTDEFINE.INC` defines one or more network interfaces and `NoNet` is passed to `BTInitIsam`, then B-Tree Filer enters "network emulation mode." The behavior of a fileblock opened with `Net` set to `True` differs depending on whether network emulation is active or not. The following table summarizes the differences:

Emulation	NoNet compilation	Network
Handles per fileblock	2	3
Write accesses required	locks not required	fileblock lock
Probability code will work correctly on a true network	medium	high
Execution speed compared higher to a true network	much higher	noticeably
Effect of read locks on faster speed of a series of read accesses	no effect	>2x

See the documentation for `BTReadLockFileBlock` in Chapter 5 for more information about the last point.

5. B-Tree Filer Identifiers

The FILER unit is the main unit of B-Tree Filer. It contains all of the fileblock management functions--many of which were used in the examples in Chapter 4. These include functions that manipulate fileblocks, add and delete records and keys, and control fileblock and record access.

This chapter describes the identifiers interfaced by the FILER unit. The functions and procedures are arranged alphabetically.

Some interfaced identifiers are not described here. Those are intended for the internal use of the unit, and are interfaced for communication with the other units provided with B-Tree Filer. See the source code for details.

All routines described in this chapter work with single user as well as network fileblocks. A few prerequisites must be met before they will work correctly in a multi-user, network environment:

1. A network directive must be chosen at compile time to select a network interface. Details can be found in `_2.A`.
2. When the procedure `BTInitIsam` is called to initialize the FILER unit, the parameter `ExpectedNet` must have a value other than `NoNet`. See `BTInitIsam` later in this section for further details.
3. Finally, network fileblocks must be opened by calling `BTOpenFileBlock` with the `Net` parameter set to `True`. New fileblocks are created by calling `BTCreateFileBlock`, then opened with `BTOpenFileBlock` in the same fashion.

Because the routines described in this chapter aren't really standalone routines--they often interact with each other closely--it isn't always possible to provide a meaningful short example. If no example is provided for a routine, look at `_4.C` and `_4.F` where the programming examples show many of these functions used together.

Not all examples contain the retry loops required for locking errors on a network. This is done for clarity and to prevent the manual from eating too many more trees. General recommendations for dealing with locking errors can be found in `_4.F`.

Note that all user-configurable constants of the FILER unit are contained in the source file `FILER.CFG`. Edit this file if you want to change compile-time constants.

Declarations

Constants

```
AddNullKeys; : Boolean = True;
```

This typed constant affects the behavior of only the `REINDEX`, `REORG`, `VREORG`, `REBUILD`, and `VREBUILD` units. When `AddNullKeys` is `True`, all keys returned by the user-defined `BuildKey` routine--even empty strings--are added to the fileblock as it is reorganized. If `AddNullKeys` is set to `False`, these units won't add empty keys to a reorganized fileblock. This provides an extra degree of flexibility for Fileblocks that don't store keys for every index and record.

```
DatExtension; : String[3] = 'DAT';  
IxExtension; : String[3] = 'IX';  
DiaExtension; : String[3] = 'DIA';  
SavExtension; : String[3] = 'SAV';  
MsgExtension; : String[3] = 'MSG';
```

names passed to the B-Tree Filer routines use these standard extensions. These strings override any extensions that you specified when opening a fileblock. DatExtension is applied to the data file, IxExtension to the index file, DiaExtension to the dialog file, SavExtension to the copy of the data file that RestructFileBlock creates, and MsgExtension to the message file that the rebuild routines create when duplicate keys are detected.

```
IsamDelayBetwLocks; : Word = 64; {milliseconds}
```

In networks where the timeout value for locking attempts is emulated through retries, there is a pause of IsamDelayBetwLocks milliseconds between the retries. For further discussion please see _4.D.

```
IsamFBlockTimeOutFactor; : Word = 4;
```

IsamLockTimeOut*IsamFBlockTimeOutFactor determines the maximum number of milliseconds BLockFileBlock is allowed to execute a locking attempt. For further discussion please see _4.D.

```
IsamFileNameLen; = 64;
```

Maximum length of any single drive, directory, and filename for a fileblock. Note that this value is less than the DOS limit of 79 characters. You can change IsamFileNameLen without rebuilding fileblocks. Three arrays of size IsamFileNameLen+1 are contained within each variable of type IsamFileBlock.

```
IsamFlushDOS33; : Boolean = True;
```

Affects the behavior of the low level IsamFlush procedure, which is called whenever B-Tree Filer needs to assure that the contents of DOS file buffers are flushed to disk. Flushing occurs whenever the application calls BTFlushFileBlock or BTFlushAllFileBlocks. It also occurs when the application has enabled "write-through" by calling BTForceNetBufferWriteThrough with a parameter of True (see that procedure for details).

The IsamFlush procedure uses any of three mechanisms to flush a file:

1. If IsamFlushDOS33 is True and the DOS version is at least 3.3, IsamFlush calls DOS function \$68, which is preferred over alternative methods because it never fails for lack of a free file handle and it doesn't disturb any locks that may be in place. Unfortunately, when running under PC-MOS, function \$68 is accepted with no error, but no actual flush occurs. This is the only known case of function \$68 problems, but other lesser networks may have the same behavior; hence the typed constant IsamFlushDOS33 which you can use to disable this method of flushing. If flush method 1 is not used, or if it fails, flush method 2 is tried.
2. IsamFlush duplicates the file handle to be flushed (by calling DOS function \$45) and then closes the duplicate handle. This sequence fails if no free file handle is available. Unfortunately, certain networks, including all SHARE-based LANs, remove locks when closing the duplicated handle. Other networks, including Novell and CBIS, leave the locks in place. If flush method 2 fails for lack of a file handle, and if the file is not part of a network fileblock, IsamFlush moves on to method 3.
3. IsamFlush closes the handle and reopens it. This technique is not even attempted for a network file because all networks, with the exception of Novell, remove locks from the closed file. If none of the methods work, IsamFlush returns with IsamOk False and an error code of 10150 in IsamError.

You will conclude that, under many circumstances, there are severe problems with flushing locked fileblocks. If your target environment falls into this category, be sure not to call FlushFileBlock or FlushAllFileBlocks while locks are in place on the data file, and be sure not to enable write-through.

A file named FLUSHTST.TXT, found in your B-Tree Filer BONUS directory, contains some test code for determining how your network behaves with regard to flush methods and locking.

```
IsamLockTimeOut; : Word = 768; {milliseconds}
```

Determines the maximum in milliseconds for a locking attempt by BTRedLockFileBlock and BTLockRec. For further discussion please see _4.D.


```
MaxHeight; = 8;
```

Specifies the deepest level of the B-tree. When BTInitIsam is called, free heap space must exist for at least MaxHeight index pages. (The heap space can be allocated from EMS memory in real mode if EMS is available and B-Tree Filer is configured correctly. See 2.B and the BTInitIsam procedure in this chapter.) MaxHeight and CreatePageSize together determine the maximum number of keys that can be added to any index. In general you should not modify MaxHeight. If you need to do so, keep the following rules in mind:

1. The smallest acceptable value for MaxHeight in B-Tree Filer is 4, regardless of how few keys will be added to the fileblock. Worst case page balancing and splitting during a call to BTAddKey requires 4 index pages to be in memory at once.
2. The B-tree manager will not detect an error if you exceed the maximum number of keys allowed by MaxHeight and CreatePageSize. Due to the dynamic nature of page balancing, error detection would require unacceptable performance overhead. It is your responsibility to choose values that work with your application.
3. The algorithm for determining the capacity of a B-tree can be summarized as follows, where Cap(i) is the capacity of level i of the tree:

```
Cap(1) = 1
Cap(2) = CreatePageSize
Cap(i) = Cap(i-1)*(1+CreatePageSize div 2)    (i >= 3)
```

and the total capacity is the sum of the capacities for each level.

For the default value of CreatePageSize=62, the following total capacities are computed:

MaxHeight	Maximum Keys	
4	65,535	
5	2,097,151	
6	67,108,863	
7	2,147,483,647	(= 2 ³¹ = MaxLongInt)
8	68,719,476,735	(> 2 ³²)

As a result, increasing the default value of 8 is meaningless, while reducing the value has to be done with the utmost caution and understanding. A rebuild of the index files after changing this constant is not required.

```
MaxKeyLen; = 30;
```

MaxKeyLen determines the maximum allowed length of a key in any fileblock. Legal values are between 1 and 255. Note that MaxKeyLen does not affect the size of the index file (whose size is determined by key lengths specified in an IsamIndDescr variable when the fileblock is created), but that MaxKeyLen does affect the size of an index page in memory and therefore the number of index pages that will fit into a given amount of heap space.

```
MaxNrOfKeys; = 100;
```

MaxNrOfKeys defines the maximum number of indexes in any one fileblock. Legal values are between 1 and 254. Increasing this constant proportionally increases the amount of storage needed for a variable of type IsamIndDescr but has no effect on index file size.

```
MaxNrOfWorkStations; : Word = 50;
```

MaxNrOfWorkStations contains the maximum number of workstations that can open a fileblock when used on a network. The default value of 50 will suffice for typical network installations. Legal values range from 1 to 65534, but unnecessarily increasing the value will increase the access time for all network operations.

The value of MaxNrOfWorkStations is only used when opening a fileblock, and when the dialog file needs to be rebuilt (either it was deleted or you are repairing it in Save mode). If the fileblock's dialog file is valid, the maximum number of workstations that can open the fileblock is taken from the dialog file itself.

```
MinimizeUseOfNormalHeap; = $40000000;
```

When this value is added to the Free parameter passed to BTInitIsam, B-Tree Filer will use the least possible normal heap space for page buffers. See BTInitIsam for further information. This constant should not be changed.

```
CreatePageSize; = 62;
MaxPageSize; = 62;
```

Specifies the maximum number of keys in a B-tree page. MaxPageSize is the value for the application as a whole, and CreatePageSize is the value for new fileblocks created by the application. A B-Tree Filer application can open any fileblock that was created with a value of CreatePageSize that is less than or equal to MaxPageSize.

The minimum amount of heap space required when calling BTInitIsam is directly related to MaxPageSize and MaxHeight. The results of many test runs indicate that the default value of 62 is optimal. It is strongly recommended that you do not change this value, but if you must, note the following restrictions:

1. Both values must be an even number between 14 and 246.
2. If you reduce MaxPageSize, you must rebuild all index files that were created with a value for CreatePageSize that is greater than the new value of MaxPageSize.
3. CreatePageSize can never be greater than MaxPageSize.
4. The best values are even numbers just below a power of two, i.e., 14, 30, 62, 126. The only desirable alternative to 62 is 30, whose data retrieval performance in a network environment is comparable to that of 62.

```
SearchForSequentialDefault; : Boolean = True;
```

Default state of the special searching method applied to BTNextKey and BTPrevKey operations. This method is useful in recovering from "sequential access not allowed" errors that arise when multiple workstations can modify a fileblock. The default value can also be changed for a single index with the procedure BTSetSearchForSequential.

```
VersionStr; = '05.50';
```

Defines the current version number of Filer where it can be accessed by an application program.

Types

```
IsamFile; = record
    Handle : Word;
    Name   : array[0..IsamFileNameLen] of Char;
end;
```

Type describing a single open file. This is used internally by B-Tree Filer and passed to lower level routines such as IsamAssign, IsamRewrite, and IsamFlush. A variable of type IsamFileBlock contains three IsamFile fields: one each for data file, index file, and dialog file. Handle is a valid handle number assigned by DOS, or \$FFFF when the file is closed. Name is the null-terminated pathname of the file.

```
IsamFileBlockName; = String[192];
```

All filenames passed to BTCreateFileBlock, BTOpenFileBlock, RestructFileBlock, ReIndexFileBlock, Rebuild(V)FileBlock, Reorg(V)FileBlock and FixToVarFileBlock must of this type. Variables of this type must contain one, two, or three valid DOS filenames when passed to one of the above routines. Drive name and path are optional. The current directory is used as a starting point if a path doesn't begin with a '\'. No extensions should be given. Extensions are automatically appended, using the constants defined above.

If multiple filenames are given, they should be separated by a semicolon (;).

The first filename determines the drive, directory, and name of the data file and also of the dialog file for a network or save-mode fileblock. The second filename specifies the name and location of the index file. The third filename is only useful for the procedures RestructFileBlock, Rebuild(V)FileBlock, Reorg(V)FileBlock, and FixToVarFileBlock, where it specifies the name and location of the saved copy of the data file. The third name may be passed to the other routines (e.g. BTOpenFileBlock), but it will be ignored there. Examples of valid names are:

```

'ADDRESS'
Data file      'ADDRESS.DAT'
Index file     'ADDRESS.IX'
'C:\FILES\ADDRESS'
Data file      'C:\FILES\ADDRESS.DAT'
Index file     'C:\FILES\ADDRESS.IX'
'C:\FILES\ADDRESS;D:\INDEX\ADDRESS'
Data file      'C:\FILES\ADDRESS.DAT'
Index file     'D:\INDEX\ADDRESS.IX'

```

The following naming convention is not recommended, even though B-Tree Filer will accept it:

```

'C:\FILES\ADDRESS;D:\INDEX\ADINDEX'
Data file      'C:\FILES\ADDRESS.DAT'
Index file     'D:\INDEX\ADINDEX.IX'

```

In this case the logical connection between the data file and the index file is no longer apparent.

```
IsamFileBlockPtr; = ^IsamFileBlock;
```

A variable of this type points to information describing each open fileblock. An IsamFileBlockPtr is allocated and initialized by a call to BTreeOpenFileBlock. All further operations on the fileblock are performed by passing this variable to a B-Tree Filer routine. You shouldn't be concerned with the contents of an IsamFileBlock record, except to note that an opened fileblock consumes about 300-500 bytes of heap space depending on the number of indexes and whether it incorporates network support. See BTreeOpenFileBlock for more information.

```
IsamFileName; = String[IsamFileNameLen];
```

A string defining the drive, directory, and name of a single file. Parameters of this type are passed to lower level B-Tree Filer routines such as IsamExists and IsamAssign.

```

IsamIndDescr; = array[1..MaxNrOfKeys] of
    record
        KeyL : 1..MaxKeyLen;
        AllowDupK : Boolean;
    end;

```

A variable of this type is passed to BTreeCreateFileBlock to describe the indexes associated with the fileblock. IsamIndDescr describes the length and type for each index. Key length can be from 1 to MaxKeyLen and is stored in the KeyL field. Keys can be of two different types, depending on whether or not duplicate keys may occur:

1. Primary Keys: AllowDupK = False. An example is a customer number that is guaranteed to be unique. Zero, one, or more primary keys can be defined for a fileblock. Remember that BTreeAddKey will not accept duplicate primary keys.
2. Secondary Keys: AllowDupK = True. Examples include last name and zip code, which are not guaranteed to be unique. BTreeAddKey will successfully add duplicate secondary keys as long as the data reference number for two such keys does differ.

```
IsamKeyStr; = String[MaxKeyLen];
```

All key strings used to perform index operations must of this type.

```
NetSupportType; = (NoNet, Novell, MsNet);
```

This type enumerates the various networks that B-Tree Filer supports. The procedure BTreeInitIsam requires a parameter of this type to specify the current network environment. Upon successful initialization of the network, the function BTreeNetSupported returns the same value.

```

ProcBTreeCharConvert; = procedure(DataPtr : Pointer; DataLen :
    LongInt;
                                PostRead : Boolean; HookPtr :
    Pointer);

```

This procedural type defines a record conversion routine that is called when a record is read and when it is written. The idea behind this type of routine is to enable you to have two different formats for your records: an external format that resides in the fileblock, and an internal format that you use in your application. Examples are support for different code pages between the fileblock and the application, for string encryption, and for record compression. DataPtr is a pointer to the record that was just read or about to be written, DataLen is its length. PostRead is True if the record has just been read from the fileblock (in other words, the routine is being called to convert the record from an external format to your internal one), and is False if the record is about to be written (in other words the record needs to be converted to the external format). HookPtr is a pointer to any user-defined data structure; it is a way of passing other information to this routine. For further information on code page support in particular, see the CCSKEYS bonus unit.

Variables

```
IsamCompiledNets; : Set of NetSupportType;
```

The initialization block of the FILER unit assigns this variable its value. The set contains all network types activated within BTDEFINE.INC which are therefore valid options to pass to BTInitIsam. This variable should be considered read-only and should not be changed.

```
IsamDOSError; : Word;
```

This variable contains a non-zero value if there was a DOS error during a call to a B-Tree Filer routine. The value is the one returned by DOS in the AX register.

```
IsamDOSFunc; : Word;
```

This variable contains the function code for the last DOS function called by B-Tree Filer. (The function code is the value passed in the AX register for the DOS int \$21 call.) If an error occurred while making the DOS function call, the value of IsamDOSFunc retains its value all the way to the end of the B-Tree Filer routine. Hence, IsamDOSFunc can be used to determine exactly which DOS function produced a particular IsamError. IsamDOSFunc has a value of 0 if the routine made no calls to a DOS function.

```
IsamError; : Integer;
```

This variable is set by every B-Tree Filer routine. If IsamOK is False, IsamError corresponds to the type of error that occurred. A complete list of error codes is given in Appendix A.

```
IsamOK; : Boolean;
```

This variable is set by every B-Tree Filer routine. Upon entry to each routine, IsamOK is initialized to True, even if a previous error was pending. If IsamOK remains True when the routine returns, the requested operation was successfully completed. Otherwise, the variable IsamError contains a code corresponding to the error that occurred. IsamError should never equal zero when IsamOK is False. IsamOK should be checked after each call to a B-Tree Filer routine.

```
IsamReXUserProcPtr; : Pointer;
```

The FILER unit sets this variable to Nil within its initialization block. When an application sets IsamReXUserProcPtr to a non-Nil value, the pointer must contain the address of a procedure that is called by RestructFileBlock, ReIndexFileBlock, Rebuild(V)FileBlock, Reorg(V)FileBlock, and FixToVarFileBlock. The procedure is called for each record and key that is processed. It can be used to display status information, compute statistics, or allow the user to abort the rebuild. The routine must be declared far, must be global, and must match the following prototype declaration:

```
{F+} {Also must be global}
procedure UserStatusRoutine(KeyNr : Integer; NumRecsRead : LongInt;
                             NumRecsWritten : LongInt; var Data; Len
: Word);
begin
  {display status...}
end;
{F-}
```

KeyNr is the index number currently being added. NumRecsRead is the number of records read up to this point and NumRecsWritten is the number of records written. Data is the data record being worked on, and Len is the number of bytes of data in the record. The rebuild routines all work by first copying each valid data record to a new data file. During this pass, UserStatusRoutine is called once for each record with KeyNr set to zero. After the data is copied, all keys for index number one (KeyNr=1) are added and UserStatusRoutine is called for each key, then all keys for index number two, and so on. See Chapter 6 for more information.

Procedures and Functions

Almost all of the following routines require the parameter IFBPtr of type IsamFileBlockPtr. Its use is to differentiate between the different fileblocks opened by BTOpenFileBlock, just as you would pass a different file variable to a Turbo Pascal Read or Write routine. BTOpenFileBlock is the only routine that allocates and initializes a variable of type IsamFileBlockPtr. BTCloseFileBlock deallocates the fileblock variable and sets the IFBPtr to Nil.

The parameter IFBPtr will not be explained again in the descriptions that follow.

Some interfaced procedures and functions of the FILER unit are not described in detail here. These routines are interfaced primarily for the use of other B-Tree Filer units such as REORG and VREORG. If you are writing similar utility units for B-Tree Filer files, you may find the following routines useful. Note that these routines return status information in IsamOK and IsamError just like all other FILER routines. See the source code for more information.

```

procedure IsamAssign;(var F : IsamFile; FName : IsamFileName);
  {-Assigns FName to the file <F>}

procedure IsamBlockRead;(var F : IsamFile; var Dest; Len : Word);
  {-Reads a block of data with Len bytes out of the file F to Dest}

procedure IsamBlockReadRetLen;(var F : IsamFile; var Dest; Len :
Word;
                               var BytesRead : Word);
  {-Reads a block of data and returns the number of bytes read}

procedure IsamBlockWrite;(var F : IsamFile; var Source; Len : Word);
  {-Writes a block of data with Len bytes to the file F from Source}

procedure IsamClearOK;;
  {-Resets all status variables, even internal ones}

procedure IsamClose;(var F : IsamFile);
  {-Closes the file F}

procedure IsamCopyFile;(Source, Dest : IsamFileBlockName;
                        DeleteSourceAfterCopy : Boolean);
  {-Copies file Source to Dest}

procedure IsamDelay;(MilliSecs : LongInt);
  {-Delays for MilliSecs ms using a DOS call (resolution 55ms)}

procedure IsamDelete;(var F : IsamFile);
  {-Deletes the file F}

function IsamExists;(Name : IsamFileName) : Boolean;
  {-Returns True if the specified file exists}

procedure IsamExtractFileNames;(FNameComp : IsamFileBlockName;
                                var FNameD, FNameI :
IsamFileBlockName);
  {-Separates two file names delimited by ";" }

procedure IsamFlush;(var F : IsamFile; var WithDUP : Boolean; NetUsed
: Boolean);
  {-Flushes the buffers of F to disk, depending on the constant
   IsamFlushDOS33 and the parameter NetUsed}

```

```

function IsamForceExtension;(Name, Ext : IsamFileName) :
IsamFileName;
    {-Forces the extension Ext onto the file name Name}
procedure IsamGetBlock;(var F : IsamFile; Ref, Len : LongInt; var
Dest);
    {-Reads a block of data with Len bytes starting at position Ref
    from the file F to Dest}
procedure IsamLongSeek;(var F : IsamFile; Ref : LongInt);
    {-Seeks to the position Ref in file F}
procedure IsamLongSeekEOF;(var F : IsamFile; var Len : LongInt);
    {-Seeks to the end of file in file F and returns its length}
procedure IsamPutBlock;(var F : IsamFile; Ref, Len : LongInt; var
Source);
    {-Writes a block of data from Source with Len bytes to the file F
    starting at position Ref}
procedure IsamRename;(var F : IsamFile; FName : IsamFileName);
    {-Renames the unopened file F to FName}
procedure IsamReset;(var F : IsamFile; NetUsed, ReadOnly : Boolean);
    {-Opens the file F in the specified mode}
procedure IsamRewrite;(var F : IsamFile);
    {-Creates the file F}

```

All normal procedures and functions in the FILER unit, with the exception of BTIsamErrorClass, reset the FILER unit's status variables (IsamOK and IsamError) as soon as they are called. In some circumstances it may be desirable to retain the previous operation's error codes while making another FILER procedure call. The following routines are available to meet this need. They work just like the corresponding routine without "Peek" in the name, except that they preserve the values of IsamOK, IsamError, IsamDOSError, and IsamDOSFunc. However, if another error occurs while in the "Peek" routine itself, that error is returned instead of any previous error. The "Peek" routines are:

```

BTPeekaRecIsLocked
BTPeekDataFileName
BTPeekDatRecordSize
BTPeekFileBlockIsLocked
BTPeekFileBlockIsOpen
BTPeekFileBlockIsReadLocked
BTPeekGetRecordInfo
BTPeekIndexFileName
BTPeekIsNetFileBlock
BTPeekKeyRecordSize
BTPeekMinimumDatKeys
BTPeekNetSupported
BTPeekNoNetCompiled
BTPeekNrOfKeys
BTPeekRecIsLocked

```

BTAddKey

Syntax

```
procedure BTAddKey; (IFBPtr : IsamFileBlockPtr; Key : Word; UserDatRef  
: LongInt;  
UserKey : IsamKeyStr);
```

Purpose

Add a key.

Description

After a new data record is added with BTAddRec, the index file should also be updated. This is done by passing the key string UserKey and the data record number UserDatRef obtained from the BTAddRec call to BTAddKey. BTAddKey must be called one or more times for every index Key, which ranges from 1 to the number of indexes in the fileblock.

BTAddKey modifies the internal sequential pointer whether or not the BTAddKey call succeeds. Hence, the sequential pointer must be refreshed before calling BTNextKey or BTPrevKey. The pointer is refreshed automatically if the typed constant SearchForSequentialDefault was True when the fileblock was opened or if BTSetSearchForSequential was used to activate the option thereafter. Otherwise, the pointer can be reinitialized by a call to BTClearKey, BTSearchKey, BTFindKey, BTFindKeyAndRef, BTSearchKeyAndRef, BTNextDiffKey, or BTPrevDiffKey.

BTAddKey can be used with a network fileblock only if the fileblock is locked for exclusive write access (level 1 or 2).

If Key denotes a primary index, UserKey must specify a string that does not currently exist in the index. If Key is a secondary index, the combination of UserKey and UserDatRef must still be unique. BTAddKey returns with error 10230 if the key is not unique.

Note that all keys must take the form of a string. The supplied NUMKEYS unit provides routines to convert all Turbo Pascal numeric types to and from strings, while maintaining correct sort order and minimizing the length of the string. NUMKEYS also provides functions to pack ASCII strings, creating keys that are as much as 50% shorter. It also provides a routine to invert a string, so that an index will sort in descending order. See _6.G for more information.

Example

See _4.C and for an example.

See Also

BTAddRec
BTSetSearchForSequential

BTDeleteKey

BTAddRec / BTaRecIsLocked

Syntax

```
procedure BTAddRec; (IFBPtr : IsamFileBlockPtr; var RefNr : LongInt;  
var Source);
```

Purpose

Add a data record.

Description

A data record is added to the data file with BTAddRec. Source is written to the end of the data file if it has no deleted entries, otherwise a record previously deleted with BTDeleteRec is overwritten and reused. RefNr returns the data reference number for the record after a successful return from BTAddRec. This long integer is used to identify the record for further access. RefNr will usually be passed in the UserDatRef parameter to BTAddKey soon after the record is added.

Since Source is an untyped parameter, any desired data structure can be passed to BTAddRec. Although this is necessary to allow BTAddRec to work for all data types, it transfers the responsibility to the programmer to pass a variable of the correct type. BTAddRec writes the number of bytes specified in DatSLen when BTCreateFileBlock was called.

If the first four byte field of the data record is reserved for a deletion tag, be sure to initialize it to zero before calling BTAddRec.

BTAddRec can be used with a network fileblock only if the fileblock is locked for exclusive write access (level 1 or 2).

Example

See _4.C for an example.

See Also

BTAddKey
BTGetRec

BTDeleteRec

Syntax

```
function BTaRecIsLocked; (IFBPtr : IsamFileBlockPtr) : Boolean;
```

Purpose

Return True if any record is locked.

Description

B-Tree Filer maintains an internal list of the records locked by each workstation.

BTaRecIsLocked returns True if the current workstation has locked one or more records. Note that this routine does not indicate whether another workstation has locked any records.

Example

```
var  
  MyFileBlockPtr : IsamFileBlockPtr;  
...  
if (BTaRecIsLocked(MyFileBlockPtr)) then  
begin  
  BTUnlockAllRecs(MyFileBlockPtr);  
  if not IsamOk then  
  begin  
    {Error handling}  
  end;  
end;
```

If any record lock placed by the current workstation exists on the fileblock MyFileBlockPtr, all record locks are removed.

See Also

BTLockRec
BTUnlockAllRecs

BTAddRec / BTaRecIsLocked

BTRecIsLocked

Syntax

```
procedure BTClearKey; (IFBPtr : IsamFileBlockPtr; Key : Word);
```

Reset the

Description

BTClearKey

BTC1e

```

if not IsamOK then begin
    {Error handling}
end;
BTNextKey(PF, KeyNr, RefNr, KeyStr);
...

```

Social Media

BTNextK

proce

Close all

Definition

All open file

information can be found in the description of `BTCloseFileBlock`.

Unlike `BTCloseFileBlock`, `BTCloseAllFileBlocks` cannot set each user variable of type

IsamFileBlockPtr to Nil when the fileblock is closed. If an application repeatedly opens fileblocks after closing them with `BTCloseAllFileBlocks`, it is a good idea to set the fileblock variables to Nil after closing them.

BTC1c

```

    if not IsamOK then begin
        {Error handling}
    end;
    BTEExitIsam;
    if not IsamOK then begin
        {Error handling}
    end;

```

3.1.1.1. *Al*

BTClose

BTCloseFileBlock

Syntax

```
procedure BTCloseFileBlock; (var IFBPtr : IsamFileBlockPtr);
```

Purpose

Close the specified fileblock.

Description

The fileblock is closed and the memory allocated by BOpenFileBlock is freed. IFBPtr is set to Nil after a successful call to BTCloseFileBlock.

All locks--write locks, read locks, record locks--placed by the calling workstation are automatically removed. Error 10322 indicates that an attempt to remove an existing fileblock lock or read lock failed. Error 10323 indicates that an attempt to remove an existing record lock failed.

Any data or index information that remains in B-Tree Filer or DOS memory buffers is flushed to disk during this call. An appropriate error code is returned if the flush fails. Only after a successful flush are the files that correspond to the fileblock closed. If an error occurs while closing a file, error code 10160 is returned. Even so, memory used by the fileblock is freed and BTCloseFileBlock can not be called again for this fileblock.

The FILER unit implements a Turbo Pascal exit procedure that automatically closes all open fileblocks when the program halts, either successfully or with a runtime error. However, if a program crashes while fileblocks are still open and index writes were performed, or if the call to BTCloseFileBlock fails within the exit procedure, it is likely that the index file will need to be rebuilt before the fileblock can be successfully opened again.

Example

See _4.C for an example.

See Also

BTCloseAllFileBlocks
BTOpenFileBlock

BTCreateFileBlock

BTCreatFileBlock

Syntax

```
procedure BTCreatFileBlock (FName : IsamFileBlockName; DatSLen :  
    LongInt;  
                                NumberOfKeys : Word; IID : IsamIndDescr);
```

Purpose

Create a new fileblock.

Description

BTCreatFileBlock creates new data and index files. FName must contain either one, two, or three valid DOS filenames, separated by semicolons, with optional drive and pathname specifications. No extensions should be specified since these are automatically appended using DatExtension (default='DAT') for the data file and IxExtension (default='IX') for the index file. See type IsamFileBlockName for more information about how to specify FName.

Existing files of the same name are overwritten without warning (although this may fail if another workstation has such files open in a non-shareable mode).

The parameter DatSLen specifies the length of each data record. (For variable length records, DatSLen specifies the section length. See _6.B for more information.) DatSLen must be in the range 21 to MaxLongInt bytes (although the practical upper limit is 65520 bytes). Error 10020 indicates an invalid record length.

In most cases, a data record definition should begin with a four byte (LongInt) field reserved for a deleted record tag. This field will contain four zero bytes for valid records, and a non-zero value for deleted records. Several B-Tree Filer units require this field: automatic fileblock rebuilding (RESTRUCT), variable length records (VREC), and fileblock browsing (BROWSER, OPBROW, TVBROWS, and WBROWSER).

A data record is handled as a typeless parameter. B-Tree Filer maintains no internal knowledge of the fields within each record, only its length. NumberOfKeys specifies the number of indexes for this fileblock. Valid values range from 0 to MaxNrOfKeys, inclusive. If NumberOfKeys is 0, no index file is created.

The parameter IID determines the length of each key and whether its associated index allows duplicate key strings. Typically, the IID parameter should be declared as a local variable and initialized just prior to calling BTCreatFileBlock. For more details, see the programming example in _4.C and the description of type IsamIndDescr near the beginning of this chapter.

BTCreatFileBlock temporarily allocates heap space (up to about 500 bytes) for internal data structures. It returns error 10030 or 10090 if insufficient memory is available.

The resulting data file is one record long. The first 21 bytes of this record contain B-Tree Filer system information matching the following template:

```
SystemRecord =  
    record  
        FirstFree : LongInt; {Ref number of first deleted record, -1  
if none}  
        NumberFree : LongInt; {Number of deleted records}  
        NumRec : LongInt; {Total number of records}  
        LenRec : LongInt; {Bytes in one record}  
        NrOfKeys : LongInt; {Number of indexes for the fileblock}  
        Changed : Boolean; {True if index or data changed but not  
flushed}  
    end;
```

You should not modify the contents of this system record.

The resulting index file contains space for CreatePageSize keys of each index. The corresponding number of bytes can be obtained by calling BTKeyRecordSize after the fileblock is open. Don't be surprised if you have an index file of several thousand bytes even though you haven't added any keys yet.

BTCreatFileBlock

BTCreatFileBlock does not initialize a variable of type IsamFileBlockPtr and the files it creates are left closed after the call. To use the fileblock, you must call BTOpenFileBlock.

See Also

BTDeleteFileBlock

BTOpenFileBlock

BTDataFileName / BTDatRecordSize

Syntax

~~function BTDataFileName; (IFBPtr : IsamFileBlockPtr) : IsamFileName;~~

Purpose

Return the name of the fileblock's data file.

Description

This function returns the data file name, including extension, of the specified fileblock. If the name passed to BTOpenFileBlock did not include a drive and directory, neither does the string returned by BTDataFileName.

See Also

BTIndexFileName

Syntax

function BTDatRecordSize; (IFBPtr : IsamFileBlockPtr) : LongInt;

Purpose

Return the size of a data record.

Description

This function returns the value of the parameter DatSLen that was passed to BTCreateFileBlock.

If BTFreeRecs returns zero, BTDatRecordSize returns the number of bytes of disk space that BTAddRec will allocate at the next call.

Example

```
if BTFreeRecs(PF) = 0 then
  if IsamOK then
    Writeln('The next call to BTAddRec will increase data file size
  by '
          BTDatRecordSize(PF), ' bytes')
  else begin
    {Error handling}
  end;
```

Displays the increase in data file size when a record is added, if no deleted record slots can be reused.

See Also

BTKeyRecordSize

BTMinimumDatKeys

BTDeleteAllKeys

Syntax

```
procedure BTDeleteAllKeys; (IFBPtr : IsamFileBlockPtr; Key : Word);
```

Purpose

Delete all keys in an index.

Description

This procedure provides support for temporary keys. If a program needs to create a new index for a specific key number at runtime, this key can then quickly and easily be deleted with BTDeleteAllKeys. Note that the index "slot" for the temporary key must be reserved when the fileblock is first created. (See _6.C for an alternate approach.)

BTDeleteAllKeys marks the associated index space available for reuse but does not actually write over the existing keys.

BTDeleteAllKeys can be used with a network fileblock only if the fileblock is locked for exclusive write access (level 1 or 2).

Example

In a fileblock with five indexes, assume that the fifth index (a secondary key) is used for generation of special lists and that when a data record is added, the fifth key is not added or is added with an empty string. When a special list needs to be created, BTDeleteAllKeys is used as follows:

```
var
  Ref      : LongInt;
  IKS      : IsamKeyStr;
  PTemp    : PersonDef;
...
{Clear prior temporary keys}
BTDeleteAllKeys(PF, 5);
{Prepare to add new special keys}
if IsamOK then BTClearKey(PF, 1);
if IsamOK then BTNextKey(PF, 1, Ref, IKS);
{Scan all records}
while IsamOK do begin
  BTGetRec(PF, Ref, PTemp, False);
  {Add special keys}
  if IsamOK then BTAddKey(PF, 5, Ref, BuildSpecKey(PTemp));
  if IsamOK then BTNextKey(PF, 1, Ref, IKS);
end;
if IsamError <> 10250 then
  {Error handling} ;
...
```

All data records are scanned using the first key and a special key is created and entered for each one. If the while loop ends with any error but 10250 (no larger key found), a real error must have occurred.

See Also

BTDeleteKey

BTDeleteFileBlock

Syntax

```
procedure BTDeleteFileBlock(FName : IsamFileBlockName);
```

Purpose

Delete a fileblock.

Description

This routine deletes all DOS files that correspond to the fileblock of name FName. The parameter FName can contain up to three semicolon-separated names, as is described by the type IsamFileBlockName. The third file name is always ignored by BTDeleteFileBlock.

The data ('DAT') and index ('IX') files of the fileblock are deleted, as well as the dialog ('DIA') file if it exists. Files with extensions 'MSG' and 'SAV', created by the procedure RebuildFileBlock among others, are never deleted.

If a fileblock variable (of type IsamFileBlockPtr) is associated with the given files, it should be closed by calling BTCloseFileBlock before BTDeleteFileBlock is called.

Example

```
BTDeleteFileBlock( 'C:\DATA\STUFF;D:\INDEX\STUFF' );  
if not IsamOK then begin  
  {Error handling}  
end;
```

The following files are deleted:

```
C:\DATA\STUFF.DAT  
D:\INDEX\STUFF.IX  
C:\DATA\STUFF.DIA (if it exists)
```

See Also

BTCreateFileBlock

BTDeleteKey

Syntax

```
procedure BTDeleteKey; (IFBPtr : IsamFileBlockPtr; Key : Word; _____  
UserDatRef : LongInt; UserKey : IsamKeyStr);
```

Purpose

Delete a key.

Description

When a data record is deleted with BTDeleteRec, the index file also needs to be updated. This occurs by calling BTDeleteKey for every index in the fileblock. Key is the index number (in the range 1 to BTNrOfKeys). UserKey is the key string and UserDatRef the associated data reference number, which is usually the parameter returned by one of the key search operations (e.g. BTFindKey). UserDatRef is ignored for a primary index, but must contain the correct value to delete a secondary key.

Error 10220 means that the key UserKey (in combination with UserDatRef for secondary indexes) was not found and therefore could not be deleted.

BTDeleteKey modifies the internal sequential pointer whether or not the BTDeleteKey call succeeds. Hence, the sequential pointer must be refreshed before calling BTNextKey or BTPrevKey. Although the automatic sequential searching option activated by BTSetSearchForSequential can sometimes recover from the modification, it is usually necessary to take additional precautions when a key is deleted. See the example below.

BTDeleteKey can be used with a network fileblock only if the fileblock is locked for exclusive write access (level 1 or 2).

Example

```
var  
  Ref, NextRef : LongInt;  
  IKS, NextIKS : IsamKeyStr;  
  OK           : Boolean;  
...  
BTClearKey(PF, 1);  
if IsamOK then BTNextKey(PF, 1, Ref, IKS);  
while IsamOK do begin  
  if WantToDeleteThisRecord then begin  
    {Find the next key now}  
    BTNextKey(PF, 1, NextRef, NextIKS);  
    OK := IsamOK;  
    {Delete record and associated keys}  
    BTDeleteKey(PF, 1, Ref, IKS);  
    if IsamOK then BTDeleteRec(PF, Ref);  
    {Reinitialize the sequential pointer}  
    if OK then BTFindKeyAndRef(PF, 1, NextRef, NextIKS);  
    IsamOK := IsamOK and OK;  
  end else  
    BTNextKey(PF, 1, Ref, IKS);  
end;
```

This example scans all keys in index 1 (assumed to allow duplicates) and decides whether to delete each associated record. If a record and its keys are to be deleted, a call to BTNextKey determines the next reference number and key string prior to deleting the key. After the record and key are deleted, a call to BTFindKeyAndRef repositions the sequential pointer over the next key.

See _4.C and _4.F for additional examples.

See Also

BTAddKey

BTDeleteRec

Syntax

```
procedure BTDeleteRec,(IFBPtr : IsamFileBlockPtr, RefNr : LongInt);
```

Purpose

Delete a data record.

Description

BTDeleteRec deletes a data record from the data file. The record is really just marked as free, so that a future BTAddRec can fill this "hole."

Only existing (non-deleted) data records can be deleted. Deleting an already deleted data record corrupts an internally maintained linked list of deleted records and therefore has unpredictable results. When a record is deleted, B-Tree Filer sets the first four bytes of the record to a non-zero value. If the first four bytes of each data record are reserved for this purpose and normally set to zero, deleted records can be recognized easily.

An attempt to delete record 0 generates error 10135. Record 0 contains internal system information and is never used as a normal data record. Error 10135 is also returned if you try to delete a record larger than the total number of records in the file, or a negative record number.

BTDeleteRec can be used with a network fileblock only if the fileblock is locked for exclusive write access (level 1 or 2).

Example

See _4.C and _4.F for examples.

See Also

BTAddRec

BTDeleteKey

BTExitIsam

Syntax

```
procedure BTExitIsam;;
```

Purpose

Finish working with B-Tree Filer.

Description

This procedure frees the memory allocated by BTInitIsam on the normal heap and in EMS memory. After calling BTExitIsam, all B-Tree Filer routines are prevented from working. (All calls will generate error 10455.) Only another call to BTInitIsam will allow further work with B-Tree Filer.

All fileblocks should be closed before calling BTExitIsam. If fileblocks remain open, BTExitIsam attempts to flush data and close the fileblocks before freeing the memory buffers. If an error occurs while closing a fileblock, BTExitIsam attempts to close the remaining open fileblocks. The returned error code does not provide any information as to which fileblock could not be closed, although it holds the first error encountered. It is important to check the variable IsamError on return from BTExitIsam.

If you don't call BTExitIsam explicitly, and BTInitIsam completed normally, BTExitIsam is executed automatically by an exit procedure when the application halts.

Example

```
var
  PBS : LongInt;
...
PBS := BTInitIsam(NoNet, 50000, 0);
if not IsamOK then begin
  {Error handling}
end;
...
BTExitIsam;
if not IsamOK then begin
  {Error handling}
end;
...
PBS := BTInitIsam(Novell, 50000+MinimizeUseOfNormalHeap, 120);
if not IsamOK then begin
  {Error handling}
end;
...
...
```

B-Tree Filer is first initialized with NoNet. The buffers use 50000 bytes from the heap; the EMS heap is not used. After some work, B-Tree Filer is exited with a call to BTExitIsam and the buffers are freed. After additional non-Filer activity, B-Tree Filer is initialized again. This time it is initialized for the Novell network. The normal heap is used to store required pages only if there are not enough pages for the buffers in the EMS Heap.

See Also

BTInitIsam

BTFileBlockIsLocked

Syntax

```
function BTFileBlockIsLocked, (IFBPtr : IsamFileBlockPtr) : Boolean;
```

Purpose

Check whether the fileblock is locked.

Description

This function determines whether the fileblock is locked for writing by the calling workstation. If so, True is returned, otherwise False.

This function returns False if B-Tree Filer was compiled with the compiler directive NoNet or if IFBPtr^ is a single user fileblock. If NoNet wasn't defined, i.e., a real network interface was activated, the present lock state of a network fileblock is returned. This is also the case when a network emulation mode is active (see _4.G).

Note that BTFileBlockIsLocked doesn't determine whether another workstation has locked the fileblock. That can be determined only by attempting to lock the fileblock and checking for success.

Example

```
var
  Locked : Boolean;
...
Locked := BTFileBlockIsLocked(PF);
if not Locked then begin
  BTLockFileBlock(PF);
  if not IsamOK then begin
    {Error handling}
  end;
end;
...
if not Locked then begin
  BTUnlockFileBlock(PF);
  if not IsamOK then begin
    {Error handling}
  end;
end;
end;
```

This example locks a fileblock only if it wasn't already locked. Later, the fileblock lock is released only if there was no fileblock lock prior to this example. This way the code doesn't change the lock state of the fileblock for the functions that called it.

See Also

BTFileBlockIsReadLocked

BTFileBlockIsOpen

Syntax

```
function BTFileBlockIsOpen; (IFBPtr : IsamFileBlockPtr) : Boolean;
```

Purpose

Check whether the fileblock is open.

Description

This functions returns True if the fileblock is open, False if not. This allows an application to avoid opening a fileblock more than once simultaneously, which can cause odd DOS-specific side effects besides just wasting file handles and heap space.

If two or more variables of type IsamFileBlockPtr are being reused to represent several actual fileblocks, it is important for the application to assure that the application's IsamFileBlockPtr variable is set to Nil as soon as the fileblock is closed. Otherwise it is possible that BTFileBlockIsOpen may return an incorrect value of True when it is passed a parameter containing a previously initialized value.

Example

```
var
  Open : Boolean;
...
Open := BTFileBlockIsOpen(PF);
if not Open then begin
  BTOpenFileBlock(PF, FName, False, False, False, False);
  if not IsamOK then begin
    {Error handling}
  end;
end;
...
if not Open then begin
  BTCloseFileBlock ( PF );
  if not IsamOK then begin
    {Error handling}
  end;
end;
```

Opens a fileblock only if it was not previously opened. Subsequently the fileblock is closed only if it was previously closed. In this way the code doesn't change the open state of the fileblock for the functions that called it.

See Also

BTCloseFileBlock

BTOpenFileBlock

BTFileBlockIsReadLocked

Syntax

```
function BTFileBlockIsReadLocked, (IFBPtr : IsamFileBlockPtr) :  
    Boolean;
```

Purpose

Check whether the fileblock is read locked.

Description

This function determines whether the fileblock is locked for reading by the calling workstation. If so, True is returned, otherwise False.

This function returns False if B-Tree Filer was compiled with the compiler directive NoNet or if IFBPtr^ is a single user fileblock. If NoNet wasn't defined, i.e., a real network interface was activated, the present read lock state of the network fileblock is returned. This is also the case when a network emulation mode is active (see _4.G).

Note that BTFileBlockIsReadLocked doesn't determine whether another workstation locked the fileblock. That can be determined only by attempting to lock the fileblock and checking for success. (Multiple stations can read lock a fileblock; but only one can lock it for writing.)

Example

```
var  
    ReadLocked : Boolean;  
...  
ReadLocked := BTFileBlockIsReadLocked(PF);  
if not ReadLocked then begin  
    BTReadLockFileBlock(PF);  
    if not IsamOK then begin  
        {Error handling}  
    end;  
end;  
...  
if not ReadLocked then begin  
    BTUnlockFileBlock(PF);  
    if not IsamOK then begin  
        {Error handling}  
    end;  
end;  
end;
```

This example read locks a fileblock only if it wasn't already read locked. The fileblock read lock is released only if there was no read lock prior to this example. In this way the code doesn't change the lock state of the fileblock for functions that called it.

See Also

BTFileBlockIsLocked

BTFileLen / BTFindKey

Syntax

```
function BTFileLen; (IFBPtr : IsamFileBlockPtr) : LongInt;
```

Purpose

Determine the total number of records in the data file.

Description

This record count includes the first reserved data record, the deleted records, and the presently used data records.

The physical length of the data file can be calculated as follows:

```
BTDatRecordSize(IFBPtr) * BTFileLen(IFBPtr)
```

Example

```
var
    FLen : LongInt;
...
    FLen := BTFileLen(PF);
    if not IsamOK then begin
        {Error handling}
    end;
    Writeln('The total number of records is ', FLen);
    Writeln('Data file size is ', BTDatRecordSize(PF)*FLen, '
bytes');
```

In a network environment, BTFileLen must read the header of the fileblock, which could lead to an error. In contrast, the value returned by BTDatRecordSize is obtained without accessing the disk and therefore can never encounter an error.

See Also

BTFreeRecs

BTUsedRecs

Syntax

```
procedure BTFindKey; (IFBPtr : IsamFileBlockPtr; Key : Word;
    var UserDatRef : LongInt; UserKey : IsamKeyStr);
```

Purpose

Search for a specific key.

Description

The index file is searched for an exact match with the key UserKey of index Key. The matching data record reference is returned in UserDatRef. BTFindKey sets the sequential pointer upon successful execution, so that the procedures BTPrevKey and BTNextKey can be used thereafter.

If a secondary key that is entered more than once in the index file is found, the smallest matching data record reference is returned in UserDatRef. Successive calls to BTNextKey return increasingly larger values for UserDatRef for the identical key. If no data records have ever been deleted, this order is the same one in which the records were added to the fileblock. Never depend on this order, however, since by deleting a data record and then adding a new one, this order can be changed.

If the specified key string is not found, the variable IsamError contains the value 10200. If another workstation has the fileblock locked for writing, IsamError will contain 10399.

In a network environment, if the fileblock is not read locked when this routine is called, then a temporary read lock is placed while BTFindKey performs its search through the index.

Example

See _4.C and _4.F for examples.

BTFileLen / BTFindKey

See Also

~~BTFindKeyAndRef~~

BTSearchKey

~~BTKeyExists~~

BTSearchKeyAndRef

BTFindKeyAndRef

Syntax

```
procedure BTFindKeyAndRef; (IFBPtr : IsamFileBlockPtr; Key : Word;  
                           var UserDatRef : LongInt; var UserKey :  
                           IsamKeyStr;  
                           NotFoundSearchDirection : Integer);
```

Purpose

Search for a key with the specified data record reference.

Description

This procedure searches for the key UserKey that also has the data reference number UserDatRef. If this combination is not found, the parameter NotFoundSearchDirection determines whether to search further. A value of 0 means to end the search. A positive value means to search for a following key and a negative value means to search for a previous key.

If no key with the given reference is found, the variable IsamError contains the value 10270. If NotFoundSearchDirection doesn't have a value of 0, an error code associated with the procedures BTNextKey and BTPrevKey is returned if the additional search is not successful.

If another workstation has the fileblock locked for writing, IsamError contains 10399.

This procedure is not designed to return the data reference number of a specific key; this must already be known. More importantly, it allows the sequential pointer to be positioned in the middle of a block of identical secondary keys. This is especially useful when browsing or creating lists.

BTFindKeyAndRef sets the sequential pointer upon successful execution, so that the procedures BTPrevKey and BTNextKey can be used thereafter.

In a network environment, if the fileblock is not read locked when this routine is called, then a temporary read lock is placed while BTFindKeyAndRef performs its search through the index.

Example

```
var  
  Ref      : LongInt;  
  SRef     : LongInt;  
  IKS      : IsamKeyStr;  
  SIKS     : IsamKeyStr;  
  PTemp    : PersonDef;  
  Stop     : Boolean;  
...  
  BTFindKey(PF, 2, Ref, IKS);  
  Stop := False;  
  while IsamOK and (Not Stop) do begin  
    BTGetRec(PF, Ref, PTemp, False);  
    if IsamOK then begin  
      if PTemp.Age >= 18 then begin  
        SRef := Ref;  
        SIKS := IKS;  
        Stop := True;  
      end else  
        BTNextKey(PF, 2, Ref, IKS);  
    end;  
  end;  
  if BTIsamErrorClass > 1 then begin  
    {Error handling}  
  end;  
...  
  BTFindKeyAndRef(PF, 2, SRef, SIKS, 0);
```

BTFindKeyAndRef

```
if not IsamOK then begin
    {Error handling}
end;
```

The beginning of this example searches for a specific key in index number 2. The while loop then finds the first data record that has an Age field greater than or equal to 18. After checking for errors, further new key search operations can be performed. Finally BTFindKeyAndRef is used to set the sequential pointer on the data record that was found in the while loop.

See Also

BTFindKey
BTNextDiffKey
BTPrevDiffKey
BTSearchKey

BTKeyExists
BTNextKey
BTPrevKey
BTSearchKeyAndRef

BTFindRecRef / BTFlushAllFileBlocks

Syntax

```
procedure BTFindRecRef;(IFBPtr : IsamFileBlockPtr; var UserDatRef :  
LongInt;  
NotFoundSearchDirection : Integer);
```

Purpose

Search for a record with the specified reference number.

Description

This routine mimics the BTFindKeyAndRef routine without using an index. The record with reference UserDatRef is read; if it is not deleted then this routine returns immediately without error.

If the record is deleted, the parameter NotFoundSearchDirection determines whether to search further through the data file. A value of 0 means to end the search. A positive value means to search forwards sequentially through the records until the first active record is found, or the end of the file is reached. A negative value means to search backwards through the records until the first active record is found, or the start of the file is reached.

If an active record is found, its reference number is returned in UserDatRef with IsamOK set to True and IsamError set to zero. If an active record could not be found (i.e., NotFoundSearchDirection was zero and the record was deleted, or the search could not find an active record), then IsamOk is set to False and IsamError is set to 10275.

If a locked record was encountered during the initial read or the subsequent searching, this procedure returns immediately with IsamError set to 10390 and the offending record reference number in UserDatRef. You can then decide whether to jump over this record or perform some other type of action.

See Also

BTNextRecRef

BTPrevRecRef

Syntax

```
procedure BTFlushAllFileBlocks;;
```

Purpose

Write buffered data for all open fileblocks to disk.

Description

This procedure writes all buffered new data for all open fileblocks to disk. For network fileblocks, only those that are locked will be written. Fileblocks opened in save mode never have data buffered in memory, so no action occurs for those fileblocks.

Example

This procedure needs to be called only when an application wants to assure that all files are completely updated on disk. A short meaningful example is therefore difficult to construct. An appropriate situation might exist when all open fileblocks contain related information and a transaction affecting several of the fileblocks was just completed.

See Also

BTFlushFileBlock

BTFlushFileBlock / BTForceNetBufferWriteThrough

Syntax

```
procedure BTFlushFileBlock; (IFBPtr : IsamFileBlockPtr);
```

Purpose

Write buffered data for the specified fileblock to disk.

Description

A fileblock's data and index information may be buffered in B-Tree Filer page buffers, in disk buffers managed by DOS, in disk caches, or in a file server's memory. This procedure writes all new data in such buffers to the physical disk device and updates the file directory entries.

BTFlushFileBlock has roughly the same effect as the sequence BTCloseFileBlock, BTOpenFileBlock except that BTFlushFileBlock is more efficient.

Calling this procedure is pointless for fileblocks that were opened in save mode, since there is never any new data in memory. BTFlushFileBlock does nothing for network fileblocks that are not locked for writing at the time of the call. No error is generated in this case.

The flushing mechanism used by BTFlushFileBlock varies depending on the value of the typed constant IsamFlushDOS33 and on whether the fileblock is a network fileblock. See the description of IsamFlushDOS33 at the beginning of this chapter for more information.

If BTFlushFileBlock fails because it cannot obtain a free file handle, IsamError returns 10150.

See Also

BTFlushAllFileBlocks
BTForceWritingMark

BTForceNetBufferWriteThrough

Syntax

```
procedure BTForceNetBufferWriteThrough; (DoIt : Boolean);
```

Purpose

Activate write-through for all save mode files in a network environment.

Description

This routine is relevant only when one or more network fileblocks are being used in save mode. The default value for write-through is False.

Write-through refers to the process of flushing data from the memory buffers of a network file server or a multi-user operating system monitor. Many multi-user systems and networks buffer data being written to files instead of performing the write to disk immediately. Although the operating system correctly provides the most recent data values to all workstations that request them, this buffering can cause lost data in the event of a crash at the file server, even for fileblocks opened in save mode.

Save mode normally flushes all data from B-Tree Filer's internal buffers before any B-Tree Filer call returns. However, save mode does not guarantee that operating system buffers are flushed. After you activate write-through by calling BTForceNetBufferWriteThrough with the parameter True, all save mode routines also flush operating system buffers before returning.

Automatic flushing activated via BTForceNetBufferWriteThrough uses the same techniques as does BTFlushFileBlock. The flushing mechanism varies depending on the value of the typed constant IsamFlushDOS33. See the description of IsamFlushDOS33 at the beginning of this chapter for more information.

See Also

BTFlushFileBlock

BTForceWritingMark

BTForceWritingMark

Syntax

```
procedure BTForceWritingMark; (FFM : Boolean);
```

Purpose

Enable or disable write-through of the modified mark.

Description

When any B-Tree Filer procedure modifies the data or index file, a flag is set in the system record (reference number 0) of the data file. This flag indicates that data is buffered, either in B-Tree Filer internal buffers or in operating system file buffers. The flag remains set until the fileblock is closed or a flush is executed by calling BTFlushFileBlock or by an automatic internal flush. If a later call to BTOpenFileBlock finds the flag still set in the data file, it returns with an error code of 10010, which means that the integrity of the index and data files is questionable.

So far so good; unfortunately the flag itself may remain in an operating system buffer, and therefore this check for integrity may not be effective. Calling BTForceWritingMark with FFM set to True enables a mode in which the writing mark is immediately flushed to disk after it is written.

A call to this procedure with the value True increases the integrity of your data. The default value is False since the execution speed of B-Tree Filer is otherwise reduced slightly. For the greatest possible guarantee of data integrity (and additional degradation in speed), open the fileblock in save mode and activate BTForceNetBufferWriteThrough.

Example

```
BTAddRec (PF, RefNr, PersonRec);
```

If the computer is rebooted unexpectedly after this call to BTAddRec, the probability is high that neither the data record nor the header of the data file made it to the disk. In this case, the data file's modified flag will not indicate that the file integrity is questionable even though the data file is incomplete. If BTForceWritingMark had been called earlier with the parameter True, the modified flag would be guaranteed to indicate poor file integrity after these events.

See Also

BTFlushFileBlock

BTForceNetBufferWriteThrough

BTFreeRecs / BTGetAfterNextUsedAddRecRef

Syntax

```
function BTFreeRecs; (IFBPtr : IsamFileBlockPtr) : LongInt;
```

Purpose

Determine the number of free data records.

Description

A "hole" exists in the data file for every record that was deleted using the procedure BTDeleteRec. The "holes" will be refilled after later calls to BTAddRec or after the fileblock is rebuilt with RebuildFileBlock. BTFreeRecs returns the current number of these deleted records.

Example

```
var
    FRecs : LongInt;

...
FRecs := BTFreeRecs(PF);
if not IsamOK then begin
    {Error handling}
end;
Writeln('The number of free data records is ', FRecs);
Writeln('This is equivalent to ', BTDatRecordSize(PF)*FRecs,
        ' free bytes in the data file.');
```

To determine the present count of records in a network environment BTFileLen must read the header of the data file, which could lead to an error. By contrast the value returned from BTDatRecordSize is obtained without file access and therefore can never encounter an error.

See Also

BTFileLen

BTUsedRecs

Syntax

```
function BTGetAfterNextUsedAddRecRef; (IFBPtr : IsamFileBlockPtr) :
LongInt;
```

Purpose

Return the data reference that will be assigned to the record added after the next record is added.

Description

This function is primarily for internal use.

See Also

BTGetNextUsedAddRecRef

BTGetAllowDupKeys

Syntax

```
function BTGetAllowDupKeys(IFBPtr : IsamFileBlockPtr; KeyNr : Word )  
: Boolean;
```

Purpose

Return True if an index allows duplicate keys.

Description

This function allows you to determine at runtime whether the index specified by KeyNr of fileblock IFBPtr allows duplicate keys. This information is encoded into the fileblock by BTCreateFileBlock.

See Also

BTCreateFileBlock

BTGetKeyLen

BTGetApprKeyAndRef

Syntax

```
procedure BTGetApprKeyAndRef, (IFBPtr : IsamFileBlockPtr; Key : Word;  
                                RelPos : Word; Scale : Word;  
                                var UserKey : IsamKeyStr; var UserDatRef  
                                : LongInt);
```

Purpose

Return the key and data reference for the record at the specified relative position.

Description

Given a relative position RelPos in the range 0..Scale, BTGetApprKeyAndRef returns the existing key UserKey and UserDatRef whose position within the B-tree most closely approximates the specified position. This is provided to convert a thumb position within a scroll bar to the corresponding key and record number for highlighting a record within a file browser. Scale should typically be the number of screen characters assigned to the scroll bar, minus 1, and RelPos should correspond to the slider position relative to the base location of the scroll bar.

BTGetApprKeyAndRef is *not* intended to return an exact key string for a RelPos in the range 0..BTUsedKeys-1, as would be required to create a Turbo Professional or Object Professional pick list that directly displayed the elements of a fileblock.

This procedure is written to obtain sufficient speed for interactive operation; good performance is achieved by reading only a small part of the B-tree to perform the calculations. The drawback to this method is the inexactness of the result. Only a full traversal of the tree could return an exact result, and this would lead to unacceptable execution time.

Large calculation errors occur only when Scale is too large. Acceptable values for Scale depend on the instantaneous state of the B-tree and the value of the constant CreatePageSize (default = 62). Results are good if Scale is no larger than about 0.67*CreatePageSize at most. The procedure always attempts to choose the key in the middle of the range that corresponds to equivalent RelPos.

To obtain the most stable results, it's a good idea to pass the values returned by BTGetApprKeyAndRef to BTGetApprRelPos and then use the RelPos returned by that routine to call BTGetApprKeyAndRef once more. In this way the two routines tend to cancel out errors.

BTGetApprKeyAndRef returns error 10280 if the specified index contains no keys. It returns error 10420 if Scale = 0 or if RelPos is larger than Scale.

Example

Assume that Scale is set to 19 and that there are 400 keys (numbered 1 to 400) in a well balanced B-tree. The following table shows the approximate keys returned by BTGetApprKeyAndRef:

RelPos	UserKey
0	10
1	30
10	210
19	390

Note that the extremes of the RelPos range do not return the extremes of the key range. Instead the returned keys are in the center of the group of keys that evaluate to the same RelPos.

See Also

BTGetApprRelPos

BTGetApprRecPos / BTGetApprRecRef

Syntax

```
procedure BTGetApprRecPos; (IFBPtr : IsamFileBlockPtr; var RelPos :  
Word;  
Scale : Word; UserDatRef : LongInt);
```

Purpose

Calculate the relative position of a record in the data file.

Description

This procedure works in an analogous manner to BTGetApprRelPos but does not refer to any index or to the B-Tree; it just uses the data file. Given a UserDatRef data reference it calculates a relative position RelPos of the record in the data file, in the range 0..Scale.

This routine could be used to convert a data reference into a thumb position within a scrollbar. In text mode, Scale should typically be the number of screen characters assigned to the scrollbar, minus 1.

If Scale is zero, this procedure returns with IsamError set to 10425.

See Also

BTGetApprRelPos

BTGetApprRecRef

Syntax

```
procedure BTGetApprRecRef; (IFBPtr : IsamFileBlockPtr; RelPos : Word;  
Scale : Word; var UserDatRef : LongInt);
```

Purpose

Return the data reference of the record at the specified relative position.

Description

This procedure works in an analogous manner to BTGetApprKeyAndRef but does not refer to any index or to the B-Tree; it just uses the data file. Given the relative position RelPos in the range 0..Scale, BTGetApprRecRef returns the data reference UserDatRef of the record that most closely approximates the specified position.

The reference number returned will not necessarily refer to an active record. It could be for a deleted record.

This routine could be used to calculate a record number given a thumb position within a scrollbar. In text mode, Scale should typically be the number of screen characters assigned to the scrollbar, minus 1.

If Scale is zero, or RelPos is greater than Scale this procedure returns with IsamError set to 10420.

See Also

BTGetApprKeyAndRef

BTGetApprRecPos

BTGetApprRelPos

Syntax

```
procedure BTGetApprRelPos, (IFBPtr : IsamFileBlockPtr; Key : Word;  
                           var RelPos : Word; Scale : Word; UserKey :  
                           IsamKeyStr;  
                           UserDatRef : LongInt);
```

Purpose

Compute the relative position of a key and data reference in the B-tree.

Description

Given UserKey and UserDatRef, which specify a key within index Key of a fileblock, BTGetApprRelPos returns a value RelPos in the range 0..Scale, which corresponds to the relative position of the key within the index. This routine is provided to convert a key string to a thumb position within a scroll bar. Scale should typically be the number of screen characters assigned to the scroll bar, minus 1.

See BTGetApprKeyAndRef for further information.

Example

Assume that Scale is set to 19 and that there are 400 keys (numbered 1 to 400) in a well balanced B-tree. The following table shows the ranges of keys that return equivalent values of RelPos.

UserKey range	RelPos returned
1-20	0
21-40	1
201-220	10
381-400	19

See Also

BTGetApprKeyAndRef

BTGetInternalDialogID

Syntax

```
function BTGetInternalDialogID(IFBPtr : IsamFileBlockPtr) : Word;
```

Purpose

Return the internal number for a fileblock.

Description

If IFBPtr points to a fileblock used in single user mode or net emulation mode, zero is returned. For a genuine network or multi-tasking system, the returned value is between 1 and MaxNrOfWorkStations.

BTGetInternalDialog may return a different number for each fileblock opened by an application. To determine the internal dialog ID, B-Tree Filer places a "phantom lock" at an offset just preceding \$7FFFFFFF in the fileblock's dialog file. The lock is removed when the fileblock is closed.

There is no need for you to use this number. Note that the value may vary from run to run of a program even on the same fileblock. The only guarantee is that no two workstations that have the same fileblock open will have the same internal workstation number (unless there is a setup problem or a bug in the network operating system).

Example

```
var
  MyFileBlockPtr : IsamFileBlockPtr;

...
if BTIsNetFileBlock(MyFileBlockPtr) then
  if (BTGetInternalDialogID(MyFileBlockPtr) <> 0) then
    {Net fileblock}
  else
    {Net emulation}
  else
    {single user fileblock}
...
```

If the fileblock specified by MyFileBlockPtr is a network fileblock, the internal workstation number is retrieved. If the internal workstation number is zero, the fileblock was opened in net emulation mode. If the internal workstation number is not zero, it was opened in network mode.

BTGetKeyLen / BTGetNextUsedAddRecRef

Syntax

```
function BTGetKeyLen; (IFBPtr : IsamFileBlockPtr; KeyNr : Word) :  
Word;
```

Purpose

Return the length of the keys in an index.

Description

This function allows you to determine at runtime how long the keys are in index KeyNr of fileblock IFBPtr. This information is encoded into the fileblock by BTCreateFileBlock.

See Also

BTCreateFileBlock

BTGetAllowDupKeys

Syntax

```
function BTGetNextUsedAddRecRef; (IFBPtr : IsamFileBlockPtr) :  
LongInt;
```

Purpose

Return the data reference that will be assigned to the next record added.

Description

If you use this function in a network setting, be sure to lock the fileblock before calling it and leave the lock in place until the record is actually added. Otherwise, the number returned may be changed by another workstation before the record is added.

See Also

BTGetAfterNextUsedAddRecRef

BTGetRec

Syntax

```
procedure BTGetRec; (IFBPtr : IsamFileBlockPtr; RefNr : LongInt; var  
Dest;  
ISOLock : Boolean);
```

Purpose

Read the data record with the specified reference number.

Description

RefNr is usually obtained from a prior index operation. For example, the UserDatRef parameter returned by a successful call to BTFindKey could be passed to BTGetRec.

Since Dest is untyped, the compiler will not detect an invalid variable passed in this position. Assure that the variable passed is large enough to hold a complete data record.

The parameter ISOLock has meaning only for a network fileblock. With a value of True, BTGetRec reads the record even if the fileblock is locked for writing (level 1 or 2) by another workstation. With a value of False, the operation is aborted and returns error 10399 when such a lock is found on the fileblock. A record lock (level 3) causes BTGetRec to fail no matter what the state of ISOLock. The procedure BTGetRecReadOnly must be used to get around such a lock.

The parameter ISOLock should be set to True with extreme care. It should be clear to the user that the data record might have just been changed and therefore possibly doesn't contain the newest data.

Example

See _4.C and _4.F for examples.

See Also

BTGetRecReadOnly

BTGetRecordInfo

Syntax

```
procedure BTGetRecordInfo, (IFBPtr : IsamFileBlockPtr; Ref : LongInt;  
                           var Start : LongInt; var Len : LongInt;  
                           var Handle : Word);
```

Purpose

Return information about a specific data record.

Description

By using the information returned by BTGetRecordInfo, an application can perform additional forms of record locking not provided directly by B-Tree Filer. The three var parameters Start, Len, and Handle fully describe where data record Ref is to be found. Handle is the file handle of the data file, Start is the offset of the first byte of the data record, and Len is the length of the data record.

Example

```
var  
  DataHandle : Word;  
  RecRef, RecStart, RecLen : LongInt;  
...  
function NovellLockRecShare(Start, Len : LongInt;  
                           Handle : Word) : Boolean;  
  
type  
  LoHi = record Lo, Hi : Word; end;  
var  
  IRR : Registers;  
begin  
  with IRR do begin  
    BX := Handle;  
    CX := LoHi(Start).Hi; DX := LoHi(Start).Lo;  
    SI := LoHi(Len).Hi; DI := LoHi(Len).Lo;  
    BP := $0000;  
    AX := $BC03;  
    MsDos(IRR);  
    NovellLockRecShare := (AL = 0);  
  end;  
end;  
...  
BTFindKey(PF, 1, RecRef, 'SMITH');  
if not IsamOK then begin  
  {Error handling}  
end;  
BTGetRecordInfo(PF, RecRef, RecStart, RecLen, DataHandle);  
if not NovellLockRecShare(RecStart, RecLen, DataHandle) then  
begin  
  {Error handling}  
end;  
...  
end;
```

This example implements a function that locks a file with a "shareable lock" on a Novell NetWare network, providing that NETX has been loaded at the workstation. The data record reference returned by BTFindKey is passed to BTGetRecordInfo. The parameters returned by BTGetRecordInfo are passed to the Novell locking routine.

BTGetRecReadOnly / BTGetSearchForSequential

Syntax

```
procedure BTGetRecReadOnly; (IFBPtr : IsamFileBlockPtr; RefNr : LongInt; var Dest);
```

Purpose

Read a data record, even if the record is locked.

Description

RefNr is usually obtained from a prior index operation. For example, the UserDatRef parameter returned by a successful call to BTFindKey could be passed to BTGetRecReadOnly.

Since Dest is untyped, the compiler will not detect an invalid variable passed in this position. You must assure that the variable passed is large enough to hold a complete data record.

BTGetRecReadOnly reads a fileblock locked by another workstation without difficulty. To this extent, it works identically to BTGetRec with ISOLock set to True. In addition, however, BTGetRecReadOnly will read a record even with a record lock. Because the first four bytes of the data record cannot be read when it is locked, these bytes are not initialized by BTGetRecReadOnly (only if the record is locked). However, if these bytes are reserved for B-Tree File's use as a deletion tag, no critical information is lost. The remaining bytes of the record are filled in as usual. When the record is locked and the first four bytes are lost, BTGetRecReadOnly returns a warning code of 10205 in IsamError.

BTGetRecReadOnly should be used only with extreme care. It should be clear to the user that the data record might have just been changed and therefore possibly doesn't contain the newest data. Because the first four bytes of a locked record cannot be read, it's also possible that the application may be dealing with a just-deleted record that it can't detect.

See Also

BTGetRec

Syntax

```
procedure BTGetSearchForSequential; (IFBPtr : IsamFileBlockPtr; Key :  
Word;  
var SFS : Boolean);
```

Purpose

Return the SearchForSequential state for the specified index.

Description

This routine can be used to save the current state of the SearchForSequential option for a particular index and fileblock, so that the state can be changed and later restored. See BTSetSearchForSequential for further information.

Example

```
var  
  SFS : Boolean;  
...  
  BTGetSearchForSequential (PF, 1, SFS);  
  BTSetSearchForSequential (PF, 1, True);  
...  
  BTSetSearchForSequential (PF, 1, SFS);
```

To perform a certain set of index scanning operations, it is imperative that SearchForSequential be enabled for index number 1. To preserve the prior setting of SearchForSequential, its value is saved and then restored at the end.

See Also

BTSetSearchForSequential

BTGetStartingLong / BTIndexFileName

Syntax

```
procedure BTGetStartingLong, (IFBPtr : IsamFileBlockPtr, RefNr :  
    LongInt;  
                                var Dest : LongInt);
```

Purpose

Read the first four bytes of the specified record.

Description

This routine provides a fast way to read just the first four bytes of a record. The first four bytes are generally used to determine whether the specified record is deleted.

Example

```
var  
    DelFlag : LongInt;  
...  
    BTGetStartingLong(PF, RefNr, DelFlag);  
    if IsamOK and (DelFlag = 0) then begin  
        {Record is not deleted}  
        ...  
    end;
```

Checks to see whether a record is deleted before performing an operation on it.

See Also

BTGetRec

Syntax

```
function BTIndexFileName; (IFBPtr : IsamFileBlockPtr) : IsamFileName;
```

Purpose

Return the DOS name of the fileblock's index file.

Description

This function returns the index file name, including extension, of the specified fileblock. If the name passed to BTOpenFileBlock did not include a drive and directory, neither does the string returned by BTIndexFileName.

Example

See BTDataFileName for a related example.

See Also

BTDataFileName

BTInformTTSAbortSuccessful

Syntax

```
procedure BTInformTTSAbortSuccessful; (IFBPtr : IsamFileBlockPtr);
```

Purpose

Clear all buffered internal data and remove all locks from a file after a successful NetWare TTS transaction abort.

Description

This function can be called *only* if you are using a Novell network with TTS and have successfully executed the nwTTSAbort function from the NWTTS unit (see _8.A.7).

This routine releases all locks on the fileblock without writing any buffered data. All buffered data is marked invalid, so that only information still located in the files is valid.

Remember that the dialog file of a fileblock should never be marked transactional. See _8.A.7 for more information and an example.

BTInitIsam

Syntax

```
function BTInitIsam, (ExpectedNet : NetSupportType, Free : LongInt,  
                    NrOfEMSTreePages : Word) : LongInt;
```

Purpose

Initialize B-Tree Filer and allocate memory from the normal heap and/or the EMS heap.

Description

This function must be called before any routines of the FILER unit can be called either directly or indirectly.

BTInitIsam performs two tasks. First, the network specified by ExpectedNet is initialized. To successfully initialize a network, the appropriate compiler directives must be specified in BTDEFINE.INC in order to link the corresponding network interface code into the executable (see _2.A).

If BTDEFINE.INC specifies the NoNet define, BTInitIsam ignores the ExpectedNet parameter and initializes the FILER unit for single-user operation.

If BTDEFINE.INC defines one or more network interface directives, ExpectedNet can take any of the corresponding values in NetSupportType, in addition to NoNet. If ExpectedNet is set to NoNet, B-Tree Filer enables its network emulation mode and no network is initialized. In this case B-Tree Filer locking and unlocking routines don't really do anything. See _4.G for additional information about network emulation.

To enable true network operation, ExpectedNet must be set to a value other than NoNet. A successful call to BTInitIsam initializes procedure pointers for routines that perform network-specific operations for locking, unlocking, and program shutdown.

The parameter ExpectedNet allows you to choose a network at runtime, thus allowing one program to be used on different networks. The determination of network type can be obtained from a command line parameter (as in the example NETDEMO.PAS) or by another method (as in NETINFO.PAS). Changing networks, for example from Novell to NoNet, is even possible at runtime. To do so, all open fileblocks must be closed and the procedure BTExitIsam must be called to leave the previous network.

The second task performed by BTInitIsam is to configure internal index buffers. The parameters Free and NrOfEMSTreePages control the configuration. Free concerns itself with the normal heap and NrOfEMSTreePages with the EMS Heap.

The Free parameter specifies how many bytes of the normal heap to leave available after the call. This allows some memory to be reserved for other purposes. If BTInitIsam should allocate the absolute minimum amount of memory required for B-Tree Filer functionality from the normal heap, the value of MinimizeUseOfNormalHeap can be added to the value of Free.

If your application will run in real mode, the optimal value for Free depends on how the rest of the application uses the heap. Even if the program does not directly allocate heap space, other routines from B-Tree Filer require heap space themselves. For example, a call to BTOpenFileBlock uses about 500 bytes of heap space. Many windowing routines (such as those from Object Professional) also make liberal use of the heap.

If the amount of heap space needed for the rest of the program is known in advance, then the Free parameter should simply specify that amount of heap space plus a safety margin. For example, if a program will open two fileblocks and use no other heap space, then a safe value for Free would be 2000 (two 500 byte blocks plus 1000 bytes of margin). This value of Free would maximize the number of page buffers and optimize indexing performance, especially in a single-user environment.

At the other extreme is an application that makes extensive use of the heap itself; for example, a text editing program that stores text strings on the heap. In this case, you should use the special constant MinimizeUseOfNormalHeap to tell BTInitIsam to use as little normal heap space as possible. If no EMS memory is available, the minimum normal heap usage is 19688 bytes as calculated above for default settings. If sufficient EMS is available, normal heap usage can be limited to 225 bytes. See the examples below.

BTInitIsam

If your application will run in protected mode or under Windows, the heap size can run into megabytes, and to make things worse you may have a virtual memory manager too. In this case you must look at the Free parameter from the opposite direction, not how much memory the rest of the application needs, but how much memory B-Tree Filer can profitably use. The best advice here is to use MemAvail-SomeFixedAmount for the Free parameter.

There is another consideration to take into account if the application is multi-user. Page buffers are invalidated and must be reloaded when another workstation modifies the index of a fileblock (this happens automatically). As a result, the effectiveness of a large index buffer is not so clear when running on a network. You may be wasting a lot of heap space because the application may run just as fast with fewer page buffers. This consideration is the same in real mode, protected mode or Windows.

So, in conclusion, the best advice might be to experiment with different values of Free to get the best balance between the amount of memory used and the speed of the application.

In protected mode and Windows, it is generally best to force BTInitIsam to use a fixed size page buffer as shown in the following example:

```
Pages := BTInitIsam(WhateverNet, MemAvail-500000, 0);
```

This call allocates about 500,000 bytes for B-Tree Filer's page stack. In this environment, it is never a good idea to allow B-Tree Filer to allocate all but a fixed size of memory, as it could conceivably take a long time to allocate the memory and experiments have shown that the heap manager will run out of selectors before the free memory is fully allocated.

In real mode, NrOfEMSTreePages determines the maximum number of index pages that should be allocated on the EMS heap. To use the EMS heap at all, the conditional UseEMSHeap must be defined in BTDEFINE.INC. Otherwise, the NrOfEMSTreePages parameter is ignored; as it is when running in protected mode and in Windows. More information can be found in _2.B.

It is not an error if the number of pages specified by NrOfEMSTreePages is not available. BTInitIsam will then use whatever it can obtain. If NrOfEMSTreePages or the obtained number of pages is less than the constant MaxHeight (default value of 8), additional buffers are allocated from the normal heap.

For every page allocated from the EMS heap, 25 bytes from the normal heap are also allocated to install a descriptor for the EMS page. As long as the number of pages in EMS is not higher than a few hundred, this allocated memory is not significant. On the other hand, if more than 1000 pages are used in the EMS heap, the descriptors will use over 25000 bytes of the normal heap. Hence, the value of Free is taken into account even when allocating pages on the EMS heap, to prevent the descriptors from using normal heap space required by another portion of the program.

The index buffers are organized as a circular list of descriptors, each of which contains a pointer to the actual index page. The descriptors are always allocated from the normal heap and each consumes 25 bytes. The actual index pages are allocated either from the normal heap or in EMS memory. The size of each index page in the buffer can be calculated as:

$$(<MaxKeyLen>+9) * <MaxPageSize> + 18$$

Using the standard values for MaxKeyLen (30) and MaxPageSize (62) leads to a memory requirement of 2436 bytes for each page.

At the minimum, BTInitIsam must be able to allocate MaxHeight page buffers. For the standard value of MaxHeight (8), the normal heap usage can thus range from a minimum of 200 bytes if all pages are stored in EMS, to 19688 bytes if all pages are stored on the normal heap.

MaxKeyLen, MaxHeight, and MaxPageSize must be adjusted with caution. See the description of these constants at the beginning of this chapter.

The function return value of BTInitIsam is of type LongInt. If IsamOK is True after BTInitIsam returns, the low word of this LongInt contains the number of pages that were allocated in the normal heap, and the high word contains the number of pages in the EMS heap. If IsamOK is False and IsamError is 10000, the LongInt returns the number of index pages that could have been allocated. The difference between MaxHeight and this number can be used to calculate how much more memory is needed for a successful initialization. For any other IsamError, the LongInt function result returns 0.

BTInitIsam

Important note: the minimum requirement for index buffers does not increase as more fileblocks are opened. B-Tree Filer shares the index buffers among all open fileblocks on a demand basis.

However, performance may be improved by allocating an increased number of page buffers when many fileblocks are to be open simultaneously.

It is an error (10450) to call BTInitIsam more than once without an intervening call to BTExitIsam.

Examples

```
Pages := BTInitIsam(Novell, 30000, 300);
```

Initializes B-Tree Filer for Novell NetWare. At least 30000 bytes of the normal heap are to remain free after the call completes. Up to 300 page buffers are used on the EMS heap.

```
Pages := BTInitIsam(Novell, 30000+MinimizeUseOfNormalHeap, 300);
```

Initializes B-Tree Filer for Novell NetWare. At least 30000 bytes of the normal heap are to remain free. Up to 300 buffers are used on the EMS heap. Only if less than MaxHeight buffers could be allocated from the EMS heap is memory allocated from the normal heap. For example, if only 3 buffers could be allocated in EMS, MaxHeight-3 = 5 buffers are allocated from the normal heap, subject to leaving at least 30000 bytes of normal heap space free.

```
Pages := BTInitIsam(NoNet, 10000+MinimizeUseOfNormalHeap, 0);
```

Initialize B-Tree Filer without a network. Keep at least 10000 bytes of the normal heap free. No buffers are allocated from the EMS heap, and MaxHeight buffers are allocated from the normal heap. If BTInitIsam cannot allocate MaxHeight pages *and* leave 10000 bytes free, it fails.

```
Pages := BTInitIsam(MsNet, MinimizeUseOfNormalHeap, 0);
```

Initializes B-Tree Filer for an MsNet-compatible network. Memory usage in normal RAM is minimized, and no EMS space is used.

BTIsamErrorClass / BTIsamLockRecord

Syntax

```
function BTIsamErrorClass; : Integer;
```

Purpose

Determine the error class for the current value of IsamError.

Description

This function returns one of five error class values dependent upon the current value of IsamError. This provides a coarser and simpler picture of error recognition.

All error codes are summarized with their error classes in Appendix A. The following error classes can occur:

- 0 No error (IsamError is zero)
 - 1 User error
This is more of a warning than an error. The previous operation could not be successfully completed. Examples include a key that couldn't be found with BTFindKey and a record whose first four bytes couldn't be read with BTGetRecReadOnly. All values of the variable IsamError between 10200 and 10299 are user errors, among others.
 - 2 Locking error (only on a network)
Errors of this class occur when a file access couldn't be completed. As a rule this error class occurs when a lock from another workstation prevents access. Another possible cause is an empty diskette drive or lack of privileges of the current user in the network. If after a retry the access is still not possible, the user should be given the opportunity to abort the operation.
 - 3 Operation did not succeed in save mode
This error class deals with serious errors and occurs only in save mode. All files of the corresponding fileblock are still functional but the desired operation could not succeed. Therefore this error class is a precursor to an error of class 4. If the condition that lead to the error cannot be corrected immediately, the best program response is to close up gracefully and abort.
 - 4 Hard error
An immediate correction of the error condition is required. If this is not possible, the program should abort. In some cases fileblocks were corrupted and require reconstruction.
-

Syntax

```
function BTIsamLockRecord; (Start : LongInt; Len : LongInt; Handle :  
Word;  
TimeOut : Word; DelayTime : Word) :  
Boolean;
```

Purpose

Place a lock using the current locking model.

Description

The bytes from Start to Start+Len-1 of the file with the specified DOS file handle are locked. BTIsamLockRecord retries for up to TimeOut milliseconds. After each failed lock attempt, it delays internally for DelayTime milliseconds. On a Novell network, retries are done internally at the file server. If the lock is placed successfully, True is returned; otherwise False.

BTIsamUnlockRecord / BTIsInitialized

Syntax

```
function BTIsamUnlockRecord(Start : LongInt; Len : LongInt;  
                           Handle : Word) : Boolean;
```

Purpose

Unlock records locked by BTIsamLockRecord.

Description

The bytes from Start to Start+Len-1 of the file with the specified handle are unlocked. The start and length must agree exactly with the parameters passed to BTIsamLockRecord when the lock was placed.

Syntax

```
function BTIsInitialized; : Boolean;
```

Purpose

Determine if B-Tree Filer was initialized.

Description

B-Tree Filer is initialized with BTInitIsam and closed down with BTExitIsam. This function returns False before BTInitIsam is called, True afterwards. When BTExitIsam is called, it again returns False.

Example

```
if not BTIsInitialized then  
  BTInitIsam(MsNet, MemAvail-50000, 0);
```

If B-Tree Filer is not initialized, then do so for an MsNet network and allow up to 50000 bytes for the page stack.

See Also

BTInitIsam

BTExitIsam

BTIsNetFileBlock

Syntax

```
function BTIsNetFileBlock; (IFBPtr : IsamFileBlockPtr) : Boolean;
```

Purpose

Determine if a fileblock is a network fileblock.

Description

The term "network fileblock" is somewhat nebulous due to the numerous ways of configuring B-Tree Filer and opening fileblocks.

BTIsNetFileBlock returns False when the FILER unit was compiled with the NoNet compiler directive. Since the code for use on a network was never compiled in, all fileblocks are automatically local. BTIsNetFileBlock also returns False if the parameter Net passed to BOpenFileBlock was False when the fileblock was opened.

BTIsNetFileBlock returns True if the FILER unit was compiled with any network directive besides NoNet and the BOpenFileBlock was called with the Net parameter set to True. This occurs even if BTInitIsam was called with ExpectedNet set to NoNet because in this case B-Tree Filer emulates the network operation.

Example

```
{Make open request for a network fileblock}
BOpenFileBlock(PF, FName, False, False, False, True);
if not BTIsNetFileBlock(PF) then begin
    ...
end;
```

There are two reasons why BTIsNetFileBlock might return False for this fileblock: either the FILER unit was compiled with the NoNet define in BTDEFINE.INC, or the value for ExpectedNet passed to BTInitIsam did not correspond to a network interface activated in BTDEFINE.INC.

See Also

BTNetSupported

BTNoNetCompiled

BTKeyExists

Syntax

```
function BTKeyExists, (IFBPtr : IsamFileBlockPtr, Key : Word,  
    UserDatRef : LongInt;  
    UserKey : IsamKeyStr) : Boolean;
```

Purpose

Determine the existence of a particular key.

Description

This function searches for the key UserKey of index number Key. True is returned if the combination of UserKey and UserDatRef is found in the index, otherwise False.

If you are searching a primary index, the value of UserDatRef is not used by BTKeyExists.

If you are searching a secondary index and know the data reference number of the desired key, specify its value in the UserDatRef parameter. If you don't know the data reference number, specify zero for UserDatRef. In this case, BTKeyExists returns True if any key with the value UserKey, regardless of data reference number, exists in the index.

Note that this function is not for obtaining the data reference of a key; use BTFindKey for that purpose. The advantage of BTKeyExists is that it does not disturb the internal sequential pointer used by BTNextKey and BTPrevKey. For this reason and for its slightly faster speed, BTKeyExists may be the preferred routine for certain applications.

Example

Suppose you want to find all people in zip code 95060 (Santa Cruz) whose last name doesn't also occur in the zip code area 95066 (Scotts Valley). BTKeyExists can be used to formulate a solution as follows:

```
var  
    Person    : PersonDef;  
    Name,  
    KeyStr1,  
    KeyStr2 : IsamKeyStr;  
    Ref1,  
    Ref2    : LongInt;  
    Match    : Boolean;  
  
...  
KeyStr1 := '95060';  
BTSearchKey(PF, 2, Ref1, KeyStr1); {Index 2 is zip code}  
while IsamOK and (KeyStr1 = '95060') do begin  
    BTGetRec(PF, Ref1, Person, False);  
    if IsamOK then begin  
        Name := Copy(CreateKey(Person, 1), 1, 20);  
        KeyStr2 := Name;  
        BTSearchKey(PF, 1, Ref2, KeyStr2); {Index 1 is last name}  
    end;  
    Match := False;  
    while IsamOK and not Match and (Name = Copy(KeyStr2, 1, 20)) do  
begin  
    Match := BTKeyExists(PF, 2, Ref2, '95066');  
    if IsamOK and not Match then  
        BTNextKey(PF, 1, Ref2, KeyStr2);  
end;  
If BTIsamErrorClass < 2 then begin  
    IsamClearOK;  
    if not Match then begin  
        {Print out the data record}  
    end;  
end;  
end;
```


BTKeyExists

```
if IsamOK then
  BTNextKey(PF, 2, Ref1, KeyStr1);
end;
if BTIsamErrorClass > 1 then begin
  {Error handling}
end;
```

This example consists of two embedded while loops. The outer loop sequentially scans all records in zip code 95060. The inner loop obtains all data record references that correspond to the people who have the same last name. If one of these people lives in zip code 95066, the previously found person is not printed out.

At the point where BTKeyExists is used, there are two other ways to obtain the same information. Both of them have drawbacks. The first alternative is to read the data record with reference Ref2 and check the ZipCode field for '95066'. This requires an additional record variable and an additional data file access. The additional file access is unlikely with BTKeyExists because the corresponding page of the index tree is probably already in a buffer.

The second possibility is to use the procedure BTFindKeyAndRef, which is just a little slower than BTKeyExists. This approach is undesirable because the sequential pointer of index number 2 is changed. Consequently, the call to BTNextKey in the outer while loop would return the wrong value. To compensate you need to call BTFindKeyAndRef just prior to BTNextKey to restore the sequential pointer to its original position. This costs additional time.

See Also

BTFindKey
BTSearchKey

BTFindKeyAndRef
BTSearchKeyAndRef

BTKeyRecordSize

Syntax

```
function BTKeyRecordSize (IFBPtr : IsamFileBlockPtr) : LongInt;
```

Purpose

Determine the size of a page block.

Description

When adding a key, BTAddKey will eventually need to allocate a new page on the disk (when a previous index page is filled and must be split). The amount of additional disk space for one new index page is returned by BTKeyRecordSize. The same value also determines the minimum size for a newly created index file.

The value doesn't change once a fileblock is open. Therefore there is no need to determine this value more than once.

The value returned by BTKeyRecordSize is determined using the following calculation:

```
var
  K : Word;
  KRS : LongInt;
begin
  KRS := 0;
  for K := 1 to BTNrOfKeys (IFBPtr) do
    inc (KRS, LongInt (KeyLen[K]+9) * CreatePageSize+6);
  BTKeyRecordSize := KRS;
end;
```

where KeyLen[K] is the maximum key length (as originally specified in the IsamIndDescr passed to BTCreateFileBlock) for index number K. For example, if a fileblock had 5 indexes, each accepting keys up to 19 characters long (plus length byte), and CreatePageSize had its standard value of 62, BTKeyRecordSize would be 8710 bytes.

This function can also be used to estimate the size of an index file. Assuming that one key string is added for each index for each record, and that each index page is 75% full on average, the expected size of the index file is:

```
IndexSize := (1.5*BTUsedRecs (IFBPtr) *BTKeyRecordSize (IFBPtr))
div CreatePageSize;
```

Example

```
if DiskFree(1) < BTKeyRecordSize(PF) then
  Writeln('Addition of a key may not be possible');
BTAddKey(PF, 1, Ref, KeyStr);
```

The destination diskette is checked before a key is added to see whether enough space is available for the worst case expansion of the index file. Note that BTAddKey and other B-Tree Filer routines will detect an error if space runs out anyway.

See Also

BTDatRecordSize

BTMinimumDatKeys

BTLockAllOpenFileBlocks

Syntax

```
procedure BTLockAllOpenFileBlocks;;
```

Purpose

Reserve all open network fileblocks for exclusive write access.

Description

All open network fileblocks are locked (level 1 lock). This or a call to BTLockFileBlock is a prerequisite for modifying a network fileblock. Following this call, no other workstation can read or write these fileblocks, except by calling BTGetRecReadOnly or by calling BTGetRec or BTPutRec with parameter ISOLock set to True.

Calling this routine more than once is not an error. The network fileblocks remain locked. All open network fileblocks are unlocked if BTLockAllOpenFileBlocks fails for any reason.

Example

```
BTLockAllOpenFileBlocks;  
if BTIsamErrorClass = 2 then begin  
    Writeln('Locking attempt failed.');
```

```
end;
```

An error class of 2 means that the fileblock lock couldn't be obtained, possibly because another workstation already has a lock on a file.

See Also

BTLockFileBlock

BTUnlockAllOpenFileBlocks

BTReadLockAllOpenFileBlocks

BTLockFileBlock

Syntax

```
procedure BTLockFileBlock; (IFBPtr : IsamFileBlockPtr);
```

Purpose

Reserve a network fileblock for exclusive write access.

Description

The network fileblock is locked (level 2 lock). This or a call to BTLockAllOpenFileBlocks is a prerequisite for modifying a network fileblock with any of the following routines:

- BTAddKey
- BTAddRec
- BTDeleteKey
- BTDeleteRec
- BTPutRec

Following this call, no other workstation can read or write the fileblock, except by calling BTGetRecReadOnly or by calling BTGetRec or BTPutRec with parameter ISOLock set to True.

Calling this routine more than once is not an error. The network fileblock remains locked. If a fileblock was read locked prior to calling BTLockFileBlock, the read lock is converted to a write lock. If BTLockFileBlock fails, however, the original read lock is lost.

A fileblock write lock is implemented by a physical lock over a segment of the dialog file whose extent is proportional to the maximum number of workstations on the network. A read lock is placed by locking just the current workstation's segment of this same region of the dialog file. As a result, a read lock by another station prevents a write lock. Similarly, a write lock prevents any other station from placing a read lock.

B-Tree Filer performs automatic retry when placing fileblock locks. The retry is controlled by the IsamLockTimeout, IsamFBlockTimeoutFactor, and IsamDelayBetwLocks variables discussed early in this chapter.

Care must be taken if more than one fileblock needs to be locked simultaneously to perform a transaction, since it is easy for two workstations to enter a "fatal embrace." It's often better to call BTLockAllOpenFileBlocks in this case, since that routine guarantees that all of the fileblocks are locked, or none of them. See _4.F for more information.

To achieve good multi-user performance it is essential to minimize the amount of time that fileblocks remain locked. Again, see _4.F.

Example

```
BTLockFileBlock(PF);  
if BTIsamErrorClass = 2 then begin  
  Writeln('Locking attempt failed.');
```

```
end;
```

An error code class of 2 means that the fileblock lock couldn't be obtained, possibly because another workstation already has a lock on the file.

See _4.F for a more complete example.

See Also

BTLockAllOpenFileBlocks
BTUnlockFileBlock

BTReadLockFileBlock

BTLockRec / BTMinimumDatKeys

Syntax

```
procedure BTLockRec; (IFBPtr : IsamFileBlockPtr; Ref : LongInt);
```

Purpose

Reserve the specified data record for exclusive access.

Description

The data record with reference Ref in the data file of the network fileblock is locked (level 3 lock). Following this call, no other workstation can read or write the record, with the one exception of calling BTGetRecReadOnly to read all but the first four bytes of the record.

This procedure is provided primarily to allow writing an edited record back to the data file without placing a fileblock lock. To do so, call BTLockRec for the record to be modified, then call BTPutRec with the parameter ISOLock set to True. In this way, the record can be modified even if another workstation has a fileblock lock in force, with the guarantee that no other workstation is simultaneously modifying the same record. The entire fileblock must always be locked in order to modify the index file. Be aware that using record locks in this way complicates logic throughout the program.

B-Tree Filer maintains an internal list of locked records. As a result, BTLockRec must sometimes allocate a small block of heap space (21 bytes for the default settings.) It returns error 10337 if unable to do so.

Calling this routine multiple times for the same Ref is not an error. B-Tree Filer recognizes the duplicate locking attempt and does not call the operating system to place the lock again.

Example

See _4.F for an example.

See Also

BTLockFileBlock
BTUnlockRec

BTReadLockFileBlock

Syntax

```
function BTMinimumDatKeys; (IFBPtr : IsamFileBlockPtr; Space : LongInt  
 ) : LongInt;
```

Purpose

Determine the minimum number of records and keys that fit into the specified disk space.

Description

This function determines how many data records and keys can be written to a storage device with Space bytes left. The worst case scenario for filling of the index tree is used (i.e., index pages are assumed to be only half full). It is therefore possible that up to 70% more data records with their keys can be written. The routine assumes that each data record will have one key per index.

BTMinimumDatKeys does not access the disk, so it can be expected not to fail.

If more data records and their keys are to be written than BTMinimumDatKeys returns, BTDatRecordSize and BTKeyRecordSize can be used to test if this can safely be done.

Although this function is primarily useful for diskette drives, it may also be useful for hard drives. For example, it can determine at the onset how many data records with keys will definitely fit on a 20MB hard drive.

Example

```
Writeln('On drive C: at least: ',  
      BTMinimumDatKeys(PF, DiskFree(3)),  
      ' data records with keys can be stored.');
```

BTLockRec / BTMinimumDatKeys

See Also

~~BTDataRecordSize~~ ~~BTKeyRecordSize~~

BTNetSupported

Syntax

```
function BTNetSupported; : NetSupportType;
```

Purpose

Return the currently initialized network.

Description

This function allows you to determine which network was initialized through the function BTInitIsam. This is primarily useful for showing status.

BTNetSupported returns NoNet when BTDEFINE.INC defines the compiler directive NoNet and also when the ExpectedNet parameter passed to BTInitIsam is equal to NoNet. Use the function BTNoNetCompiled to differentiate between these two cases.

BTNetSupported can also be used as a debugging tool. If the net passed to BTInitIsam is not returned by BTNetSupported, this means the network wasn't successfully initialized. This can occur if the setting in BTDEFINE.INC is incorrect.

Example

```
const
  NetSupportString : array[NetSupportType] of String[7] =
    ('NoNet', 'Novell', 'MsNet');

...
  Writeln('The current network is ',
    NetSupportString[BTNetSupported]);
```

The return value of BTNetSupported can be used as an index into an array of strings.

See Also

BTInitIsam
BTNoNetCompiled

BTIsNetFileBlock

BTNextDiffKey

Syntax

```
procedure BTNextDiffKey; (IFBPtr : IsamFileBlockPtr; Key : Word;  
                        var UserDatRef : LongInt; var UserKey :  
                        IsamKeyStr);
```

Purpose

Search for the next larger key that differs from the specified key.

Description

BTNextDiffKey searches index Key of the index file for the next key after UserKey that differs from it. The data reference number is returned in UserDatRef and the differing key string is returned in UserKey. The smallest data reference is returned for a secondary key that occurs more than once in the index file. If no such key exists (UserKey is larger than or equal to the largest key), the search fails and IsamError 10240 is returned.

This procedure is especially useful to jump over a number of identical secondary keys and to obtain the next key, without performing a sequence of BTNextKey calls.

BTNextDiffKey sets the sequential pointer upon successful completion, so that the procedures BTNextKey and BTPrevKey can be used immediately after.

Example

```
var  
  KeyStr : IsamKeyStr;  
  Ref    : LongInt;  
...  
KeyStr := '';  
repeat  
  BTNextDiffKey(PF, 2, Ref, KeyStr);  
  if IsamOK then  
    Writeln(KeyStr);  
until not IsamOK;  
if BTIsamErrorClass > 1 then begin  
  {Error handling}  
end;
```

This small piece of code reports all unique keys that are available in index 2.

See Also

BTNextKey

BTPrevDiffKey

BTNextKey

Syntax

```
procedure BTNextKey; (IFBPtr : IsamFileBlockPtr; Key : Word;  
                    var UserDatRef : LongInt; var UserKey :  
                    IsamKeyStr);
```

Purpose

Obtain the next key in ascending sequence.

Description

BTNextKey advances one key further in index Key of the fileblock's index file. The next key string is returned in UserKey along with its corresponding data reference in UserDatRef. Commonly, BTFindKey or BTSearchKey is used to find a certain key and then BTNextKey is called to find all keys greater than the first until some condition is met or no more keys exist.

If there is no next key available, IsamError is set to 10250.

The functionality of BTNextKey is dependent upon the validity of the internal sequential pointer. If the sequential pointer is valid, the next key is returned independent of the initial values of the parameters UserKey and UserDatRef. In this case both of these var parameters can be passed uninitialized to BTNextKey.

If the sequential pointer was disturbed--by calling BAddKey or BDeleteKey or if another workstation called the same routines--the action of BTNextKey is dependent on the state of the SearchForSequential option for index Key of the fileblock. If the option is disabled, BTNextKey returns immediately with IsamError 10255. However, if the option is enabled (as it is by default), the parameters are used to reestablish the sequential pointer by calling BTFindKey. Then the next key is determined as usual. Hence, when calling BTNextKey in a loop, the values of UserKey and UserDatRef returned by the previous iteration should be passed into BTNextKey to enable automatic repositioning.

The internal pointer is valid after calls to the following procedures: BClearKey, BTSearchKey, BTFindKey, BTNextDiffKey, BTPrevDiffKey, BTFindKeyAndRef and BTSearchKeyAndRef.

Example

```
var  
  KeyStr : IsamKeyStr;  
  Ref    : LongInt;  
  Person : PersonDef;  
...  
KeyStr := '96000';  
BTSearchKey(PF, 2, Ref, KeyStr);  
while IsamOK and (KeyStr <= '96999') do begin  
  BGetRec(PF, Ref, Person);  
  if IsamOK then begin  
    {display information about the Person}  
    BTNextKey(PF, 2, Ref, KeyStr);  
  end;  
end;  
if BTIsamErrorClass > 1 then begin  
  {Error handling}  
end;
```

This is a typical use of the procedure BTNextKey. All persons who have zip codes between 96000 and 96999 are displayed.

In a network fileblock for which the SearchForSequential option is enabled, this example automatically corrects itself if another workstation adds or deletes a key during the while loop. That's because the Ref and KeyStr variables returned by one call to BTNextKey are passed into the next call.

See Also

BDeleteKey

BTPrevKey

BTNextRecRef / BTNoCharConvert

Syntax

```
procedure BTNextRecRef; (IFBPtr : IsamFileBlockPtr; var UserDatRef :  
    LongInt);
```

Purpose

Return the next active record.

Description

This procedure searches forward sequentially through the records in the data file, starting at UserDatRef+1, until an active record is found or the end of file is encountered. If an active record is found, its reference number is returned in UserDatRef. If the end of file is encountered without finding an active record, IsamError is set to 10275.

If a locked record is found during the search, IsamError is set to 10390 and the locked record reference number is returned in UserDatRef. Calling BTNextRecRef immediately with that value of UserDatRef causes the locked record to be skipped. If however you would like to try that record again, decrement UserDatRef by one and call BTNextRecRef again after a small delay.

Example

```
var  
    DatRef : LongInt;  
...  
    DatRef := 0;  
    while IsamOK do begin  
        BTNextRecRef(IFB, DatRef);  
        if IsamOK then  
            {do something with record DatRef}  
    end;
```

Reads sequentially through the IFB fileblock's data file and does some processing for every non-deleted record.

See Also

BTFindRecRef

BTPrevRecRef

Syntax

```
procedure BTNoCharConvert; (DataPtr : Pointer; DataLen : LongInt;  
    PostRead : Boolean; HookPtr : Pointer);
```

Purpose

Provide a record conversion routine that does nothing.

Description

Procedures of type ProcBTCharConvert (of which this is one) are designed to provide support for different external and internal representations of data records, in particular multiple code pages. BTNoCharConvert does nothing; the record pointed to by DataPtr is not altered.

This procedure can be passed to a call to RestructFileBlock (in the RESTRUCT unit) if you do not want any character conversion to take place.

For information on code page support see the CCSKEYS bonus unit.

See Also

RestructFileBlock (RESTRUCT)

BTNoNetCompiled / BTNrOfKeys

Syntax

```
function BTNoNetCompiled; : Boolean;
```

Purpose

Determine whether B-Tree Filer was compiled with the NoNet define.

Description

This function allows you to determine at runtime whether the FILER unit was compiled with the compiler directive NoNet (True) or not (False). Use it in combination with BTNetSupported if your application needs to determine its network capabilities.

Example

```
const
  NetSupportString : array[NetSupportType] of String[7] =
    ('NoNet', 'Novell', 'MsNet');
...
  Writeln('The current network is ',
    NetSupportString[BTNetSupported]);
  if BTNetSupported = NoNet then
    if BTNoNetCompiled then
      Writeln('FILER was compiled with the NoNet define')
    else
      Writeln('FILER was initialized for network emulation');
```

The example from BTNetSupported was expanded upon here, in order to differentiate the return value of NoNet.

See Also

BTInitIsam
BTNetSupported

BTIsNetFileBlock

Syntax

```
function BTNrOfKeys; (IFBPtr : IsamFileBlockPtr) : Word;
```

Purpose

Return the number of indexes in the fileblock.

Description

This function allows you to determine at runtime how many indexes were defined when the fileblock was created with BTCreateFileBlock. This is generally useful for informational purposes, but also can be part of general-purpose shells around the B-Tree Filer routines.

Example

```
var
  NumberKeys,
  KeyNr      : Word;
...
  NumberKeys := BTNrOfKeys(PF);
  if NumberKeys > 0 then begin
    Writeln('Please choose index number to Browse from 1 to ',
      NumberOfKeys);
    Readln(KeyNr);
  end else
    Writeln('No index is available to browse.');
```

This shows how BTNrOfKeys can be used to determine the maximum number of indexes and how to use it for a general purpose querying routine.

Also see the examples in _4.C.

BTNoNetCompiled / BTNrOfKeys

See Also

~~BTCreateFileBlock~~

BTOpenFileBlock

Syntax

```
procedure BTOpenFileBlock; (var IFBPtr : IsamFileBlockPtr;  
                           FName : IsamFileBlockName;  
                           ReadOnly, AllReadOnly, Save, Net :  
                           Boolean);
```

Purpose

Open an existing fileblock.

Description

All B-Tree Filer routines that deal with a fileblock can be called only after the fileblock has been opened with BTOpenFileBlock.

The parameter FName contains up to three DOS filenames without extensions and separated by semicolons. The third one is ignored since it has no meaning to BTOpenFileBlock. See the discussion of type IsamFileBlockName at the beginning of this chapter for more information about how to specify FName.

Four boolean parameters determine the mode in which the fileblock is opened. The fileblock is opened for reading only if ReadOnly has a value of True. All attempts to perform a write operation with the fileblock will result in a locking error. Retry attempts will never succeed in this case. If an application allows certain users write access and other users read access only, it must be carefully coded so that the read-only users are not confused by error messages that report locks.

There is another consideration to be made in a network environment if you set ReadOnly to True. B-Tree Filer uses the dialog file as an inter-workstation communications device on a network, for one workstation to inform all the other workstations of changes to an index of the fileblock (by adding or deleting a key). It does this by altering some bytes in the dialog file; each workstation will have its own set of bytes. In normal operation, a workstation will notice that 'its' bytes have changed, reset them and then reread the index pages it has in memory. In ReadOnly mode, a workstation cannot reset its dialog file bytes: it would entail writing to the dialog file. The upshot is that the workstation will continually have to reread its index pages for the fileblock, resulting in a much slower operation.

If AllReadOnly is True, a special situation that only makes sense for a network environment occurs. This mode specifies that all workstations have only read access (i.e., that no workstation can modify the fileblock). In this way, B-Tree Filer's internal buffers are guaranteed to remain valid at all times. As a result, internal operations that check and maintain the validity of these buffers do not have to be executed, resulting in faster execution. It is *essential* for the application to ensure that if any workstation sets AllReadOnly to True, then all workstations set AllReadOnly to True. Any one workstation writing to the fileblock will result in very confused reader workstations. The AllReadOnly option is provided primarily to optimize access to a read only drive such as a CD-ROM and should be set to True only for similar situations.

When Save is set to True, the fileblock is opened in save mode. This activates a data integrity enhancement scheme that is described more in _4.A and _4.G.

If Net is True, the fileblock is opened as a network fileblock. Please note the requirements listed at the beginning of this chapter for successfully opening a network fileblock.

These four boolean parameters are not totally independent. Therefore the following automatic corrections are made in the passed arguments. If AllReadOnly is True, ReadOnly is also set to True. If ReadOnly is True, then Save is set to False (no write operations can be performed). Net is always set to False if the FILER unit was compiled with the compiler directive NoNet.

BTOpenFileBlock allocates memory from the heap in order to store certain data. The amount of heap space can be calculated as follows:

$$82 * (\text{NumberOfKeys} + 1) + 234$$

This memory does not need to be one contiguous chunk. It is actually used in multiple small pieces. An additional 37 bytes are required if the fileblock was opened with the parameter Net set to True.

BTOpenFileBlock

Examples

```
BTOpenFileBlock(PF, FName, False, False, False, False);
```

This fileblock is opened for reading and writing without a network. Save mode is disabled.

```
BTOpenFileBlock(PF, FName, False, False, True, True);
```

This fileblock is opened for reading and writing on a network. Save mode is enabled.

```
BTOpenFileBlock(PF, FName, True, False, False, True);
```

This fileblock is opened only for reading on a network. Save mode is disabled.

```
BTOpenFileBlock(PF, FName, True, True, False, True);
```

This fileblock is opened only for reading on a network. In addition, all workstations on the network have only read access. Save mode is disabled.

```
BTOpenFileBlock(PF, FName, False, True, True, True);
```

This fileblock is opened only for reading on a network (automatic correction of ReadOnly). In addition, all workstations in the network have only read access. Save mode is disabled (automatic correction of Save).

See Also

[BTCloseFileBlock](#)

BTOtherWSChangedKey

Syntax

```
function BTOtherWSChangedKey; (IFBPtr : IsamFileBlockPtr; Key : Word)  
: Boolean;
```

Purpose

Return True if another workstation modified the specified index.

Description

This function is useful in combination with a file browser. If BTOtherWSChangedKey returns True, the application can automatically update the screen to show the changes of other workstations without wasting too much effort.

BTOtherWSChangedKey is a relatively fast routine: it places a temporary read lock and reads a short area in the dialog file from which it can determine whether a change occurred. If BTOtherWSChangedKey is unable to obtain the read lock, or if an error occurs while reading the dialog file, it returns False.

Once BTOtherWSChangedKey returns True, it will continue to return True until a B-Tree Filer routine is called that reloads the page index buffers from the disk file. Such routines include BTFindKey, BTFindKeyAndRef, and other key access routines.

Note that BTOtherWSChangedKey cannot be used to detect changes to the *data* file made by other workstations.

Example

```
while not KeyPressed do begin  
  if BTOtherWSChangedKey(PF, 1) then begin  
    FastWrite('Changed', 25, 70, $70);  
    Delay(1000);  
    BTFindKeyAndRef(PF, 1, CurDatRef, CurUserKey, 1);  
  end else  
    FastWrite('      ', 25, 70, $70);  
end;
```

Maintains a simple status line on row 25 that displays a message for one second whenever another workstation modifies index 1. This routine would be called whenever an application waited for a keypress. CurDatRef and CurUserKey must contain values for a known record.

BTPrevDiffKey

Syntax

```
procedure BTPrevDiffKey; (IFBPtr : IsamFileBlockPtr; Key : Word;  
                        var UserDatRef : LongInt; var UserKey :  
                        IsamKeyStr);
```

Purpose

Search for the next smaller key that differs from the specified key.

Description

BTPrevDiffKey searches index Key of the index file for the last key before UserKey that differs from it. The data reference number is returned in UserDatRef and the differing key string is returned in UserKey. The largest data reference is returned for a secondary key that occurs more than once in the index file. If no such key exists (UserKey is smaller than or equal to the smallest key), the search fails and IsamError 10245 is returned.

This procedure is especially useful to jump over a number of identical secondary keys and to obtain the previous key, without performing a sequence of BTPrevKey calls.

BTPrevDiffKey sets the sequential pointer upon successful completion, so that the procedures BTNextKey and BTPrevKey can be used immediately after.

Example

```
var  
  KeyStr : IsamKeyStr;  
  Ref    : LongInt;  
...  
KeyStr := #255;  
repeat  
  BTPrevDiffKey(PF, 2, Ref, KeyStr);  
  if IsamOK then  
    Writeln(KeyStr);  
until not IsamOK;  
if BTIsamErrorClass > 1 then begin  
  {Error handling}  
end;
```

Reports all unique keys that are available in index 2, in descending order.

See Also

BTNextDiffKey

BTPrevKey

BTPrevKey

Syntax

```
procedure BTPrevKey; (IFBPtr : IsamFileBlockPtr; Key : Word;  
                    var UserDatRef : LongInt; var UserKey :  
                    IsamKeyStr);
```

Purpose

Obtain the previous key.

Description

BTPrevKey moves one key back in index Key of the fileblock's index file. The previous key string is returned in UserKey along with its corresponding data reference in UserDatRef. Commonly, either BTFindKey or BTSearchKey is used to find a certain key and then PrevKey is called to find all keys less than the first until some condition is met or no more keys exist.

If there is no previous key available, IsamError is set to 10260.

BTPrevKey is dependent upon the validity of the internal sequential pointer. If the sequential pointer is valid, the next key is returned independent of the initial values of the parameters UserKey and UserDatRef. In this case both of these var parameters passed to BTPrevKey can be uninitialized.

If the sequential pointer has been disturbed--by calling BTAddKey or BTDeleteKey or if another workstation called the same routines--the action of BTPrevKey is dependent on the state of the SearchForSequential option for index Key of the fileblock. If the option is disabled, BTPrevKey returns immediately with IsamError 10265. However, if the option is enabled (as it is by default), the parameters are used to reestablish the sequential pointer by calling BTFindKeyAndRef. Then the previous key is determined as usual. Hence, when calling BTPrevKey in a loop, the values of UserKey and UserDatRef returned by the previous iteration should be passed into BTPrevKey to enable automatic repositioning.

The internal pointer is valid after calls to the following procedures: BTClearKey, BTSearchKey, BTFindKey, BTNextDiffKey, BTPrevDiffKey, BTFindKeyAndRef and BTSearchKeyAndRef.

Example

```
var  
  KeyStr : IsamKeyStr;  
  Ref    : LongInt;  
  Person : PersonDef;  
...  
KeyStr := '96000';  
BTSearchKey(PF, 2, Ref, KeyStr);  
while IsamOK and (KeyStr >= '95000') do begin  
  BTGetRec(PF, Ref, Person);  
  if IsamOK then begin  
    {Display information about Person}  
    BTPrevKey(PF, 2, Ref, KeyStr);  
  end;  
end;  
if BTIsamErrorClass > 1 then begin  
  {Error handling}  
end;
```

This is a typical use of the procedure BTPrevKey. All persons who have zip codes between 96000 and 95000 are displayed in descending order.

In a network fileblock for which the SearchForSequential option has been enabled, this example automatically corrects itself if another workstation adds or deletes a key during the while loop. That's because the Ref and KeyStr variables returned by one call to BTPrevKey are passed into the next call.

BTPrevKey

See Also

[BTPrevDiffKey](#) [BTSearchKey](#)
[BTNextKey](#)

BTPrevRecRef / BTPutRec

Syntax

```
procedure BTPrevRecRef; (IFBPtr : IsamFileBlockPtr; var UserDatRef :  
    LongInt);
```

Purpose

Return the preceding active record.

Description

This procedure searches backward sequentially through the records in the data file, starting at UserDatRef-1, until an active record is found or the start of the file is encountered. If an active record is found, its reference number is returned in UserDatRef. If the start of the file was encountered without finding an active record, IsamError is set to 10275.

If a locked record was found during the search, IsamError is set to 10390 and the locked record reference number is returned in UserDatRef. Calling BTPrevRecRef immediately with that value of UserDatRef causes the locked record to be skipped. However, if you need to access that record again, increment UserDatRef by one and call BTPrevRecRef after a small delay.

See Also

BTFindRecRef

BTNextRecRef

Syntax

```
procedure BTPutRec; (IFBPtr : IsamFileBlockPtr; RefNr : LongInt; var  
    Source;  
                    ISOLock : Boolean);
```

Purpose

Write a data record back to the data file.

Description

RefNr is usually retained from a previous BTGetRec operation. BTPutRec is used to write an edited record back to the same data file location it came from.

The data to be written is passed to BTGetRec via the Source parameter. Since Source is untyped, any desired data structure can be passed as a parameter.

When ISOLock is False, this procedure operates successfully in a network environment only if the fileblock is locked for exclusive write access (a level 1 or 2 lock). When ISOLock is True, the data record is written regardless of whether the fileblock is locked by either the calling workstation or any other workstation. Only a record lock (level 3) placed by another workstation can prevent writing the data record in this case.

Warning: Never add a new data record to the file with BTPutRec. Use BTAddRec to do that! Use BTPutRec only to update an existing record.

An attempt to update record 0 with BTPutRec generates IsamError 10130. B-Tree Filer reserves record 0 for its own internal information.

Example

See _4.C for an example.

See Also

BTAddRec

BTGetRec

BTReadLockAllOpenFileBlocks

Syntax

```
procedure BTReadLockAllOpenFileBlocks;;
```

Purpose

Prevent all workstations from writing to open network fileblocks.

Description

This routine places a read lock on all network fileblocks that the calling workstation currently has open. As a result, no other workstation can write to the fileblocks. (The only exception is that another station can still update individual data records by calling BTPutRec with ISOLock set to True.) All workstations can continue to read from the fileblocks as usual. Any number of workstations can simultaneously place read locks on the same fileblocks.

It is not possible to read lock any fileblock if another workstation already has an exclusive write lock. IsamError will unlock all open fileblocks and return 10332 in this case.

BTReadLockAllOpenFileBlocks converts any write locks already obtained by the calling workstation into read locks, effectively removing the write locks. Remove read locks by calling BTUnlockAllOpenFileBlocks or BTUnlockFileBlock.

Example

```
BTReadLockAllOpenFileBlocks;  
if BTIsamErrorClass = 2 Then  
  Writeln('Read lock attempt failed.');
```

A class 2 error code means that one of the read locks could not be completed because another workstation already had a fileblock write lock.

See Also

BTLockAllOpenFileBlocks
BTUnlockAllOpenFileBlocks

BTReadLockFileBlock

BTReadLockFileBlock

Syntax

```
procedure BTReadLockFileBlock; (IFBPtr : IsamFileBlockPtr);
```

Purpose

Prevent another workstation from writing to the specified fileblock.

Description

This routine read locks the specified fileblock, which means that no other workstation can write to it. (The only exception is that another station can still update individual data records by calling BTPutRec with ISOLock set to True.) All workstations can continue to read from the fileblock as usual. Any number of workstations can simultaneously place read locks on the same fileblock.

It is not possible to read lock a fileblock if another workstation already has an exclusive write lock. IsamError returns 10332 in this case.

Redundant calls to BTReadLockFileBlock are not an error. The fileblock remains read locked.

BTReadLockFileBlock converts any write locks already obtained by the calling workstation into read locks, effectively removing the write locks. Remove a read lock by calling BTUnlockFileBlock.

B-Tree Filer uses temporary read locks internally to assure that index buffers remain stable while completing a sequence of operations. These internal read locks are placed and removed only if the fileblock is not already locked for reading or writing. An application can therefore achieve higher performance if it places its own read locks prior to a series of B-Tree Filer key operations. As with all locks, however, it is important to minimize the total time that the lock is maintained. Remember that no other workstation can write to the fileblock as long as the read lock remains active.

Example

```
BTReadLockFileBlock(PF);  
if BTIsamErrorClass = 2 then  
  Writeln('Read lock attempt failed');
```

A class 2 error code means that the read lock could not be completed because another workstation already had a fileblock write lock.

See Also

BTLockFileBlock
BTUnlockFileBlock

BTReadLockAllOpenFileBlocks

BTRecIsLocked / BTSearchKey

Syntax

```
function BTRecIsLocked, (IFBPtr : IsamFileBlockPtr, Ref : LongInt) :  
    Boolean;
```

Purpose

Return True if the specified record is locked.

Description

B-Tree Filer maintains an internal list of the records locked by each workstation. BTRecIsLocked returns True only if the current workstation locked the record. The only way to determine whether another workstation locked a record is to attempt to lock it from the current station and to check for success.

See Also

BTaRecIsLocked

BTLockRec

Syntax

```
procedure BTSearchKey, (IFBPtr : IsamFileBlockPtr; Key : Word;  
    var UserDatRef : LongInt; var UserKey :  
    IsamKeyStr);
```

Purpose

Search for nearest matching key.

Description

The index file is searched for the key UserKey in the index Key. The matching key string is returned in UserKey and the data record reference is returned in UserDatRef. If an exact match is not found, the next larger key is returned in UserKey along with its associated record number in UserDatRef. The search fails and IsamError 10210 is returned only when the initial UserKey is larger than all other keys.

If a secondary key that is entered more than once in the index file is found, the smallest associated data record reference is returned in UserDatRef.

BTSearchKey sets the sequential pointer upon successful execution, so that the procedures BTPrevKey and BTNextKey can be used immediately. BTSearchKey is implemented by first calling BTFindKey, and then BTNextKey if the first call was unsuccessful.

Example

See _4.C and _4.F as well as the example for BTNextKey.

See Also

BTFindKey

BTNextKey

BTSearchKeyAndRef

BTSearchKeyAndRef

Syntax

```
procedure BTSearchKeyAndRef; (IFBPtr : IsamFileBlockPtr; Key : Word,  
                             var UserDatRef : LongInt; var UserKey :  
                             IsamKeyStr);
```

Purpose

Search for a specified key and data record reference.

Description

Index Key of the index file is searched for the key UserKey that also has the record number UserDatRef. If it is found, the sequential pointer is set to point to this entry. If it is not found and there are any larger keys, the next larger key and record number are returned. If there are no larger keys, the next smaller key is returned. IsamError 10260 is returned if no match is found. This occurs only if the index is empty.

Example

```
var  
  Ref, SRef : LongInt;  
  IKS, SIKS : IsamKeyStr;  
  PTemp     : PersonDef;  
  Stop      : Boolean;  
...  
BTFindKey(PF, 2, Ref, IKS);  
Stop := False;  
while IsamOK and not Stop do begin  
  BTGetRec(PF, Ref, PTemp, False);  
  if IsamOK then  
    If PTemp.Age >= 18 then begin  
      SRef := Ref;  SIKS := IKS;  Stop := True;  
    end else  
      BTNextKey(PF, 2, Ref, IKS);  
  end;  
  if BTIsamErrorClass > 1 then begin  
    {Error handling}  
  end;  
  ...  
  BTSearchKeyAndRef(PF, 2, SRef, SIKS, 0);  
  if not IsamOK then begin  
    {Error handling}  
  end;  
end;
```

The example first searches for a specific key of index 2. The while loop then scans for the first data record that has an Age field greater than or equal to 18 and saves the key string and data reference number for this record. Finally BTSearchKeyAndRef attempts to set the sequential pointer on the data record that was found in the while loop. It will succeed in returning a nearby key even if the original record was deleted in the meantime by another workstation.

See Also

BTFindKey

BTSearchKey

BTSetCharConvert

Syntax

```
procedure BTSetCharConvert, (IFBPtr : IsamFileBlockPtr; CCProc :  
  ProcBTCharConvert;  
                                HookPtr : Pointer; DestrWrite : Boolean);
```

Purpose

Set the record conversion routine for a fileblock.

Description

Record conversion routines are called after a record is read from a fileblock and before a record is written to a fileblock. This interface enables you to convert a record from an external representation to an internal one (or vice versa). For example, you can convert text strings in a record from the calling system's code page to the code page of the fileblock.

This procedure sets the record conversion routine which will be used for the fileblock IFBPtr (which must be open). CCProc is a procedure of type ProcBTCharConvert that does the conversion. HookPtr is a pointer to any type of data structure; this pointer is passed unmodified to CCProc when a record is read or written.

When a record is written to the fileblock's data file, CCProc is called to convert the record in place (a copy of the record is not made). If, after the record is written, the data in the record buffer will not be used by the rest of the application, set DestrWrite to True. If the data in the record buffer will be used by the application after writing the record, set the DestrWrite parameter to False. In this case the CCProc routine is called a second time to convert the record back again to the internal format (as if the record had just been read).

For more information on code page support, see the CCSKEYS bonus unit.

BTSetDosRetry

Syntax

```
function BTSetDosRetry; (NrOfRetries, WaitTime : Integer) : Boolean;
```

Purpose

Set the number of retry attempts and the timeout for a read attempt aborted by a lock.

Description

This function does something only when the DOS file-sharing module SHARE.EXE is loaded or an equivalent network service is active. It requires DOS version 3.1 or later.

BTSetDosRetry calls DOS function \$440B to set the number of retries DOS should attempt when it detects that a file is locked during a read request. The delay between attempts is set to WaitTime "loop delays." A loop delay is equal to the time it takes for an assembly language LOOP instruction to run 65536 times. This delay varies from machine to machine.

SetDosRetry returns True unless the carry flag is set when the DOS function returns.

DOS does not restore the original values for NrOfRetries and WaitTime when a program ends. Generally, the application should restore the default system values (NrOfRetries = 3 and WaitTime = 1) before halting.

Experience shows that this function doesn't work as expected on every network. Some networks just ignore the retry values. Another problem is that the supplied values can be interpreted differently depending on the network, so that one might use timer ticks (1/18 of a second) and another might use the number of runs of an internal loop. However, this function is especially useful when using PC-MOS/386.

Example

The PC-MOS/386 defaults for retries and delay seem to be high. As a result, when BROWSER attempts to display a data record that another workstation has locked, the delay is annoyingly long. The following code is therefore recommended when initializing B-Tree Filer for PC-MOS/386:

```
if not BTSetDosRetry(1, 1) then begin
    {Error handling}
end;
```

BTSetSearchForSequential Syntax

```
procedure BTSetSearchForSequential, (IFBPtr : IsamFileBlockPtr, Key :  
Word;  
ToOn : Boolean);
```

Purpose

Set the SearchForSequential option for the specified index.

Description

This procedure activates or deactivates a special method for finding the next and previous keys from the current position in index Key of a fileblock. It primarily affects the behavior of the BTNextKey and BTPrevKey routines.

When you call BTNextKey (or BTPrevKey) the next key is computed based on the "current" position within the index. B-Tree Filer stores the current position by using a data structure referred to as the "sequential pointer." This is not a pointer in the usual sense of a physical memory address. Instead, the sequential pointer is an array of values that specify elements within a series of index pages, leading from the root page of the B-tree down to the page where the current key is located.

An alternate way to remember the current position would be to keep a copy of its key string and data reference number. B-Tree Filer doesn't use this approach. For one thing, in many cases the string would use more memory. More importantly, the sequential pointer approach is much faster because it retains exactly the information needed when time comes to move to the next or previous key; an additional search through the B-tree isn't required every time.

The disadvantage of the sequential pointer approach is that the pointer is specific to a particular configuration of the B-tree. If the B-tree is modified by adding or deleting a key, the numbers stored in the sequential pointer become meaningless. B-Tree Filer uses an internal flag to detect this condition. If you call BTNextKey or BTPrevKey in such a case, it returns with IsamError set to 10255 or 10265: sequential access not allowed.

For single user applications this behavior is acceptable, because the application knows exactly when it has disturbed the B-tree and can reinitialize the sequential pointer by calling BTFindKeyAndRef or a related key searching routine. In a network application, however, it is common for the sequential pointer to be corrupted by another workstation, which might call BTAddKey or BTDeleteKey at almost any time.

This is where the SearchForSequential option comes in. If the option is enabled and the flag indicating an invalid sequential pointer is set, BTNextKey automatically uses UserKey and UserDatRef (always passed to BTNextKey anyway) with BTFindKey to reinitialize the sequential pointer.

Whether the SearchForSequential option is enabled by default depends on the state of the global typed constant SearchForSequentialDefault interfaced by the FILER unit.

SearchForSequentialDefault defaults to True, enabling the special search method for all fileblocks and indexes. Use BTSetSearchForSequential to disable the option for specific indexes.

See Also

BTAddKey	BTDeleteKey
BTGetSearchForSequential	BTNextKey

BTUnlockAllOpenFileBlocks / BTUnlockAllRecs

Syntax

```
procedure BTUnlockAllOpenFileBlocks;;
```

Purpose

Unlock all open fileblocks.

Description

This procedure unlocks all open network fileblocks. This reverses the action of BTLockAllOpenFileBlocks or BTReadLockAllOpenFileBlocks. At the same time, all new data of all fileblocks locked for writing is flushed to disk.

Calling this procedure repeatedly is not an error. The network fileblocks remain unlocked. If an unlock fails for one of the fileblocks, an attempt is still made to unlock all others.

Example

```
BTUnlockAllOpenFileBlocks;  
if BTIsamErrorClass <> 0 then begin  
    WriteLn('Hard Error: Unlock attempt failed.');
```

{A more exact failure analysis could occur here.}

```
end;
```

If BTUnlockAllOpenFileBlocks returns an error code, it is always a severe error. Either data could not be written to disk, or a physical lock could not be removed.

See Also

BTLockAllOpenFileBlocks BTReadLockAllOpenFileBlocks
BTUnlockFileBlock

Syntax

```
procedure BTUnlockAllRecs;(IFBPtr : IsamFileBlockPtr);
```

Purpose

Unlock all locked records of a fileblock.

Description

B-Tree Filer maintains an internal list of the records locked by each workstation.

BTUnlockAllRecs scans this list and releases each lock. If an error occurs while releasing one lock, it continues to release the rest.

BTUnlockAllRecs removes only the locks placed by the current workstation, not by other workstations.

See Also

BTaRecIsLocked BTUnlockRec

BTUnlockFileBlock / BTUnlockRec

Syntax

```
procedure BTUnlockFileBlock, (IFBPtr : IsamFileBlockPtr);
```

Purpose

Unlock a fileblock.

Description

The network fileblock is unlocked. This reverses the action of BTLockFileBlock or BTReadLockFileBlock. If the fileblock was locked for writing, all new data is flushed to disk.

Calling this procedure repeatedly is not an error. The fileblock remains unlocked.

Example

```
BTUnlockFileBlock(PF);  
if BTIsamErrorClass <> 0 then begin  
  Writeln('Hard Error: Unlock attempt failed.');
```

{A more exact failure analysis could occur here.}

```
end;
```

If BTUnlockOpenFileBlock returns an error code, it is always a severe error. Either data could not be written to disk, or the physical lock could not be removed.

See _4.F for a more detailed example.

See Also

BTLockFileBlock

BTReadLockFileBlock

Syntax

```
procedure BTUnlockRec; (IFBPtr : IsamFileBlockPtr; Ref : LongInt);
```

Purpose

Remove a lock from a record.

Description

A record lock (level 3) is removed from the specified record. Calling this routine for a record that is already unlocked is not an error. B-Tree Filer does not call the operating system to remove the lock again.

Example

See _4.F for an example.

See Also

BTLockRec

BTUnlockFileBlock

BTUnlockAllRecs

BTUsedKeys

Syntax

```
function BTUsedKeys(IFBPtr : IsamFileBlockPtr; Key : Word) : LongInt;
```

Purpose

Determine the number of keys added to an index.

Description

B-Tree Filer keeps count of each key added or deleted from each index. Hence, this call doesn't need to scan the entire index; instead it gets the count by reading a single information record from the index file.

BTUsedKeys is especially useful with variable length records (when exactly one key is added for each logical record for a certain index), since there is no other fast method to determine the number of logical records.

Example

This can be used to perform a simple consistency check on the data and index files if exactly one key is added for every data record.

```
var
  URecs : LongInt;
  UKeys : LongInt;
  I      : Word;
...
  URecs := BTUsedRecs(PF);
  if not IsamOK then begin
    {Error handling}
  end;
  for I := 1 To BTNrOfKeys(PF) do begin
    UKeys := BTUsedKeys(PF);
    if not IsamOK then begin
      {Error handling}
    end;
    if URecs <> UKeys then begin
      Writeln('Error! The number of data records is ', URecs);
      Writeln('The number of keys in index ', I, ' is only ',
        UKeys);
    end;
  end;
```

If the number of keys doesn't equal the number of data records, there must be an error. This kind of error can be corrected with the REINDEX unit.

BTUsedRecs

Syntax

```
function BTUsedRecs (IFBPtr : IsamFileBlockPtr) : LongInt;
```

Purpose

Return the number of valid data records in the data file.

Description

After a data record is deleted by BTDeleteRec, a "hole" remains in the data file. The "holes" can be filled by later calls to BTAddRec. BTUsedRecs returns the number of actual data records (not counting deleted records) in the data file.

Example

```
var
  URecs : LongInt;
  DRecs : LongInt;
...
  URecs := BTUsedRecs (PF);
  if not IsamOK then begin
    {Error handling}
  end;
  DRecs := BTFreeRecs (PF);
  if not IsamOK then begin
    {Error handling}
  end;
  Writeln('The number of used records is ', URecs);
  Writeln('The number of deleted records is ', DRecs);
```

Displays the number of used and deleted records in the fileblock. Note that both routines must access information in the system record of the data file and therefore may generate errors.

See Also

BTFileLen

BTFreeRecs

6. Tools

The tools described in this chapter make database applications even easier to write by building on the FILER unit. Each tool is implemented as a Turbo Pascal unit that can be used whenever the program needs it. These units are designed to work in either single user or network mode.

The following units are described in this chapter.

EMSHEAP

Memory can be allocated and freed like the conventional heap by using this expanded memory heap manager. The FILER unit uses EMSHEAP to manage index page buffers stored in EMS memory. You can use it to store your own data structures in expanded memory. This unit can only be used in real mode.

VREC

This unit provides special reading and writing routines for variable length data records. These routines are used in place of BTAddRec, BTPutRec, and BTGetRec. Among other uses, VREC enables the storage of "memo fields" (variable length notes) without wasting the space of the longest note for every record.

RESTRUCT/REINDEX

Restructure the database by copying it to a new database, skipping deleted records and reformatting records if necessary. Reindex a database by creating a new index file.

REBUILD/VREBUILD

These units are provided for backward compatibility (you should use RESTRUCT/REINDEX whenever possible). They regenerate the data and index files of a fileblock to remove deleted record space and create a clean index. Each key string is provided by a user-supplied routine. The units work with fixed and variable length records, respectively.

REORG/VREORG

These units are provided for backward compatibility (you should use RESTRUCT/REINDEX whenever possible). They are similar to REBUILD and VREBUILD, but provide the additional capability of modifying each data record as it is passed from input to output. Use them to import data from foreign fixed-length record formats or to add/delete fields from existing B-Tree Filer fileblocks. Data record modification occurs within a user-supplied routine.

FIXTOVAR

Convert a fixed length record fileblock to a variable length record fileblock with this unit.

NUMKEYS

This unit provides a variety of functions for manipulating key strings, including routines to convert numbers to and from strings, and routines to compress ASCII strings, yielding up to 50% reduction in index size.

ISAMTOOL

This is a collection of miscellaneous routines to extend the FILER unit, which for various reasons were felt to be unsuitable for inclusion in the unit itself. Included are routines for increasing the number of file handles available to a program, returning a text string for each IsamError code, and inverting a key string to provide for descending indexes.

MSORT/MSORTP

Any type and number of data items can be sorted with these units, which implement a merge sort algorithm that can utilize disk space or EMS for secondary storage. Control of input, comparison criteria, and output is provided by user-supplied routines.

DBIMPEXP

This unit converts dBase III or IV data and memo files to a B-Tree Filer fileblock, and back again.

EMSHEAP offers the same basic services as Turbo Pascal's built-in heap manager, with the exception that EMSHEAP uses EMS memory for its source of storage space. EMSHEAP can only be used in real mode, since EMS is not available to protected mode and Windows programs. The FILER unit calls EMSHEAP to manage the storage of index page buffers in EMS memory; for that application you won't need to know the details of EMSHEAP operation. You can also call EMSHEAP directly if your data storage requirements exceed the capacity of the real mode DOS 640K memory limit.

EMSHEAP offers a small but powerful set of routines for managing EMS:

- GetEMSMem and FreeEMSMem allocate and deallocate blocks of EMS, with a minimum granularity of 64 bytes and a maximum block size of 32K bytes.
- MapEMSPtr converts a logical pointer returned by GetEMSMem into a physical pointer usable in normal Pascal expressions.
- EMSMemAvail and EMSMaxAvail return available EMS memory like their Turbo Pascal counterparts.
- SaveEMSCtxt and RestoreEMSCtxt save and restore the state of the EMS page frame to avoid disturbing other users of EMS memory.
- InitEMSHeap and ExitEMSHeap initialize and dispose of EMS memory management structures at runtime.

This section describes how to configure EMSHEAP, guidelines for using it, and some of its limitations.

EMSHEAP depends on the services of an expanded memory driver compatible with Intel specification version 3.2 or later. The source of physical memory for the EMS is not important, except to the extent that it determines performance. For example, EMS that is implemented with mapped extended memory on an 80386 machine (using, for example, a 386MAX or QEMM386 driver) is usually the fastest. EMS implemented with a separate hardware card (such as an Intel AboveBoard) is as fast as the 386 or perhaps a little slower. An EMS driver that emulates expanded memory using 80286 extended memory is significantly slower, since memory blocks must be frequently copied to and from extended memory. And of course the slowest EMS driver is one that uses disk space to emulate expanded memory.

EMSHEAP isolates itself from direct calls to the expanded memory manager by using routines in the EMSSUPP unit, which is also supplied. EMSSUPP is patterned after the TPEMS and OPEMS units from Turbo Professional and Object Professional, respectively, so that you can replace EMSSUPP with these units if you're already using them in your program. Since EMSSUPP is not designed for direct use, it is not documented here.

You can skip the following few paragraphs if you're already familiar with EMS memory.

The EMS standard developed by Lotus, Intel, and Microsoft is defined by the following basic characteristics:

- a separately addressed RAM area of up to 32MB, addressable in 16KB logical pages
- a "page frame" that typically consists of 64KB of addressing space located in the top portion of normal memory. The 32MB of EMS memory are accessed by mapping one 16K block at a time into each of 4 "physical pages" (numbered 0..3) that comprise the page frame

- a device driver that isolates the application from the EMS hardware. This program, often called the EMM (expanded memory manager), will be referred to here as the EMS driver

The basic functions of the EMS driver are to manage allocation and deallocation of the logical pages, and to map the logical pages into the physical page frame for access by the application.

A group of logical pages is assigned a handle number for identification when it is allocated by the EMS driver. The EMS driver uses the handle number when it maps this group of logical pages into the page frame. At the same time, the EMS driver requires the logical page number within the group (starting from 0 for each group) and the physical page number (0..3) to map it into.

This is not the only use of an EMS handle. The EMS driver can remember a particular context for the page frame (which logical pages and handles are currently mapped into it) and restore it later. Each such context must be associated with a given EMS handle. As a result, the number of available context mappings is equal to the number of EMS handles allocated, which has direct consequences on the configuration of EMSHEAP, as you'll see below.

Configuration of EMSHEAP

Before adding EMSHEAP to the uses list of a program or another unit, you should check and perhaps modify EMSHEAP.PAS to specify the desired compilation defines. The following defines have an effect on EMSHEAP. Note that you activate a define by removing the period between the '{' and '\$', and deactivate it by inserting a period there.

UseTPEMS;

UseOPEMS;

If your application uses TPEMS (Turbo Professional) or OPEMS (Object Professional), you can define one of these conditionals to prevent the use of the EMSSUPP unit. This saves about 5.5KB of code.

DebugEMSHeap;

NoErrorCheckEMSHeap;

DebugEMSHeap should be used only if problems occur with the EMSHEAP unit. It adds checks within the EMSHEAP routines that increase the size of the unit. When an error occurs, these checks display an error code and, in some cases, abort the program. DebugEMSHeap activates error checking in the initialization block of EMSHEAP. If no EMS driver is found, the EMS version is too old, not enough EMS memory is available, or some other initialization error occurs, an error code is reported but the application continues (without access to EMS, of course). It modifies the error checking of GetEMSMem so that the application halts if there is not enough memory available. It also activates a check for mapping the Nil pointer in MapEMSPtr.

Definition of the symbol NoErrorCheckEMSHeap disables certain standard error checks, which makes the EMSHEAP unit as fast as possible. This should be done only after your application is bug free.

ManualInitEMSHeap;

Define ManualInitEMSHeap if you want to specify the configuration of EMS memory at runtime. ManualInitEMSHeap is not defined by default; in this case the EMSHEAP unit configures itself automatically. In some cases, especially when your program uses other units that consume EMS space, it may be desirable to decide how much EMS to allocate at runtime. See InitEMSHeap later in this section for more information.

StdEMSInstCheck;

When this define is not active (the default), EMSHEAP determines whether EMS memory is available by looking for a signature in memory near where the int \$67 vector points. When the define is active and the first test fails, EMSHEAP then tries to open the device 'EMMXXXX0'. Some older EMS drivers fail the first test even though EMS is actually present.

EMSHEAP.CFG

The file EMSHEAP.CFG also contains four constants that control the configuration of the EMSHEAP unit.

```
HandlesToUseForAlloc = 8;
```

The number of EMS handles allocated determines how many page frame mapping contexts can be stored simultaneously. The default value of HandlesToUseForAlloc can manage up to eight unrelated mappings. If EMS is used extensively, a larger number of available mappings will lead to better performance. The acceptable range for HandlesToUseForAlloc is from 1 to 252. The lower value of 1 must be raised to 2 if EMSDisturbance is defined in BTDEFINE.INC (because the FILER unit then reserves one handle to preserve its own page mapping context). The upper limit of 252 is set by the EMS driver's limit of 255 after accounting for handles that EMSHEAP uses internally.

In addition, the constant HandlesToUseForAlloc influences the minimum and maximum possible size of the EMS heap.

```
MinEMSHeapPages = HandlesToUseForAlloc;
MaxEMSHeapPages = 2048;
```

These constants determine the minimum and maximum number of logical pages that can be allocated for the EMS heap. Each page corresponds to 16KB of EMS memory. MinEMSHeapPages must be at least as large as HandlesToUseForAlloc, since each handle must allocate at least one page. MaxEMSHeapPages must be at least as large as MinEMSHeapPages. The maximum value for both constants is 2048, which corresponds to the 32MB maximum of EMS memory.

Normally these constants don't need to be modified. However, if a program absolutely requires a certain minimum amount of EMS memory, MinEMSHeapPages could be set to this value. If the computer has less than MinEMSHeapPages of free EMS available, the EMSHEAP initialization block sets the global boolean variable EMSHeapInitialized to False. If a program wants to reserve some EMS space for uses besides EMSHEAP, it can set MaxEMSHeapPages to limit the space EMSHEAP will use. However, it's better to use ToLetFreePages for this purpose.

```
ToLetFreePages = 1
```

This constant determines the number of EMS pages to leave free after EMSHEAP initialization is complete. If left at the default value of 1, EMSHEAP will allocate all available EMS memory except for 1 page worth, up to MaxEMSHeapPages. The maximum acceptable value for ToLetFreePages is

```
(Number of available logical pages) - 4 - MinEMSHeapPages
```

That is, EMSHEAP must be able to allocate at least MinEMSHeapPages pages plus four more pages that it uses to maintain an internal free list of disposed blocks.

Do not set ToLetFreePages to zero, the minimum value of 1 circumvents a bug with some versions of the MS-DOS EMM386.EXE driver.

Effects of Configuration

When ManualInitEMSHeap is undefined, the EMSHEAP initialization code allocates as many logical pages as it can, consistent with these constants. EMSHEAP then manages suballocation of requested blocks from these pages. For internal reasons, EMSHEAP cannot give more than 4 megabytes to one EMS handle, so the maximum number of bytes in the EMS heap is

```
HandlesToUseForAlloc * 4194304
```

As mentioned above, EMSHEAP allocates at least one logical page for each EMS handle, so the minimum number of bytes in the EMS heap is

$$\text{HandlesToUseForAlloc} * 16384$$

Hence, using the default constants, at least 192KB of EMS (8*16KB + 64KB for the free list) must be free for successful initialization of EMSHEAP. If initialization fails, EMSHeapInitialized is set to False.

In all cases EMSHEAP uses no memory from the normal heap. It does consume a small amount of space in the data segment, where memory use can be calculated as follows:

$$60 + 13 * \text{HandlesToUseForAlloc}$$

For the default settings this amounts to 164 bytes of data segment space.

The granularity of the EMS heap is set to 64 bytes, primarily to minimize fragmentation. All block sizes passed to GetEMSMem and FreeEMSMem are automatically adjusted to a multiple of 64.

The maximum size of any one memory block allocated by a program that uses EMSHEAP is 32KB. The most important reason for this limitation is that it allows copying directly from one EMS memory block to another without intermediate page mapping or use of an intermediate buffer.

Note that EMSHEAP installs a Turbo Pascal exit procedure: when the program halts (either normally or with a runtime error), EMSHEAP automatically frees all EMS memory that it allocated. There is one important caveat: if you are running your program under Turbo Debugger and stop the program by exiting the debugger, the program's exit procedures are not executed, and EMS memory remains allocated. In this case, you need to reboot your computer to regain access to this memory.

Coexisting with Turbo Pascal's OVERLAY Unit

The OVERLAY unit provided with Turbo Pascal can benefit greatly from the availability of EMS. The overlay manager allocates EMS at the time of the call to OvrInitEMS. In order to assure friendly coexistence between EMSHEAP and OVERLAY, there are three cases to consider:

1. If ManualInitEMSHeap is undefined, and OvrInitEMS is to be called at any time after EMSHEAP's initialization block is executed, you must prevent EMSHEAP from allocating all available EMS space. You can do so by setting ToLetFreePages to an appropriate value, typically equal to the size of the OVR file generated by the compiler, rounded up to the next even multiple of 16K, divided by 16K. The drawback to this method is that ToLetFreePages must be adjusted for each overlay file and each time the overlay file changes size. As a result, the following two approaches are recommended instead.
2. If ManualInitEMSHeap is defined, then InitEMSHeap should be called with a parameter of FreePages large enough to leave space for the overlay. Then OvrInitEMS can be called.
3. If OvrInitEMS is called prior to the initialization block of EMSHEAP, then OVERLAY can allocate the space that it needs in EMS, leaving the remainder free for the EMSHEAP unit. To do this, OvrInitEMS must be called from the initialization block of another unit, and that unit must be listed in the uses statement of the main program before EMSHEAP or any unit that uses EMSHEAP. The Turbo Pascal manual describes the initialization of the overlay manager in this fashion; see the Borland manual for more information.

User Hooks for Hard Errors and Allocation Failures

EMSHEAP's default action upon encountering a hard error--such as calling an EMSHEAP routine when EMSHeapInitialized is False or failure of a call made to the EMS driver--is to display the error number on the screen and to abort the program. The following typed constant allows modification of the default action:

```
EMSHardErrorFuncPtr : Pointer = Nil;
```

To provide a non-default action, assign EMSHardErrorFuncPtr the address of a function that you have written. This function must be compiled with the far model, must be global, and must match the following prototype declaration:

```
function MyHardErrorFunc(Error : Word) : Boolean;
```

When the error function is called, Error contains the code for the error that just occurred (error codes are described below). The function should return True to abort the program, or False to continue. Program continuation after an EMS hard error is not generally recommended; actions should usually be limited to a graceful shutdown of non-EMS program operations.

A hard error function can be installed with the following statement:

```
EMSHardErrorFuncPtr := @MyHardErrorFunc;
```

In order to return to the default action, just use:

```
EMSHardErrorFuncPtr := Nil;
```

Warning: the hard error function cannot call any routine of the EMSHEAP unit either directly or indirectly.

The default action of GetEMSMem when called with a request for more bytes of EMS memory than are available is to return a Nil pointer. The following typed constant allows modification of the default action:

```
EMSHeapErrorFuncPtr : Pointer = Nil;
```

EMSHeapErrorFuncPtr is analogous to Turbo Pascal's HeapError function pointer. Whenever a GetEMSMem request cannot be fulfilled, EMSHEAP calls the corresponding function. EMSHeapErrorFuncPtr contains the address of a function that must be compiled with the far model, must be global, and must match the following prototype declaration:

```
function MyHeapErrorFunc(Size : Word) : Integer;
```

When the EMS heap error function is called, Size contains the number of bytes (after rounding to a 64 byte boundary) that were requested but could not be allocated. The function returns one of three values:

- 0 Call the hard error function
- 1 Return a Nil pointer
- 2 Retry the operation

When EMSHeapErrorFuncPtr is Nil, EMSHEAP's behavior is equivalent to the error function returning a 1. Note the contrast to Turbo Pascal's normal heap error function, which by default returns 0 to halt the application.

Install a non-default EMS heap error function with a statement like the following:

```
EMSHeapErrorFuncPtr := @MyHeapErrorFunc;
```

Warning: The heap error function can call only the following routines from the EMSHEAP unit: GetEMSMem, FreeEMSMem, EMSMemAvail, and EMSMaxAvail.

Issues When Using EMSHEAP

For the most part, EMSHEAP works like the normal Turbo Pascal heap and the calls you make to it are handled the same way. However, there are four important differences caused by the nature of the compiler and the nature of EMS memory.

New and Dispose

The first difference is that there is no EMSHEAP counterpart for New and Dispose. These procedures are more like compiler macros, since they push a hidden parameter that specifies the size to allocate before calling an internal system allocation routine. When used for objects, New and Dispose are even trickier, since these operations actually occur within the constructor or destructor of the object. EMSHEAP routines can be used to replace New and Dispose for non-object applications simply by passing the size of the block to be allocated explicitly to the routine.

Accessing EMS Memory

The second difference occurs because all EMS memory is accessed through the tiny 64K page frame. Hence, only a small part of EMS is immediately available and thereby useful. Before the data can be used, it must be mapped into the page window. The mapping operation occurs when MapEMSPtr converts a "logical" pointer returned by GetEMSMem to a "physical" pointer usable by other Turbo Pascal routines. Since every call to MapEMSPtr can remap the page frame, there's no guarantee that the pointer returned by a prior call to MapEMSPtr remains valid. As a result, the application program must take care to assure that physical pointers to the EMS page frame remain valid when they are used.

The simplest and safest method to assure validity is to call MapEMSPtr immediately before using each EMS pointer. Be sure to realize the consequences of choosing this strategy: No pointer returned by MapEMSPtr can be passed as a parameter to a routine that also calls MapEMSPtr. No pointer returned by MapEMSPtr can be used as the argument to a With statement within which MapEMSPtr is called. No pointer returned by MapEMSPtr can be assigned to a variable that is used after another call to MapEMSPtr.

This simple method is not usually optimal because it generates unneeded calls to MapEMSPtr, which reduce performance. (Note, however, that MapEMSPtr does detect when the page frame is already mapped correctly and does not call the EMS driver unnecessarily.) An optimal method is available, but it requires a more detailed understanding of how EMSHEAP works. The FILER unit optimizes its calls to MapEMSPtr in this way.

The basis for the optimization is knowing the life span of a page mapped via MapEMSPtr. The life span is dependent upon the size of the mapped block and the mappings that follow it. Since the largest possible block allocated via GetEMSMem is 32KB, blocks may consist of at most two EMS pages. EMSHEAP uses the following rules to decide how to remap the page frame for a given series of calls to MapEMSPtr:

1. If memory blocks all smaller than or equal to 16KB are mapped, the last four mappings remain valid.
2. If memory blocks whose size is greater than 16KB (and less than or equal to 32KB, the maximum) are mapped one immediately after the other, the last two mappings remain valid.

3. If a block larger than 16KB is mapped just after a block smaller than 16KB, all previous mappings become invalid.
4. If a block smaller than 16KB is mapped just after a block larger than 16KB, the larger block's mapping remains valid. Another block smaller than 16KB can also be mapped, leaving three mappings valid.

The following example program may help to make these rules more tangible.

```

uses
  EMSHeap;
type
  MyArraySmall = array[1..16000] of Byte;
  SmallPtr     = ^MyArraySmall;
  MyArrayBig   = array[1..32000] of Byte;
  BigPtr       = ^MyArrayBig;
var
  I      : Integer;
  EMSPtr1 : array[1..5] of EMSPtr;
  EMSPtr2 : array[1..3] of EMSPtr;
  DataPtr1 : array[1..5] of SmallPtr;
  DataPtr2 : array[1..3] of BigPtr;
begin
  for I := 1 To 5 do begin
    GetEMSMem(EMSPtr1[I], SizeOf(MyArraySmall));
    if EMSPtr1[I] = Nil then begin
      Writeln('Not enough memory available.');

```

Five sections of 16000 bytes are allocated on the EMS heap. The logical pointers of type EMSPtr returned by GetEMSMem are stored in the pointer array EMSPtr1. MapEMSPtr is then used to map the data area into the EMS frame and return a physical pointer to this area. It must be typecast to the proper type (SmallPtr) since MapEMSPtr returns a generic pointer. The area pointed to by DataPtr1[I]^ is then filled with a value.

How should the EMS pointers be used later in the program? First, note that the last four mapped pointers, DataPtr[2] through DataPtr[5], are still valid after the For loop completes. Therefore a statement like

```
DataPtr1[3]^ := DataPtr1[2]^;
```

is valid without another call to MapEMSPtr. By contrast, the following statement will copy the wrong data into DataPtr1[3]^:

```
DataPtr1[3]^ := DataPtr1[1]^;
```

Now assume that the main block of the example is replaced with the following code which allocates blocks larger than 32KB:

```

for I := 1 To 3 do begin
  GetEMSMem(EMSPtr2[I], SizeOf(MyArrayBig));
  if EMSPtr2[I] = Nil then begin
    Writeln('Not enough memory available');

```

```

        Halt;
    end;
    DataPtr2[I] := BigPtr(MapEMSPtr(EMSPtr2[I]));
    FillChar(DataPtr2[I]^, SizeOf(MyArrayBig), I);
end;
{Further use of the initialized sections}

```

Now only DataPtr2[2] and DataPtr2[3] are valid for the further use. DataPtr2[1] points to an undefined area.

Finally, assume that both the EMSPtr1 and EMSPtr2 arrays have been allocated, leading to a mix of blocks with sizes less than and greater than 16KB, and consider the following sequence of calls to MapEMSPtr:

```

DataPtr1[1] := SmallPtr(MapEMSPtr(EMSPtr1[1]));
DataPtr1[2] := SmallPtr(MapEMSPtr(EMSPtr1[2]));
DataPtr1[3] := SmallPtr(MapEMSPtr(EMSPtr1[3]));
DataPtr1[4] := SmallPtr(MapEMSPtr(EMSPtr1[4]));
{Checkpoint 1}
DataPtr2[1] := BigPtr(MapEMSPtr(EMSPtr2[1]));
{Checkpoint 2}
DataPtr2[2] := BigPtr(MapEMSPtr(EMSPtr2[2]));
{Checkpoint 3}
DataPtr1[1] := SmallPtr(MapEMSPtr(EMSPtr1[1]));
DataPtr1[2] := SmallPtr(MapEMSPtr(EMSPtr1[2]));
{Checkpoint 4}

```

At checkpoint 1, all four values in the DataPtr1 array remain valid (this is an example of rule 1). At checkpoint 2, only DataPtr2[1] is valid (rule 3). At checkpoint 3, both DataPtr2[1] and DataPtr2[2] are valid (rule 2). At checkpoint 4, DataPtr2[2], DataPtr1[1], and DataPtr1[2] are valid (rule 4).

Fairly complicated rules such as these may lead you to choose the simpler method of just calling MapEMSPtr every time. Your choice will depend on the importance of EMS access speed for your application. Remember that EMSHEAP is unable to detect failure on your part to follow the mapping rules; the application will just use the wrong data leading to unpredictable consequences.

Saving/Restoring the EMS Context

The third difference between the normal heap and an EMS heap is also caused by the nature of page frame mapping. This issue arises only when two or more different EMS users are accessing the EMS page frame within the same system. In this case, it's important that the two users don't disturb one another, i.e., that one or both of the users leave the page frame undisturbed.

A common example occurs when a disk cache program that uses EMS is loaded while a program that also uses EMS is accessing the disk. The disk cache is responsible for saving the page frame context before changing it, and then restoring the page frame before returning control to the application. This is essential because the application would have no way of knowing when its page frame was disturbed. Another common example would occur when the Turbo Pascal OVERLAY unit is enabled for EMS usage and the EMSHEAP unit is also used. The OVERLAY unit is designed to save and restore the EMS context, so it won't disturb EMSHEAP.

Even though EMSHEAP need not save and restore context for these two examples, it's still a good idea for it to do so in general, since other cases may arise where it's essential. (Such a case would occur if EMSHEAP and the EMS array routines from Turbo Professional or Object Professional are both used.) To serve this need, EMSHEAP offers the SaveEMSCtxt and RestoreEMSCtxt

procedures. If the example program shown above were to be localized in a function, it would use the context saving routines as follows:

```
function DoSomethingWithData : Boolean;
type
  MyArraySmall = array[1..16000] of Byte;
  SmallPtr     = ^MyArraySmall;
  MyArrayBig   = array[1..32000] of Byte;
  BigPtr       = ^MyArrayBig;
var
  I           : Integer;
  EMSPtr1     : array[1..5] of EMSPtr;
  EMSPtr2     : array[1..3] of EMSPtr;
  DataPtr1    : array[1..5] of SmallPtr;
  DataPtr2    : array[1..3] of BigPtr;
  SaveHandle  : Byte;
begin
  DoSomethingWithData := False;
  SaveHandle := SaveEMSCtxt;
  for I := 1 To 5 do begin
    GetEMSMem(EMSPtr1[I], SizeOf(MyArraySmall));
    if EMSPtr1[I] = Nil then begin
      RestoreEMSCtxt(SaveHandle);
      Exit;
    end;
    DataPtr1[I] := SmallPtr(MapEMSPtr(EMSPtr1[I]));
    FillChar(DataPtr1[I]^, SizeOf(MyArraySmall), I);
  end;
  {Further use of the initialized sections}
  RestoreEMSCtxt(SaveHandle);
  DoSomethingWithData := True;
end.
```

The function returns False if the required EMS memory is not available, but even in this case it restores the page frame before returning. (The example does not deallocate the EMS memory already allocated, which it should do in a real application.) Before the routine exits, it calls RestoreEMSCtxt to restore the page frame.

Note that calls to GetEMSMem or FreeEMSMem do not disturb the page frame. If this example routine never called MapEMSPtr, there would be no need for it to call SaveEMSCtxt or RestoreEMSCtxt.

The context management routines are described more thoroughly later in this section.

Releasing EMS Storage

The fourth difference between the normal heap and an EMS heap is that the size provided when releasing EMS storage must be exactly the same as the size given when it was allocated. It isn't possible to allocate a large block and then to free part of it later.

Error Codes

Some of the following error codes are generated only if particular EMSHEAP conditional defines are activated. Such codes are indicated by {\$IFDEF Symbol} or {\$IFNDEF Symbol} in the table.

If DebugEMSHeap is defined, initialization errors are reported when they occur with a routine that calls Writeln, then waits for <Enter> to be pressed. Execution of the program continues, but

EMSHeapInitialized is set to False. If DebugEMSHeap is not defined, no error code is reported, but EMSHeapInitialized is set to False.

When an error occurs after initialization, EMSHEAP calls the hard error routine, unless the error was the result of insufficient free EMS space, in which case it calls the allocation error routine. See "User Hooks for Hard Errors and Allocation Failures" earlier in this section for more information.

Error codes during initialization (all `{ $IFDEF DebugEMSHeap }`)

```
1  EMS driver not found
2  EMS version is not 3.2 or greater
3  Pointer to the EMS page frame could not be determined
4  Number of usable EMS pages could not be determined
5  HandlesToUseForAlloc is less than MinEMSHeapPages
6  Number of free EMS pages is less than MinEMSHeapPages
7  Too many pages were requested for one EMS handle. The maximum
   is 256 pages
   (4MB per EMS handle). HandlesToUseForAlloc needs to be
   increased.
8  EMS memory could not be allocated
9  EMS page frame context could not be saved
10 Mapping a logical EMS page was not possible
11 Not enough EMS handles to fulfill memory request
12 EMS page frame context could not be saved
13 EMS page frame context could not be restored
20 Invalid page frame pointer (offset is not 0)
21 Attempt to use EMS in protected mode
```

Other errors

```
50 { $IFDEF NoErrorCheckEMSHeap }
   An EMSHEAP routine was called while EMSHeapInitialized was
   False
60 { $IFDEF NoErrorCheckEMSHeap }
   A disallowed EMSHEAP routine was called from the EMSHeapError
   function
70 { $IFDEF ManualInitEMSHeap }
   Manual initialization is possible only if ManualInitEMSHeap is
   defined
75 { $IFDEF ManualInitEMSHeap }
   Manual initialization was already performed
80 { $IFDEF ManualInitEMSHeap }
   Manual deinitialization is possible only if ManualInitEMSHeap
   is defined
100 EMS page frame context could not be saved before mapping in the
    free list
101 Mapping the free list was not possible
102 EMS page frame context could not be saved before unmapping the
    free list
103 EMS page frame context could not be restored after unmapping
    the free list
110 EMS page frame context could not be saved in SaveEmsCtxt
111 SaveEmsCtxt failed because no free EMS handles were available.
    Increase HandlesToUseForAlloc.
112 EMS page frame context could not be restored in RestoreEmsCtxt
```

```

120 {$IFDEF NoErrorCheckEMSHeap}
    Attempt to allocate a block larger than 32KB
121 Insufficient free EMS is available
130 {$IFDEF NoErrorCheckEMSHeap}
    The pointer passed to FreeEMSMem is Nil
131 {$IFDEF NoErrorCheckEMSHeap}
    The size passed to FreeEMSMem is greater than 16KB, and the
corresponding
    pointer doesn't match
132 {$IFDEF NoErrorCheckEMSHeap}
    The size passed to FreeEMSMem is less than 16KB, and the
corresponding
    pointer doesn't match
140..148 Mapping a logical EMS page was not possible
150 {$IFDEF DebugEMSHeap}
    The virtual pointer passed to MapEMSPtr is Nil

```

Declarations

Constants

```

DoManualInitEMSHeap; =
{$IFDEF ManualInitEMSHeap}
    True;
{$ELSE}
    False;
{$ENDIF}

```

This constant reflects the state of the ManualInitEMSHeap define. If DoManualInitEMSHeap is False, you should not call InitEMSHeap or ExitEMSHeap.

```
EMSHardErrorFuncPtr; : Pointer = Nil;
```

Address of the function for hard errors. The default value of Nil disables any user defined function.

```
EMSHeapErrorFuncPtr; : Pointer = Nil;
```

Address of the heap error function which is called when memory requests cannot be fulfilled. The default value of Nil disables any user defined function.

```
HandlesToUseForAlloc; = 8;
```

Number of EMS handles used to allocate the EMS heap. Valid values are between 1 and 252. See "Configuration of EMSHEAP" earlier in this section for more information.

```
MaxEMSHeapPages; = 2048;
```

Maximum number of logical pages that EMSHEAP can use. Valid values are between MinEMSHeapPages and 2048. See "Configuration of EMSHEAP" earlier in this section for more information.

```
MinEMSHeapPages; = HandlesToUseForAlloc;
```

Minimum number of logical pages that must be available to initialize EMSHEAP successfully. Valid values are between HandlesToUseForAlloc and 2048. See "Configuration of EMSHEAP" earlier in this section for more information.

```
ToLetFreePages; = 1;
```

Number of logical pages that EMSHEAP will leave free at initialization, for the use of another application or another part of the program. ToLetFreePages must be small enough so that at least MinEMSHeapPages+4 pages can be allocated by EMSHEAP. See "Configuration of EMSHEAP" earlier in this section for more information.

Types

```
EMSPtr = Pointer;
```

All logical pointers returned by `GetEMSMem` or passed to `FreeEMSMem` are of this type. This pointer cannot be used directly, but must always be passed to `MapEMSPtr` which returns a physical pointer.

Variables

```
EMSHeapInitialized; : Boolean;
```

This variable is set by the initialization code of the EMSHEAP unit and should not be modified by the user program. If `EMSHeapInitialized` is `False`, no routines from EMSHEAP can be called. To determine the reason for a `False` result, enable the `DebugEMSHeap` define.

EMSMaxAvail / EMSMemAvail

Syntax

```
function EMSMaxAvail; : Word;
```

Purpose

Return the size of the largest available block.

Description

The return value is between 0 and 32KB, since EMSHEAP does not allow blocks larger than 32KB. A following call to GetEMSMem will be successful if called with a size less than or equal to the value returned by EMSMaxAvail.

Example

```
var
  MyArray      : array[1..20000] of Byte;
  MyArEMSPtr   : EMSPtr;
...
if EMSMaxAvail >= SizeOf(MyArray) then
  GetEMSMem(MyArEMSPtr, SizeOf(MyArray))
else
  Writeln('Not enough EMS memory.');
```

The call to EMSMaxAvail determines that there is still enough free memory to allocate MyArray in the EMS heap. By checking EMSMaxAvail first, there is no concern that GetEMSMem will activate the EMS heap error function and no need to check whether GetEMSMem returns a Nil pointer. On the other hand, by calling EMSMaxAvail first, the EMS free list is scanned twice, so this code may be slower than calling GetEMSMem and testing whether it failed.

See Also

EMSMemAvail

GetEMSMem

Syntax

```
function EMSMemAvail; : LongInt;
```

Purpose

Return the total amount of free EMS space.

Description

If you call EMSMemAvail before calling GetEMSMem the first time, you get the total size of the EMS heap. The total size is between MinEMSHeapPages*16KB and MaxEMSHeapPages*16KB.

GetEMSMem might fail even if EMSMemAvail returns a value larger than the needed block size. That's because the total returned by EMSMemAvail can be composed of many small non-contiguous pieces of EMS.

Example

```
Writeln('Free memory in the EMS heap: ', EMSMemAvail, ' Bytes');
```

Displays available space on the EMS heap.

See Also

EMSMaxAvail

ExitEMSHeap / FreeEMSMem

Syntax

```
procedure ExitEMSHeap;
```

Purpose

Deinstall the EMSHEAP unit.

Description

This routine frees all EMS pages allotted to EMSHEAP. Nothing happens if EMSHEAP has already been deinstalled. The boolean variable EMSHeapInitialized is set to False after successful completion.

Calling this routine when DoManualInitEMSHeap has a value of False generates a call to the hard error function with an error code of 80.

If EMSHEAP is being used in combination with the FILER unit, do not call ExitEMSHeap until after you have called BTExitIsam.

ExitEMSHeap is called automatically by the EMSHEAP unit's exit procedure.

Example

```
if DoManualInitEMSHeap then  
  ExitEMSHeap;
```

Deinstalls the EMSHEAP unit at the end of a program.

See Also

InitEMSHeap

Syntax

```
procedure FreeEMSMem; (EPtr : EMSPtr; Size : Word);
```

Purpose

Free a previously allocated section of the EMS heap.

Description

EPtr is a logical pointer that was obtained through a previous call to GetEMSMem. Size must be exactly the same as originally allocated.

FreeEMSMem updates the internal list of free blocks. If the new free block adjoins free blocks on one or both sides of itself, those free blocks are merged into one. Hence, if all allocated EMS space is freed by calling FreeEMSMem, the EMS heap will return to the same state it had after initialization.

Example

```
var  
  MyArray      : array[1..20000] of Byte;  
  MyArEMSPtr   : EMSPtr;  
  
  ...  
  GetEMSMem(MyArEMSPtr, SizeOf(MyArray));  
  {Error checking}  
  ...  
  FreeEMSMem(MyArEMSPtr, SizeOf(MyArray));
```

The variable MyArEMSPtr is invalid and cannot be passed to MapEMSPtr after the call to FreeEMSMem.

See Also

GetEMSMem

GetEMSMem

Syntax

```
procedure GetEMSMem; (var EPtr : EMSPtr; Size : Word);
```

Purpose

Allocate a memory block on the EMS heap.

Description

Size is the size of the block to allocate, which must be in the range of 1 to 32768, inclusive. If Size is zero, GetEMSMem sets EPtr to Nil and exits without generating an error. If Size is larger than 32768, GetEMSMem generates hard error 120 and exits. All Size values are then adjusted to a multiple of 64 bytes.

EPtr returns a "logical" pointer to the allocated block. This logical pointer can only be passed to MapEMSPtr or FreeEMSPtr; it should never be used in a Pascal expression directly.

GetEMSMem scans the free list of EMS blocks to find a block large enough to hold the requested block. It takes the first sufficiently large block that it finds. (Requested blocks larger than 16KB may actually reside within two non-adjacent logical EMS pages.) If GetEMSMem cannot find sufficient space, its behavior depends on a heap error function. By default, GetEMSMem returns EPtr set to Nil in this case. See "User Hooks for Hard Errors and Allocation Failures" earlier in this section for more information.

If you call EMSMaxAvail to determine whether enough memory is available first, you can guarantee (short of a sudden hardware breakdown) that GetEMSMem will succeed.

Example

```
GetEMSMem(MyArEMSPtr, SizeOf(MyArray));  
if MyArEMSPtr = Nil then  
  Writeln('Not enough EMS memory.');
```

Contrast this example to the one shown for EMSMaxAvail. This approach is generally faster but assumes that the default heap error function, or another one with similar behavior, is used.

See Also

EMSMaxAvail

FreeEMSMem

InitEMSHeap / MapEMSPtr

Syntax

```
procedure InitEMSHeap, (FreePages : Word);
```

Purpose

Initialize the EMSHEAP unit.

Description

InitEMSHeap initializes the EMSHEAP unit and leaves at least FreePages pages of EMS memory for other EMS users. FreePages must meet the same constraints described for the constant ToLetFreePages previously. InitEMSHeap offers the advantage that FreePages can be computed at runtime rather than at compile time.

After a successful call to InitEMSHeap, the boolean variable EMSHeapInitialized is set to True.

Calling this routine when DoManualInitEMSHeap has a value of False generates a call to the hard error function with an error code of 70. Calling this routine twice without an intervening call to ExitEMSHeap generates a hard error of 75.

If you are using the EMSHEAP unit to supply EMS memory to the FILER unit, be sure to call InitEMSHeap before calling BInitlsam.

Examples

```
if DoManualInitEMSHeap then
  InitEMSHeap(0);
```

Initializes the EMSHEAP unit to use all available EMS memory; no EMS pages will be available to other processes or units.

```
if DoManualInitEMSHeap then begin
  assign(F, 'MYPROG.OVR');
  reset(F, 1);
  InitEMSHeap((FileSize(F)+16383) div 16384);
  close(F);
end;
```

Initializes the EMSHEAP unit to use all EMS space except for enough to load the program's overlay file into EMS.

See Also

ExitEMSHeap

Syntax

```
function MapEMSPtr; (EPtr : EMSPointer) : Pointer;
```

Purpose

Map the memory block and return a physical pointer to the data.

Description

This function is central to the functionality of the EMSHEAP unit. Pointer values of type EMSPointer returned by GetEMSMem cannot be used directly, but must first be passed as the EPtr parameter to MapEMSPtr. MapEMSPtr returns the physical address of the data after it is mapped into the EMS page frame.

Because MapEMSPtr returns an untyped pointer, the value must generally be typecast to a pointer of the desired type.

The pointer returned by MapEMSPtr has a limited lifetime; that is, further calls to MapEMSPtr invalidate the pointers returned by prior calls because the EMS page frame is remapped. See "Issues When Using EMSHEAP" earlier in this section for complete information.

Example

See "Issues When Using EMSHEAP" earlier in this section.

See Also
GetEMSMem

InitEMSHeap / MapEMSPtr

RestoreEMSCtxt / SaveEMSCtxt

Syntax

```
procedure RestoreEMSCtxt; (HandleInd : Byte);
```

Purpose

Restore a previously saved EMS context.

Description

HandleInd is a value previously returned by SaveEMSCtxt. RestoreEMSCtxt restores the page frame mapping that was active when SaveEMSCtxt was called.

Example

See "Issues When Using EMSHEAP" earlier in this section.

See Also

SaveEMSCtxt

Syntax

```
function SaveEMSCtxt; : Byte;
```

Purpose

Save the current state of the EMS page frame and return a handle index.

Description

Call SaveEMSCtxt when you need to save the page frame mapping at a particular time in order to avoid disturbing other users of EMS. SaveEMSCtxt returns a handle index (not a true EMS handle) that identifies the particular context to restore when RestoreEMSCtxt is called later. The returned value is between 1 and HandlesToUseForAlloc.

If all handles are already in use when SaveEMSCtxt is called, it generates a hard error of code 111. If the hard error function does not return True to halt the program, SaveEMSCtxt returns a handle of 255. To avoid this error, increase the constant HandlesToUseForAlloc, which defaults to 8.

Example

See "Issues When Using EMSHEAP" earlier in this section.

See Also

RestoreEMSCtxt

The VREC unit uses a clever method to extend B-Tree Filer for variable length records. Such records become extremely important when a database requires memo fields, since allocating record space for the longest possible memo can waste lots of disk storage fast.

To provide support for variable length records, you first Use the VREC unit in your program, following the FILER unit on which it depends. Then, instead of calling FILER's record access routines (e.g., BTAddRec, BTGetRec, BTPutRec), you call the analogous routines from VREC. FILER's routines for managing fileblocks and keys are used without change. VREC's record access routines are called almost like those in FILER, with the primary exception of an added Len parameter, which specifies or returns the length of the variable record.

There is one other important difference. Whereas it was previously recommended that each data record reserve the first four bytes for B-Tree Filer's use, this becomes a requirement for variable length records. Otherwise it is nearly impossible to reconstruct a damaged data file.

Here is VREC's clever idea: each variable length record is composed of a series of shorter fixed length "sections." The VREC unit subdivides and recomposes a varying record from the sections. FILER manages the fixed length sections as usual.

To make this work, FILER must be fooled a little bit. Specifically, the record length (DatSLen) that is passed to BTCreateFileBlock is the length of the fixed section rather than the varying length of the record itself.

How should the section length be chosen? The following rules should be applied:

- the section length should be kept as small as is consistent with the remaining rules in order to conserve disk space
- the section length should typically be a multiple of 2 (64, 128, 256, 512, 1024) because DOS disk access is faster for those block sizes. An especially good block size is 512 bytes since this is the standard disk sector size
- the largest record shouldn't be much larger than about 8 sections. While this isn't a firm limit, performance will suffer if each record must be pieced together from too many sections
- a section shouldn't be much larger than the smallest version of the record
- the section length should account for the fact that VREC uses 6 bytes of the first section and 7 bytes of each subsequent section for management of the record

A contrived example helps to make these rules clear. Suppose an application has a fixed minimum record size of 114 bytes (deletion tag, name, company, address, phone, customer number, etc.). Notes can be added to the end of the record, ranging from 1 to 250 bytes in length (thus fitting neatly into a Pascal string). Hence, the largest record size is 364 bytes after accounting for a string length byte.

Here are two reasonable choices for section lengths:

```
64 byte sections
  minimum number of sections: 2 (completely full: 114+1+6+7=128)
  maximum number of sections: 7 (36 bytes still available)
128 byte sections
  minimum number of sections: 1 (7 bytes still available)
  maximum number of sections: 3 (completely full)
```

For this example, the 128 byte section offers probably the best tradeoff between space and speed. NETDEMO.PAS provides a working example of variable records.

When variable length records are used, B-Tree Filer's routines BTUsedRecs, BTFreeRecs, and BTDatRecordSize do not return values associated with logical record counts and sizes. As you might expect, they instead return values based on the fixed section length. Nevertheless, the values returned are still useful for computing actual disk space requirements. The VREC unit provides a replacement for BTDatRecordSize: BTGetVariableRecLength returns the actual length of a specified record.

Program organization changes slightly when using VREC. The following list indicates new requirements of the variable record length facility with an asterisk.

1. Initialize the FILER unit and create page buffers by calling BTInitIsam.
2. Optionally create one or more fileblocks by calling BTCreateFileBlock.
3. Open one or more fileblocks by calling BTOpenFileBlock.
- *. Allocate a section buffer by calling BTCreateVariableRecBuffer or BTSetVariableRecBuffer.
4. Use the B-Tree Filer procedures and functions on the opened fileblocks.
5. Close opened fileblocks with BTCloseFileBlock.
- *. Dispose of the section buffer by calling BTReleaseVariableRecBuffer.
6. Dispose of the page buffer and shut down the FILER unit by calling BTExitIsam.

The section buffer is used to temporarily store each section while building a complete record. Only one buffer can be created at a time, no matter how many variable length fileblocks are opened. The size of the buffer must be as large as the largest section in any open fileblock that uses VREC services. Note that you may allocate the buffer by calling BTSetVariableRecBuffer prior to step 3, but BTCreateVariableRecBuffer can be called only after fileblocks are open.

The following list contrasts the use of fixed and variable length records in terms of the procedures called by a program.

Opening a Fileblock

When calling BTCreateFileBlock to create a new fileblock of variable length records, remember to specify the *section length* rather than the overall record length. The section length is stored in the data file header thereafter, so that BTOpenFileBlock can be called as usual. Also remember to call BTCreateVariableRecBuffer or BTSetVariableRecBuffer, before calling any of the other routines in the VREC unit.

Data File Operations

When adding, deleting, or updating records, use the routines exported by the VREC unit rather than those from FILER. The following table indicates the correspondence:

Fixed Length	Variable Length
-----	-----
BTAddRec	BTAddVariableRec
BTDeleteRec	BTDeleteVariableRec
BTGetRec	BTGetVariableRec
BTPutRec	BTPutVariableRec
BTGetRecReadOnly	BTGetVRecReadOnly

Note that the variable length routines work in single-user or network mode. Unlike fixed length records, it is not possible to read and write variable length records in spite of locks

placed by other workstations. Performing reads and writes in spite of locks is inherently unreliable; in the case of fixed length records, the data may be out of date by the time the information is used. In the case of variable length records, the VREC unit might be unable to reconstruct the chain of sections; therefore the operation is not allowed at all.

Also, BTGetVariableRec, BTAddVariableRec, and BTPutVariableRec require an additional parameter to specify or return the length of the record in bytes.

Remember that BTUsedRecs returns the number of *sections* in use within a variable record length fileblock. There is no immediate way to obtain the actual number of records in use. See the BTUsedKeys procedure in Chapter 5 for the next best way.

Index File Operations

The use of keys is exactly the same for variable length records as for normal fixed length records. All of the same routines from the FILER unit are used identically.

Browsing a Fileblock

The browsing units (described in Chapter 7) allow browsing records of either fixed or varying length. Remember to set the VarRec parameter to True for variable length records and False otherwise.

Rebuilding or Reorganizing a Fileblock

The RESTRUCT unit allows you to restructure a fileblock with fixed or variable length records. The REINDEX unit allows you to reindex a fileblock with fixed or variable length records.

Variable Length Record Example

Assume that you want to add a variable length memo field to the example given in _4.C. The memo will be stored in a field of the following type:

```
type
  MemoFieldArray = array[1..20] of String[70];
```

After adding the new field, the record type for this application becomes:

```
type
  PersonDefVar =
    record
      Del : LongInt;
      FirstName : String[20];
      LastName : String[25];
      Street : String[30];
      City : String[30];
      State : String[2];
      ZipCode : String[9];
      Telephone : String[15];
      Age : Integer;
      Memo : MemoFieldArray;
    end;
```

The Memo field is initialized to zero length strings when it is empty. Strings are stored starting in element 1 of the MemoFieldArray when the field is used. The size of the fixed portion of the PersonDefVar is computed by

```
SizeOf(PersonDefVar)-SizeOf(MemoFieldArray)
```

This specifies how many bytes are stored if the Memo field is empty. In this case the fixed portion uses 144 bytes. The largest Memo is 20*71 bytes, or 1420 bytes. These numbers lead to a section length of 256 bytes. In the CreateFile example routine, replace

```
BTCreateFileBlock('TEST', SizeOf(PersonDef), 3, IID);
```

with

```
BTCreateFileBlock('TEST', 256, 3, IID);
```

A call to allocate the section buffer must also be added at the end of the OpenFile routine:

```
if not BTCreateVariableRecBuffer(PF) then begin
  OpenFile := False;
  {Error handling}
end;
```

Next you need a small function that returns the number of bytes to store for a given variable of type PersonDefVar, depending on how much of the Memo field is used:

```
function LenOfData(var P : PersonDefVar) : Word;
var
  LastUsed : Word;
begin
  LastUsed := 20;
  while (LastUsed > 0) and (P.Memo[LastUsed] = '') do
    dec(LastUsed);
  LenOfData := SizeOf(PersonDefVar) -
    SizeOf(MemoFieldArray) + 71*LastUsed;
end;
```

This routine scans from the last element in the array of strings to find a non-blank string. Then it computes the length of the overall record based on the number of strings that are used.

Given the LenOfData function, the example AddRecord routine is modified by replacing calls to

```
BTAddRec(PF, RefNr, P)
```

with

```
BTAddVariableRec(PF, RefNr, P, LenOfData(P));
```

Similarly, in the example for modifying a record, replace

```
BTPutRec(PF, RefNr, P):
```

with

```
BTPutVariableRec(PF, RefNr, P, LenOfData(P));
```

Also substitute BTDeleteVariableRec for BTDeleteRec. No parameter changes are required for this routine.

BTGetVariableRec returns a length word, so you must declare a new local variable of that type when you substitute the variable length routine for BTGetRec. To determine how much of the Memo field was actually returned by a call to BTGetVariableRec, use the following approach.

```
function NumberOfMemoLines(Len : Word) : Word;
begin
  NumberOfMemoLines := (Len -
    SizeOf(PersonDefVar) + SizeOf(MemoFieldArray))
    div 71;
end;
```

```

...
var
  Len : Word
  I : Word;
...
  BTGetVariableRec(PF, RefNr, P, Len);
  for I := 1 to NumberOfMemoLines(Len) do
    Writeln(P.Memo[I]);

```

If you have an existing fixed length data file for this example, you need to convert it to variable length format before you can use it again. See [_6.F](#) for more information about performing the conversion.

The NETDEMO example program shows a different technique for managing a memo field. NETDEMO's approach offers somewhat reduced disk space usage and is also compatible with the memo editors of Turbo Professional and Object Professional. See NETDEMO.PAS for details.

Variable Length Record File Format

VREC splits each variable length record into sections to store on disk. It then rejoins the sections before returning the record to you in one piece. To do this, it adds a small amount of information to each section.

Let's assume a section length of 100 bytes to illustrate how VREC works. The first section of such a variable length record would include the following information:

Bytes 00..93	data from the user record
Bytes 94..95	bytes of user data in this section
Bytes 96..99	reference number of next section

Succeeding sections of the record, if any, would include the following information:

Bytes 00..00	constant value equaling 1
Bytes 01..93	data from the user record
Bytes 94..95	bytes of user data in this section
Bytes 96..99	reference number of next section

The constant byte of 1 in follow-on sections is used by the VREBUILD unit to help it find the start of each variable record (which must always begin with four zero bytes). In the last section of a variable length record, the last four bytes are zero, which terminates the chain of sections. Note that 6 bytes of the first section, and 7 bytes of succeeding sections, are used for VREC overhead.

When VREC adds a record, it stores the sections in what might seem to be reverse order. For example, suppose that you're starting with a freshly created fileblock and you're adding a record that will consume three sections. VREC stores the last third of the user record in section 1 of the fileblock, the middle third in section 2, and the starting third of the fileblock in section 3. The reference number returned by BTAddVariableRec is 3. VREC must use this order internally to obtain the reference numbers needed to chain the sections together.

Fileblock Maintenance with Variable Length Records

If a fileblock was not properly closed by calling BTCloseFileBlock, the next attempt to open it with BTOpenFileBlock returns the error code 10010. This occurs when buffered data was not written to disk.

The procedure RestructFileBlock from the RESTRUCT unit is used to rebuild a variable length fileblock. It constructs a new data file from the original data file. This is possible only if each record reserved four initial bytes for a deletion marker. RestructFileBlock has a tougher job than the

similar routine for fixed length records because it must reconnect the sections of each variable record. If the data file becomes corrupted and no backup file is available, data records following the first corrupted one may be lost. It is therefore especially important when using variable length record fileblocks to implement a good backup strategy.

To recreate the index file, use `ReIndexFileBlock` from the REINDEX unit.

Declarations

Constants

```
MaxVariableRecLength; = $FFF0;
```

The maximum number of bytes in one variable length record, regardless of how many sections it is divided into.

Variables

```
IsamVRecBufSize; : Word;
```

The current size of the section buffer. It is set to zero by VREC's initialization block and updated when you call `BTCreateVariableRecBuffer` or `BTSetVariableRecBuffer`. `IsamVRecBufSize` must be at least as large as the largest section of any open fileblock that VREC is managing. All functions in the VREC unit automatically adjust the section buffer if the current size is smaller than the section length of the specified fileblock. However, it's best for you to allocate a sufficiently large buffer at startup to avoid heap fragmentation and performance problems.

BTAddVariableRec / BTAdjustVariableRecBuffer

Syntax

```
procedure BTAddVariableRec;(IFBPtr : IsamFileBlockPtr; var RefNr : LongInt;  
                           var Source; Len : Word);
```

Purpose

Add a variable length record to the data file.

Description

The data record is added to the end of the data file if the file has no deleted sections; otherwise one or more "holes" created by BTDeleteVariableRec or BTPutVariableRec are filled. All future references to this data record are carried out by referring to the LongInt number RefNr returned by the routine.

Len specifies the actual number of bytes in the data structure Source. The length must be independently determined by a method similar to that described in the introduction to this section. The same Len is returned when the record is later retrieved using BTGetVariableRec.

A network fileblock must be locked for exclusive write access (level 1 or 2) before BTAddVariableRec can be called successfully.

Example

See the example at the beginning of this section.

See Also

BTDeleteVariableRec BTPutVariableRec

Syntax

```
function BTAdjustVariableRecBuffer;(Size : Word) : Boolean;
```

Purpose

Increase the section buffer size.

Description

This procedure increases the size of the section buffer, providing that Size is greater than the current value of IsamVRecBufSize. If an increase is warranted, the current section buffer is freed and a new section buffer is allocated from the heap. The routine returns True if either the buffer did not need expanding or the buffer was expanded successfully. It returns False if there was an out of heap memory error.

All functions in the VREC unit automatically adjust the section buffer by calling this routine if the current size is smaller than the section length of the specified fileblock.

See Also

BTCreatVariableRecBuffer BTReleaseVariableRecBuffer
BTSetVariableRecBuffer

BTCreateVariableRecBuffer

Syntax

```
function BTCreateVariableRecBuffer(IFBPtr : IsamFileBlockPtr) :  
    Boolean;
```

Purpose

Allocate a section buffer based on the specified fileblock.

Description

This procedure must be called after the fileblock passed as a parameter is open, and before calling any procedures that use variable length data records. An internal buffer is allocated from the heap to handle the variable length data records. The buffer's size matches the value of DatSLen originally used to create the fileblock (that is, the section length). If more than one variable length fileblock is to be opened simultaneously, the IFBPtr passed to CreateVariableRecBuffer must point to the fileblock with the largest value of DatSLen.

The buffer can be expanded when required (see BTAdjustVariableRecBuffer). However, it is usually best to avoid the heap fragmentation this would produce by allocating the buffer with the correct size to begin with.

If you have multiple variable length fileblocks, it may be easier to call BTSetVariableRecBuffer instead. It allows you to specify the desired buffer size directly instead of passing a fileblock variable.

Example

```
BTOpenFileBlock(PF, 'ADDRESS', False, False, False, False);  
if not IsamOK then begin  
    {Error handling}  
end else begin  
    if not BTCreateVariableRecBuffer(PF) then begin  
        {Error handling}  
    end;  
end;
```

Opens an existing variable record fileblock and allocates a section buffer just large enough for it.

See Also

BTAdjustVariableRecBuffer

BTDeleteVariableRec / BTGetVariableRec

Syntax

```
procedure BTDeleteVariableRec; (IFBPtr : IsamFileBlockPtr; RefNr :  
LongInt);
```

Purpose

Delete a variable length record from a data file.

Description

The deleted sections are marked as free, so that future additions with BTAddVariableRec can reuse the space. RefNr specifies the data reference number as returned by BTAddVariableRec, BTFindKey, or other key search routines.

Only existing data records can be deleted. Deleting an already deleted record corrupts an internal linked list of deleted data records and therefore has unpredictable results. When a record is deleted, B-Tree Filer sets the first four bytes of each section to a non-zero value.

In a network application, this procedure can be called successfully only when the network fileblock is locked at level 1 or 2.

Example

```
BTFindKey(PF, 1, RefNr, UserKeySt);  
if IsamOK then  
  BTGetVariableRec(PF, RefNr, Len);  
if IsamOK then begin  
  {Prompt user whether to delete}  
  ...  
  BTDeleteVariableRec(PF, RefNr);  
  if not IsamOK then begin  
    {Error handling}  
  end;  
end;
```

Uses the index system to find a particular record, then offers the user an opportunity to confirm the deletion. Note that the call to BTGetVariableRec is needed only to provide information for prompting the user; BTDeleteVariableRec does not use it.

See Also

BTAddVariableRec

Syntax

```
procedure BTGetVariableRec; (IFBPtr : IsamFileBlockPtr; RefNr :  
LongInt; var Dest;  
var Len : Word);
```

Purpose

Read a variable length record.

Description

RefNr is normally obtained from a prior index operation. The actual length of the data record is returned in the parameter Len. The record is pieced back together from the chain of sections.

Since Dest is untyped, the compiler will not detect an invalid variable passed in this position. Assure that the variable passed is large enough to hold the longest data record.

Example

See the example at the beginning of this section.

See Also

BTGetVariableRecLength

BTGetVariableRecPart

BTGetVariableRecLength / BTGetVariableRecPart

Syntax

```
procedure BTGetVariableRecLength; (IFBPtr : IsamFileBlockPtr; RefNr :  
  LongInt;  
                                     var Len : Word);
```

Purpose

Return the length of a variable length record.

Description

RefNr is normally obtained from a prior index operation. BTGetVariableRecLength takes just as much time as reading the entire record, so it should be used only when a buffer size must be determined before actually reading the record.

Example

```
var  
  BufPtr : Pointer;  
  Len : Word;  
const  
  BufSize : Word = 0;  
...  
BTGetVariableRecLength(PF, RefNr, Len);  
if Len > BufSize then begin  
  if BufSize <> 0 then  
    FreeMem(BufPtr, BufSize);  
  if MaxAvail >= Len then begin  
    BufSize := Len;  
    GetMem(BufPtr, BufSize);  
  end;  
end;  
BTGetVariableRec(PF, RefNr, BufPtr^, Len);
```

Demonstrates the technique to dynamically allocate a buffer that is as large as the largest variable record.

See Also

BTGetVariableRec

BTGetVariableRecPart

Syntax

```
procedure BTGetVariableRecPart; (IFBPtr : IsamFileBlockPtr; RefNr :  
  LongInt;  
                                     var Dest; var Len : Word);
```

Purpose

Read part of a variable length record.

Description

RefNr is normally obtained from a prior index operation. Unlike BTGetVariableRecord, this routine returns just the first Len bytes of the variable length record. This is intended primarily to allow reading the fixed portion of a record that has a variable length memo field at its end.

If there are fewer than Len bytes in the record, only those that exist are returned in Dest. BTGetVariableRecPart returns the actual number of bytes it could read in the Len parameter.

Example

```
var  
  Len : Word;  
...  
Len := SizeOf(PersonDefVar) - SizeOf(MemoField);  
BTGetVariableRecPart(PF, RefNr, P, Len);
```

BTGetVariableRecLength / BTGetVariableRecPart

Reads just the fixed portion of the PersonDefVar record described in the introduction.

See Also

BTGetVariableRec

BTGetVariableRecLength

BTGetVRecPartReadOnly / BTGetVRecReadOnly

Syntax

```
procedure BTGetVRecPartReadOnly; (IFBPtr : IsamFileBlockPtr; RefNr :  
  LongInt;  
                                var Dest; var Len : Word);
```

Purpose

Read part of a variable length record despite a record lock placed by another workstation.

Description

This routine works just like BTGetVariableRecPart except that it reads a record that is locked by another workstation.

In contrast to fixed length records, a variable length record cannot be read unless a record lock, fileblock lock, or read lock is placed by the calling workstation. This constraint is due to the fact that variable length records are composed of a chain of fixed length records, which might change in the midst of a read if a lock were not in place. If no lock is already in place, BTGetVRecPartReadOnly places a temporary read lock on the fileblock. If this read lock cannot be obtained immediately (because another station has a write lock on the fileblock), an IsamError of 10332 is generated.

As in the corresponding procedures for fixed length records, BTGetVRecPartReadOnly doesn't read the first four bytes of the data record if the record is locked by another workstation. The first four bytes of Dest are left uninitialized in this case, and IsamError returns the warning code 10205.

If you receive the 10205 warning, you cannot immediately determine whether the corresponding record is deleted. (Normally you would test the first four bytes for a non-zero value.) However, the following logic guarantees that the record is not deleted, even if warning 10205 occurs:

```
Read lock  
Find an associated key (using BTFindKey for example)  
Read record using BTGetVRecPartReadOnly  
Unlock
```

The data record is guaranteed to exist because its corresponding key exists; the key could not have been deleted in the meantime because a read lock was placed.

See Also

BTGetVariableRecPart

BTGetVRecReadOnly

Syntax

```
procedure BTGetVRecReadOnly; (IFBPtr : IsamFileBlockPtr; RefNr :  
  LongInt;  
                                var Dest; var Len : Word);
```

Purpose

Read a variable length record despite a record lock placed by another workstation.

Description

This routine works just like BTGetVRecPartReadOnly except that it returns the complete variable length record in Dest and the length of the record in Len.

See Also

BTGetVariableRec

BTGetVRecPartReadOnly

BTPutVariableRec

Syntax

```
procedure BTPutVariableRec(IFBPtr : IsamFileBlockPtr; RefNr : LongInt;  
                           var Source; Len : Word);
```

Purpose

Write a variable length record back to a data file.

Description

RefNr is normally obtained from a prior index operation. Len contains the actual length of the data record, which may differ from the original value. Even when Len is longer than the original value, the data reference number for the record remains unchanged after the call to BTPutVariableRec. Whenever the record uses more than one section, BTPutVariableRec relocates sections other than the first one, which creates free sections that will be reused later.

Never add a new data record to the data file with BTPutVariableRec. Use BTAddVariableRec instead. BTPutVariableRec can be used only to write a data record back to the same location from where it was read. Indiscriminately writing into the data file with BTPutVariableRec leads to unpredictable results.

In a network application, this procedure can be called successfully only when the network fileblock is locked for exclusive write access (level 1 or 2).

Example

```
BTGetVariableRec(PF, RefNr, P, Len);  
FillChar(P.Memo, SizeOf(MemoFieldArray), 0);  
BTPutVariableRec(PF, RefNr, P, LenOfData(P));
```

Reads a variable length record of the type used in the introductory example, clears the memo field, and puts the record back to the data file with its new shorter length.

See Also

BTAddVariableRec

BTReleaseVariableRecBuffer / BTSetVariableRecBuffer

Syntax

```
procedure BTReleaseVariableRecBuffer;;
```

Purpose

Deallocate the section buffer.

Description

This procedure frees the memory allocated by BTCreateVariableRecBuffer or BTSetVariableRecBuffer. It should be called only when all variable length record operations are complete, or if the current buffer must be enlarged manually.

Example

```
BTOpenFileBlock(P, 'ADDRESS', False, False, False, False);
if IsamVRecBufSize < BTDatRecordSize(P) then begin
  BTReleaseVariableRecBuffer;

  if not BTCreateVariableRecBuffer(P) then begin
    {Error handling};
  end;
end;
```

After a new fileblock is opened, this code assures that the variable record section buffer is large enough. If not, it releases the existing buffer (if any) and allocates a new larger one.

See Also

BTAdjustVariableRecBuffer

BTCreateVariableRecBuffer

Syntax

```
function BTSetVariableRecBuffer;(Size : Word) : Boolean;
```

Purpose

Allocate a section buffer of specified size.

Description

This function is an alternative to BTCreateVariableRecBuffer. Instead of taking the buffer size from internal fields in a fileblock, the size is directly specified with the call. This may be more convenient in some cases.

Never use BTCreateVariableRecBuffer and BTSetVariableRecBuffer one after the other. Only one of the two can be used. The memory used by calling either of these two routines is freed with BTReleaseVariableRecBuffer.

Example

```
if not BTSetVariableRecBuffer(256) then begin
  {Error handling}
end;
```

Allocates a section buffer without referring to any particular fileblocks. Remember that the size specified must be at least as large as the largest section length for any fileblock used with the VREC unit.

See Also

BTAdjustVariableRecBuffer

BTCreateVariableRecBuffer

Even in a well-planned database application design there comes a time when the database schema must be changed: a new field must be added to the record layout for a fileblock. The only thing that can be done is to restructure the fileblock: each record must be read from the old data file, converted to the new layout, and written to a new data file. Finally the index file must be rebuilt.

Of course there are other situations which also require some kind of restructuring, though perhaps not as pervasive as the previous example. A good example is purging deleted records. This could occur when many records are deleted and no significant number of records will be added in the near future. The fileblock's data file has a lot of wasted space in this situation.

The worst case reason for restructuring a database is when it is corrupted. Unfortunately, it's just too easy to corrupt a database, particularly one running on a network. If the system crashes, or the power goes out, or a program error causes the system to hang, information will probably remain buffered in memory. At this point the integrity of the database is highly suspect: was a new record written to disk, while its index entries remained buffered? Was the index partially updated, while some B-tree pages remained in memory? B-Tree Filer, unlike some database managers, can at least detect this form of corruption. Whenever data is buffered, B-Tree Filer writes out a flag that remains set until the buffers are guaranteed to be flushed (usually when the fileblock is closed). If this flag is found to be set when a fileblock is opened, B-Tree Filer returns an error code of 10010, indicating that the fileblock must be "rebuilt." For the best data security, you should restructure the data file (getting rid of deleted records) and then reindex. However, it is possible to get away with just a reindex.

Traditionally in B-Tree Filer, these rebuilding and reorganizing activities have been the province of the REBUILD/VREBUILD units, the REORG/VREORG units, and the FIXTOVAR unit. Versions 5.50 and later of B-Tree Filer contain a new pair of units, written to take care of fileblock restructuring: RESTRUCT and REINDEX. The traditional (V)REBUILD, (V)REORG and FIXTOVAR units still exist for backward compatibility, but these units now interface routines that are just shells around the routines in RESTRUCT and REINDEX.

The RESTRUCT unit interfaces the procedure RestructFileBlock. This routine takes care of copying one data file to another, skipping deleted records and possibly reformatting each record. The reformatting process is performed by a function that you supply. This user-written function must also state whether the record should be skipped or not. RestructFileBlock works with fixed length record and variable length record fileblocks. It uses an internal buffering scheme to read and write several records at a time, hence speeding up the overall process. RESTRUCT uses the first four bytes of the record to determine if it is deleted. If they are non-zero, the record is assumed to be deleted.

The REINDEX unit interfaces the ReIndexFileBlock routine. This routine creates a new index file for a fileblock. You can reorganize the index file completely by specifying a new number of indexes, and new index key structures. An internal buffering scheme is used to read several data records at once to speed up the reindexing process. The ReIndexFileBlock routine can be called without restructuring the data file first, hence making recoveries from index file corruption easier and faster (the traditional route was using Rebuild(V)FileBlock which rebuilds the data file too).

The following example shows how to use RESTRUCT and REINDEX to restructure and reindex an existing fileblock. Assume that you want to add a social security field at the end of the PersonDef

record type defined in _4.C. The new social security fields will be initialized to null strings, to be filled in by some interactive process later.

```

uses
    Filer, Restruct, ReIndex;
const
    Key1Len = 30; Key2Len = 5; Key3Len = 15;
type
    PersonDef = record
        ...
    end;
    NewPersonDef = record
        OldPers : PersonDef;
        SSN      : String[9];
    end;
function ConvertRec(var DatSOld; var DatSNew;
                    var Len : Word) : Boolean; far;
begin
    if (LongInt(DatSOld) <> 0) then {deleted record}
        ConvertRec := False
    else begin
        FillChar(DatSNew, SizeOf(NewPersonDef), 0);
        Move(DatSOld, DatSNew, SizeOf(PersonDef));
        ConvertRec := True;
    end;
end;
{$F+}
function CreateKey(var DatS; KeyNr : Word) : IsamKeyStr;
begin
    ...
end;
var
    Pages : LongInt; IID : IsamIndDescr; MsgFile : Boolean;
begin
    Pages := BTInitIsam(NoNet, 40000, 0);
    RestructFileBlock('TEST', SizeOf(NewPersonDef), sizeof(PersonDef)
                      False, 0, ConvertRec, BTNoCharConvert, Nil);
    if IsamOk then begin
        IID[1].KeyL := Key1Len; IID[1].AllowDupK := False;
        IID[2].KeyL := Key2Len; IID[2].AllowDupK := True;
        IID[3].KeyL := Key3Len; IID[3].AllowDupK := True;
        ReIndexFileBlock('TEST', 3, IID, False, CreateKey, True,
MsgFile,
                        BTNoCharConvert, Nil);
        if MsgFile then
            writeln('A message file has been created');
        end;
        if not IsamOK then begin
            {Error handling}
        end;
        BTExitIsam;
    end.

```

This is a complete albeit simple program to reconstruct the fileblock for the introductory example described in _4.C. CreateKey is the same function given there.

B-Tree Filer is first initialized by calling BTInitIsam. Then RestructFileBlock is called to restructure the TEST fileblock. Note how the record sizes of both the new and old record types are passed to the routine by use of the sizeof operator. The fourth parameter (False) indicates that the fileblock is a fixed length record fileblock. The next parameter is only used for variable length

records, so it is set to 0. ConvertRec is the address of the conversion routine. It checks for a non-deleted record, and if so copies the old record over to the new one (which was previously cleared).

Once the data file is restructured, the index file is rebuilt with ReIndexFileBlock. The usual IsamIndDescr array is built and the call to ReIndexFileBlock is made. The fifth parameter (CreateKey) creates a key string given a record and an index number.

The last parameter (Msgfile) is set True by ReIndexFileBlock if there is an error. In this case a message file (called TEST.MSG) is created.

RESTRUCT Declarations

Types

```
FuncChangeDatS; = function(var DatSold; var DatSNew; var Len :  
Word) : Boolean;
```

Function prototype for a routine to convert a record from an older format to a newer format. You must write a routine of this type when you restructure a fileblock by altering the record layout. The function must be passed to RestructFileBlock in the ChangeDatSFunc parameter.

DatSold is an untyped parameter to the buffer containing the old record. DatSNew is an untyped parameter to a buffer where the new record is to be placed. Generally you will typecast these two parameters to structures defining your own record layouts. The easiest way is to declare local variables as follows:

```
var  
  MyOldRec : MyOldRecordType absolute DatSold;  
  MyNewRec : MyNewRecordType absolute DatSNew;
```

On entry to the routine, Len is the number of bytes in the old record. On exit it must have the number of bytes in the new record. Since passing the length of the new record back to the caller only has meaning for variable length records, this value is ignored when restructuring a fixed length record file (the new record length is already known).

The function returns a boolean: True if the record is to be copied into the new data file, False if the record is to be skipped (presumably because it is deleted). If the routine returns False, the DatSNew buffer does not need to be set up with a new record.

If an unrecoverable error occurs in your routine, set IsamOk to False. This causes the RestructFileBlock procedure to exit immediately with IsamOk set to False and IsamError set to 10470.

The ProcChangeDatS pointer that is passed to ReOrg(V)FileBlock in the REORG unit must also be of this type.

REINDEX Declarations

Types

```
FuncBuildKey; = function(var DatS; KeyNr : Word) : IsamKeyStr;
```

Function prototype for a routine to create a key from a record. You must write a routine of this type when you reindex a fileblock through the ReIndexFileBlock routine. The name of your key building routine must be passed in the BuildKeyFunc parameter of the ReIndexFileBlock procedure.

DatS is an untyped parameter to a buffer containing the data record. Generally you will typecast this variable to a structure defining your record layout. The easiest way is to declare a local variable as follows:

```
var  
  MyRec : MyRecordType absolute DatS;
```

KeyNr is the number of the index for which the key string is required. Your routine will be called to create all the keys for index 1, then all the keys for index 2 and so on.

The routine returns the key string as the function result. If a null key is returned and the AddNullKeys constant in the FILER unit is False, the key is skipped, otherwise it is added to the index.

If an unrecoverable error occurs in your routine, set IsamOk to False. This causes the ReIndexFileBlock procedure to exit immediately with IsamOk set to False and IsamError set to 10470.

The FuncBuildKey pointer that is passed to ReOrg(V)FileBlock in the (V)REORG units and to ReBuild(V)FileBlock in the (V)REBUILD unit must also be of this type.

ChangeDatSNoChange

Syntax

```
function ChangeDatSNoChange (var DatSOld, var DatSNew, var Len :  
Word) : Boolean;
```

Purpose

Copy a record without reformatting.

Description

This routine is a default routine that can be passed as the ChangeDatSFunc parameter to RestructFileBlock. ChangeDatSNoChange does not reformat or alter the record. If the first 4 bytes of DatSOld are all zero, it just copies Len bytes from DatSOld to DatSNew and returns True. If the first 4 bytes of DatSOld are not all zero, it returns False.

This routine is used most often when you are using RestructFileBlock to purge deleted records from your data file. No record reformatting is done; the data file is just packed.

RestructFileBlock

Syntax

```
procedure RestructFileBlock; (FBlName : IsamFileBlockName; DatSLen : LongInt;
                             DatSLenOld : LongInt; VarRec : Boolean;
                             MaxDiffBytes : LongInt;
                             ChangeDatSFunc : FuncChangeDatS;
                             CharConvProc : ProcBTCharConvert;
                             CCHookPtr : Pointer);
```

Purpose

Restructure a data file.

Description

This is the main workhorse of the RESTRUCT unit. It reads the records from the data file of an existing fileblock, calls a routine to convert each record to a new format, and then writes the records to a new data file. An index file is not created, for that you need to call ReIndexFileBlock.

FBlName is the name of the existing fileblock. It can contain up to three DOS filenames (with optional drive and pathname specifications) separated by semicolons. No extensions need to be provided, because the default ones are used. RestructFileBlock appends the DatExtension constant string (usually 'DAT') to the first file name to obtain the actual name of the data file. The IxExtension (usually 'IX') is appended to the second name (or the first file name if a second is not specified) to get the index file name. The SavExtension (usually 'SAV') is appended to the third name (or the first file name if a third one is not specified) to get the save file name.

If VarRec is False, the restructuring of the data file is assumed to be for fixed length records. In this case, DatSLenOld is the record length of the data file in the existing fileblock, and DatSLen is the record length of the new data file. The safest way to get these two values is to use the SizeOf function on your old and new record types respectively. For a fixed length record, MaxDiffBytes is not used and you can pass the value 0 for it.

If VarRec is True, the restructuring of the data file is assumed to be for variable length records. In this case, DatSLenOld is the size of the fixed length section for the data file in the existing fileblock, and DatSLen is the size of the fixed length section for the new data file. See the discussion of section length in _6.B for more information.

MaxDiffBytes is a measure of how the size of the largest variable length record in the original data file will change when restructured. There are two ways of specifying this information:

- MaxDiffBytes is greater or equal to 0. In this case the value represents the maximum change in size of the records. In other words, the maximum length of a reformatted record is assumed to be the length of the original record plus MaxDiffBytes. If your records will be shrinking in size, set MaxDiffBytes to 0.
- MaxDiffBytes is less than 0. In this case, the negative of the value represents the absolute maximum size for the new records. The size of all new records will be less than or equal to -MaxDiffBytes.

MaxDiffBytes is used to allocate an internal buffer from the heap for the conversion process. This buffer is passed to the ChangeDatSFunc, and cannot be grown. It is therefore your responsibility to declare MaxDiffBytes so that you always have enough room in the buffer for your converted records.

ChangeDatSFunc is the function that does the actual conversion of an old record to the new record format. It must be declared FAR and be of type FuncChangeDatS.

The parameters CharConvProc and CCHookPtr are used to convert the old and new records to/from their external representation. They have exactly the same meanings as the corresponding parameters in the BTSetCharConvert procedure in the FILER unit. If you do not want to have a separate conversion routine, set CharConvProc to BTNoCharConvert and CCHookPtr to nil.

The RestructFileBlock routine performs the following steps:

1. Rename or copy the data file to the save file, *if* the save file does not exist.

RestructFileBlock

2. Call `BTCreateFileBlock` with name `FBName`, record length `DatSLen`, and `NumberOfKeys` indexes which are defined by `IID`. This new fileblock is opened in non-network mode with `BTOpenFileBlock`.
3. Open the save file with `IsamAssign` and `IsamReset`.
4. Read every record (except the first which is assumed to be a system record) from the save file and pass each one to the `ChangeDatSFunc` function. If this function returns `True`, the reformatted record is added to the new fileblock with `BTAddRec` or `BTAddVariableRec`.
5. Close the new fileblock.
6. Delete the save file.

There are several things to notice with this process. If an error occurs, the routine returns immediately with `IsamOk` set to `False` and `IsamError` set to the error code describing the error. The save file is not deleted (and should not be deleted by you), the new fileblock is erased, and the original data file is recreated if possible. You must then determine what caused the error, rectify it, and reset the files so that another attempt can be made. Another point to note is that the index file is created, but no keys are added to it.

Before calling `RestructFileBlock`, be sure to call `BTInitIsam` because `RestructFileBlock` uses B-Tree Filer services. If `VarRec` is `True`, be sure to create a variable length record buffer with `BTCreateVariableRecBuffer`.

You can provide a user status routine to display status during the restruct, compute statistics, or check the keyboard for a user-initiated abort of the restruct. Your routine must be declared far, must be global, and must match the following prototype declaration:

```
{ $F+ } {Also must be global}
procedure UserStatusRoutine(KeyNr : Integer; NumRecsRead : LongInt;
                             NumRecsWritten : LongInt; var Data; Len
                             : Word);
begin
    {display status...}
end;
{ $F- }
```

`KeyNr` is the index number currently being added. `NumRecsRead` is the number of records read up to this point, and `NumRecsWritten` is the number of records written. `Data` is the data record being worked on, and `Len` is the number of bytes of data in the record.

Set `IsamReXUserProcPtr` (declared in the Filer unit) to the address of your routine. The restruct routine works by copying each valid data record to a new data file. During this pass, `UserStatusRoutine` is called once for each record with `KeyNr` set to zero.

Example

See the example at the beginning of this section.

ReIndexFileBlock

Syntax

```
procedure ReIndexFileBlock; (FBlName : IsamFileBlockName; NumberOfKeys
: Word;
                           IID : IsamIndDescr; VarRec : Boolean;
                           BuildKeyFunc : FuncBuildKey; DelRecDupKey
: Boolean;
                           var MsgFileCreated : Boolean;
                           CharConvProc : ProcBTCharConvert;
                           CCHookPtr : Pointer);
```

Purpose

Create a new index file for an existing fileblock.

Description

This routine creates a new index file for a fileblock. The number of indexes and the definition of each index key structure can be changed from the ones defined in the original fileblock. The data file of the fileblock must be 'valid' in the sense that the system record of the data file must have valid values. If the data file is found to be damaged, ReIndexFileBlock returns with IsamError 10215, and you must rebuild the data file with RestructFileBlock.

FBlName is the name of the existing fileblock. It can contain up to three DOS filenames (with optional drive and pathname specifications) separated by semicolons. No extensions need to be provided, because the default ones are used. RestructFileBlock appends the DatExtension constant string (usually 'DAT') to the first file name to obtain the actual name of the data file. The IxExtension (usually 'IX') is appended to the second name (or the first file name if a second is not specified) to get the index file name. The third file name is ignored.

The parameters NumberOfKeys and IID determine the index structure in the same manner as they do for BTCreateFileBlock. NumberOfKeys is the number of indexes for the fileblock and IID is an array defining each index (its key length and whether duplicate key strings are allowed or not).

The VarRec parameter defines whether the fileblock contains variable length records (True) or fixed length ones (False).

BuildKeyFunc is the routine that creates a key string for each record in each index. It must be declared FAR and be of type FuncBuildKey. It is called once for every record and index combination, and must generate a key for the required index from the given record. The function can return a null string, in which case the AddNullKeys identifier in the FILER unit determines whether the null key is added to the index or not.

DelRecDupKey defines what happens if a key is generated for a primary index and that key already exists in the index. A primary index is one where all the keys are unique, and so the value of DelRecDupKey determines what happens when a duplicate key is added to such an index, and it fails. If the parameter is True, then the record and all keys that were added for it are deleted. If the parameter is False, then the key for this index is skipped. Either way a log file is created in the same directory as the data file with the same name, but with MsgExtension (usually 'MSG'). A log reference record is written to this file to describe the problem, and the index rebuild continues.

If a message log file is created, MsgFileCreated is set to True. You can then examine the log message file if you want to perform further manual reconstruction.

The parameters CharConvProc and CCHookPtr are used to convert the old and new records to/from their external representation. They have exactly the same meanings as the corresponding parameters in the BTSetCharConvert procedure in the FILER unit. If you do not want to have a separate conversion routine, set CharConvProc to BTNoCharConvert and CCHookPtr to nil.

Before calling ReIndexFileBlock be sure to call BTInitIsam, because ReIndexFileBlock uses B-Tree Filer services. If VarRec is True, be sure to create a variable length record buffer with BTCreateVariableRecBuffer.

ReIndexFileBlock

You can provide a user status routine to display status during the reindex, compute statistics, or check the keyboard for a user-initiated abort of the reindex. Your routine must be declared far, must be global, and must match the following prototype declaration:

```
{ $F+ } {Also must be global}
procedure UserStatusRoutine(KeyNr : Integer; NumRecsRead : LongInt;
                             NumRecsWritten : LongInt; var Data; Len
                             : Word);
begin
    {display status...}
end;
{ $F- }
```

KeyNr is the index number currently being added. NumRecsRead is the number of records read up to this point, and NumRecsWritten is the number of records written. Data is the data record being worked on, and Len is the number of bytes of data in the record.

Set IsamReXUserProcPtr (declared in the Filer unit) to the address of your routine. The reindex routine works by first adding all keys for index number one (KeyNr=1) and UserStatusRoutine is called for each key, then all keys for index number two, and so on.

The ReIndexFileBlock procedure is declared in the REINDEX unit.

Example

See the example at the beginning of this section.

REBUILD and VREBUILD are provided for compatibility with older versions of B-Tree Filer. Whenever possible, you should use the RESTRUCT and REINDEX units instead. REBUILD/VREBUILD read an existing data file (ignoring the index file), build a new data file using just the non-deleted records, and create a complete new index file.

Reconstruction of a fileblock is inherently a single-user activity. No workstation, including the current one, should have the fileblock open when a reconstruction starts. If the fileblock is held open by another user, RebuildFileBlock fails immediately because it opens the data file in a non-shareable mode. If the fileblock was opened by the current workstation, unpredictable problems may occur.

To use the rebuild routines in a program, just add REBUILD (for fixed length records) or VREBUILD (for variable length records) to your uses statement, following the FILER unit. VREBUILD also depends on the VREC unit. The interfaced routines in the REBUILD and VREBUILD units are actually just thin wrappers around calls to RestructFileBlock and ReIndexFileBlock from the RESTRUCT and REINDEX units respectively.

Note that a reconstruction using (V)REBUILD is possible only if the first four bytes of each data record were reserved for B-Tree Filer's use, and were initialized to zero by the application whenever it added a record.

The functions in REBUILD and VREBUILD are so similar that they are documented together in the following descriptions.

Also see AddNullKeys and IsamReXUserProcPtr in Chapter 5. Both of these identifiers affect the behavior of REBUILD and VREBUILD.

RebuildFileBlock / RebuildVFileBlock

Syntax

```
procedure RebuildFileBlock, (FBName : IsamFileBlockName; DatSLen : LongInt;  
LongInt;  
IsamIndDescr;  
NumberOfKeys : Integer; IID :  
FuncBuildKey : Pointer);  
procedure RebuildVFileBlock; (FBName : IsamFileBlockName; DatSLen : LongInt;  
LongInt;  
IsamIndDescr;  
NumberOfKeys : Integer; IID :  
FuncBuildKey : Pointer);
```

Purpose

Rebuild a fileblock.

Description

This routine is a simple wrapper around a call to RestructFileBlock (in the RESTRUCT unit) followed by a call to ReIndexFileBlock (in the REINDEX unit). See _6.C for more information on RESTRUCT and REINDEX.

In effect (leaving out the error handling) RebuildFileBlock contains the following code:

```
var  
  Dummy : Boolean;  
begin  
  RestructFileBlock(FBName, DatSLen, DatSLen, False, 0,  
    ChangeDatSNoChange, BTNoCharConvert, nil);  
  if IsamOK and (NumberOfKeys > 0) then  
    ReIndexFileBlock(FBName, NumberOfKeys, IID, False,  
      REINDEX.FuncBuildKey(FuncBuildKey), True,  
      Dummy);  
end;
```

RebuildVFileBlock contains the following code:

```
var  
  Dummy : Boolean;  
begin  
  RestructFileBlock(FBName, DatSLen, DatSLen, True, 0,  
    ChangeDatSNoChange, BTNoCharConvert, nil);  
  if IsamOK and (NumberOfKeys > 0) then  
    ReIndexFileBlock(FBName, NumberOfKeys, IID, True,  
      REINDEX.FuncBuildKey(FuncBuildKey), True,  
      Dummy);  
end;
```

Notice how the parameters to the Rebuild(V)FileBlock call are mapped onto the parameters for the RestructFileBlock and ReIndexFileBlock calls.

FuncBuildKey must be of type FuncBuildKey which is declared in the REINDEX unit.

You can provide a user status routine to display status during the rebuild, compute statistics, or check the keyboard for a user-initiated abort of the rebuild. Your routine must be declared far, must be global, and must match the following prototype declaration:

```
{ $F+ } {Also must be global}  
procedure UserStatusRoutine(KeyNr : Integer; NumRecsRead : LongInt;  
  NumRecsWritten : LongInt; var Data; Len  
  : Word);  
begin  
  {display status...}
```

RebuildFileBlock / RebuildVFileBlock

```
end;  
{ $F- }
```

KeyNr is the index number currently being added. NumRecsRead is the number of records read up to this point and NumRecsWritten is the number of records written. Data is the data record being worked on, and Len is the number of bytes of data in the record.

Set IsamReXUserProcPtr (declared in the Filer unit) to the address of your routine. The rebuild routines all work by first copying each valid data record to a new data file. During this pass, UserStatusRoutine is called once for each record with KeyNr set to zero. After the data is copied, all keys for index number one (KeyNr=1) are added and UserStatusRoutine is called for each key, then all keys for index number two, and so on.

REORG and VREORG are provided for compatibility with older versions of B-Tree Filer. Whenever possible, you should use the RESTRUCT and REINDEX units instead.

Reorganizing is a process similar to rebuilding, but one that is used in different circumstances. Reorganization is used when the size of each data record or its format must be changed. The B-Tree Filer REORG unit also provides a way to import some types of foreign data into the fileblock format.

The procedures ReorgFileBlock and ReorgVFileBlock, contained in the REORG and VREORG units, implement these ideas for fixed and variable length fileblocks, respectively. ReorgFileBlock reads any file of fixed length records, whether or not it was created by B-Tree Filer. (It always skips over the first record, however.) ReorgVFileBlock is specific to the format used by B-Tree Filer variable length records.

By taking advantage of user-supplied routines to perform the data validation and conversion of each record, the REORG and VREORG units are very flexible and should be useful in many situations.

The interfaced routines in the REORG and VREORG units are actually just thin wrappers around calls to RestructFileBlock and ReIndexFileBlock from the RESTRUCT and REINDEX units respectively.

The functions in REORG and VREORG are so similar that they are documented together in the following descriptions.

ReorgFileBlock / ReorgVFileBlock

Syntax

```
procedure ReorgFileBlock; (FBName : IsamFileBlockName; DatSLen :
LongInt;
                                NumberOfKeys : Integer; IID : IsamIndDescr;
                                DatSLenOld : LongInt; FuncBuildKey :
Pointer;
                                ProcChangeDatS : Pointer);
procedure ReorgVFileBlock; (FBName : IsamFileBlockName; DatSLen :
LongInt;
                                NumberOfKeys : Integer; IID : IsamIndDescr;
                                DatSLenOld : LongInt; MaxDiffBytes : Word;
                                FuncBuildKey : Pointer; ProcChangeDatS :
Pointer);
```

Purpose

Reorganize a fileblock.

Description

These routines are simple wrappers around a call to RestructFileBlock (in the RESTRUCT unit) followed by a call to ReIndexFileBlock (in the REINDEX unit). See _6.C for more information on RESTRUCT and REINDEX.

In effect (leaving out the error handling) ReorgFileBlock contains the following code:

```
var
    Dummy : Boolean;
begin
    RestructFileBlock(FBName, DatSLen, DatSLenOld, False, 0,
                    FuncChangeDatS(ProcChangeDatS),
    BTNoCharConvert, nil);
    if IsamOK and (NumberOfKeys > 0) then
        ReIndexFileBlock(FBName, NumberOfKeys, IID, False,
                        REINDEX.FuncBuildKey(FuncBuildKey), True,
    Dummy);
end;
```

ReorgVFileBlock contains the following code:

```
var
    Dummy : Boolean;
begin
    RestructFileBlock(FBName, DatSLen, DatSLenOld, True,
    MaxDiffBytes,
                    FuncChangeDatS(ProcChangeDatS),
    BTNoCharConvert, nil);
    if IsamOK and (NumberOfKeys > 0) then
        ReIndexFileBlock(FBName, NumberOfKeys, IID, True,
                        REINDEX.FuncBuildKey(FuncBuildKey), True,
    Dummy);
end;
```

Notice how the parameters to the Reorg(V)FileBlock call are mapped onto the parameters for the RestructFileBlock and ReIndexFileBlock calls.

ProcChangeDatS must be of type FuncChangeDatS which is declared in RESTRUCT.

FuncBuildKey must be of type FuncBuildKey which is declared in REINDEX.

You can provide a user status routine to display status during the reorg, compute statistics, or check the keyboard for a user-initiated abort of the reorg. Your routine must be declared far, must be global, and must match the following prototype declaration:

ReorgFileBlock / ReorgVFileBlock

```
{SF+} {Also must be global}  
procedure UserStatusRoutine(KeyNr : Integer; NumRecsRead : LongInt;  
                           NumRecsWritten : LongInt; var Data; Len  
  : Word);  
begin  
  {display status...}  
end;  
{SF-}
```

KeyNr is the index number currently being added. NumRecsRead is the number of records read up to this point, and NumRecsWritten is the number of records written. Data is the data record being worked on, and Len is the number of bytes of data in the record.

Set IsamReXUserProcPtr (declared in the Filer unit) to the address of your routine. The reorg routines work by first copying each valid data record to a new data file. During this pass, UserStatusRoutine is called once for each record with KeyNr set to zero. After the data is copied, all keys for index number one (KeyNr=1) are added and UserStatusRoutine is called for each key, then all keys for index number two, and so on.

This small unit has a single purpose: to convert an existing B-Tree Filer fileblock containing fixed length records to a new fileblock using variable length records. This action is especially appropriate if you want to add a new variable length field to an existing fileblock.

FIXTOVAR actually uses the RESTRUCT and REINDEX units to do most of its work, so the documentation for those units is also applicable here. FIXTOVAR interfaces a single high level routine, described on the following page.

FixToVarFileBlock

Syntax

```
procedure FixToVarFileBlock, (FBIName : IsamFileBlockName; DatSLenFix  
: LongInt;  
Integer;  
Pointer);  
DatSLenVar : LongInt; NumberOfKeys :  
IID : IsamIndDescr; FuncBuildKey :
```

Purpose

Convert a fileblock based on fixed length records to variable length records.

Description

Internally, this procedure first calls RestructFileBlock to expand and initialize each fixed length record so that the fileblock looks like a variable length fileblock, all of whose records are composed of a single section. Then it calls RestructFileBlock again (this time as a variable length fileblock) to "resection" each record into one or more smaller or larger pieces.

FBIName is the name of the existing fileblock. The name can contain up to three semicolon separated DOS filenames without extensions. (See the description of type IsamFileBlockName in chapter 5.) The first filename specifies the location of the existing data file and the destination for the new variable length data file. The second filename specifies the destination for the new index file (the old index file is not used). The third filename specifies the location for the SAV file, which is a backup copy of the old fixed-length data file. When the data file is large, it might be necessary to use a different disk drive for storage of the backup data. If the data file is not found, the routine will use the SAV file immediately. If the data file is found, it is renamed or copied onto the SAV file before proceeding.

No component file of the fileblock FBIName can be open at the time of the call to FixToVarFileBlock.

DatSLenFix is the length of the existing fixed length records. DatSLenVar is the desired section length for the variable length records. See _6.B for a discussion about how to pick DatSLenVar.

NumberOfKeys and IID specify the properties of the new index file. Note that it's possible to change the way the fileblock is indexed at the same time as the data file is reorganized. See the descriptions of type IsamIndDescr and procedure BTCreatFileBlock in Chapter 5 for more information.

The FuncBuildKey parameter must be of type FuncBuildKey which is declared in REINDEX.

You can provide a user status routine to display status during the conversion, compute statistics, or check the keyboard for a user-initiated abort of the conversion. Your routine must be declared far, must be global, and must match the following prototype declaration:

```
{ $F+ } {Also must be global}  
procedure UserStatusRoutine(KeyNr : Integer; NumRecsRead : LongInt;  
NumRecsWritten : LongInt; var Data; Len  
: Word);  
begin  
  {display status...}  
end;  
{ $F- }
```

KeyNr is the index number currently being added. NumRecsRead is the number of records read up to this point, and NumRecsWritten is the number of records written. Data is the data record being worked on, and Len is the number of bytes of data in the record.

Set IsamReXUserProcPtr (declared in the Filer unit) to the address of your routine.

FixToVarFileBlock calls RestructFileBlock, which calls the user status routine for each data record. Then it calls ReIndexFileBlock to create the index file. ReIndexFileBlock calls your status routine for each key in each index.

FixToVarFileBlock

Example

Suppose that you have a fileblock containing fixed length records of size 200 bytes. Suppose also that you plan to add a variable length memo field to the end of these records, and that the memo field might contain anywhere from 1 character (a terminator) to 1000 characters. Following the guidelines given in _6.B, a suitable section length for the resulting variable length fileblock is 256 bytes.

The first step of the conversion is to make a variable length fileblock that doesn't contain the new memo field. Use FixToVarFileBlock to perform this step:

```
FixToVarFileBlock(FB1Name, 200, 256, 0, IID, Nil);
```

There's no need to generate the index twice, so zero keys are specified. The IID variable can be uninitialized at this time since it won't be used. Similarly, the FuncBuildKey pointer can be Nil because it's never called.

The next step is to add the memo field, which is initially empty for all the records. In this example, a memo field is an array of characters terminated by ^Z. Hence, an empty memo field is indicated by a single ^Z character. To perform the next step, use ReorgVFileBlock:

```
ReorgVFileBlock(FB1Name, 256, NumberOfKeys, IID, 256, 8, @BuildKey,
                @ChangeDat);
```

The section length is not changing, so DatSLen equals DatSLenOld equals 256. This time the index file will be generated, so the desired number of keys, a properly initialized IID, and the address of an appropriate BuildKey routine are provided. MaxDiffBytes is 8, a small value that allows for the addition of the empty memo field (MaxDiffBytes could theoretically be as small as 1 here). The ChangeDat function would be written as follows:

```
{ $F+ }
function ChangeDat(var DatSold; var DatSNew; var Len : Word) :
Boolean;
begin
  Move(DatSold, DatSNew, Len); {transfer existing contents}
  NewRecType(DatSNew).MemoField[0] := ^Z; {initialize empty memo
field}
  inc(Len); {indicate that new record is one byte bigger than old}
  ChangeDat := True; {all records are transferred}
end;
```

The new record type is declared as follows:

```
type
  NewRecType =
    record
      ... all the old fields ...
      MemoField : array[0..1000] of char;
    end;
```

Although each record grows by one byte during the second step of the conversion, each one still fits within a single 256 byte section. Records require more than one section only if more than about 50 characters of text are added to the memo field.

B-Tree Filer requires that all keys take the form of a string. This offers the greatest flexibility in designing indexes: keys can be composed from portions of several record fields, they can be stored in compressed format, they can be logically inverted, and so on. Nevertheless, numeric values are often desirable for keys because they encode unique information in a compact format.

The NUMKEYS unit offers routines to convert numeric values to sortable key strings and back again. The strings produced by NUMKEYS are compact, and the exact numeric value can be recovered by calling the inverse routine. Although the encoded strings cannot be viewed directly, they are carefully designed to produce the correct sorting order when used in B-tree or other data files.

The NUMKEYS unit also contains several routines that can create "packed" key strings. Given a regular string of characters, these routines return a compressed string, one in which each character is represented in either 4, 5 or 6 bits, rather than 8. In cases where the packed key strings can be used, you can thus obtain a savings of roughly 25-50% (e.g., a 50-character string can be represented in 25, 32, or 38 characters). As with the numeric key strings, these packed key strings cannot be viewed directly, but they produce the correct sorting order when used in a B-tree. Complementary routines are also provided to unpack a packed key string.

NUMKEYS also contains one routine to invert the value of a key string. Such a key will sort in descending order. Inverting it a second time returns the original string.

Note that NUMKEYS does not support the AsciiZeroKeys define of the FILER unit. NUMKEYS is likely to generate key strings that contain embedded null characters, which cannot be reliably converted to ASCIIZ strings.

Special thanks to Scott Bussinger for the ideas behind some of the routines in this unit.

Declarations

Types

```
String1 = String[1];
String2 = String[2];
String4 = String[4];
String5 = String[5];
String6 = String[6];
String7 = String[7];
String8 = String[8];
String9 = String[9];
String10 = String[10];
```

String types associated with the various numeric formats. Variables of type ShortInt are converted to type String1; Integer and Word are converted to type String2; LongInts to String4; Reals to String6; and Extendeds to String10. The intermediate string lengths (String7..String9) are provided to allow shorter keys when the accuracy of floating point variables can be reduced.

DescendingKey

Syntax

```
function DescendingKey; (S : String; MaxLen : Byte) : String;
```

Purpose

Logically invert a string so that it will sort in descending order.

Description

Each character in the string S is inverted (using the assembly language NOT instruction) and the resulting string is padded to length MaxLen with \$FF bytes. This transformation causes the returned string to sort in exactly the reverse order of the original string.

Repeating the operation returns the original string, except that the final length will always be MaxLen and any padded bytes will contain the null character. To avoid this effect, it's always best to pad the original string with blanks to the maximum desired length.

The InvertString function in the ISAMTOOL unit (_6.H) works the same as DescendingKey except that InvertString does not pad to the maximum length with \$FF.

Example

See the example for InvertString.

Numbers to Keys

Syntax

```
function ShortToKey; (S : ShortInt) : String1;  
    {-Convert a shortint to a string}  
  
function ByteToKey; (B : Byte) : String1;  
    {-Convert a byte to a string}  
  
function IntToKey; (I : Integer) : String2;  
    {-Convert an integer to a string}  
  
function WordToKey; (W : Word) : String2;  
    {-Convert a word to a string}  
  
function LongToKey; (L : LongInt) : String4;  
    {-Convert a longint to a string}  
  
function RealToKey; (R : Real) : String6;  
    {-Convert a real to a string}  
  
function ExtToKey; (E : Extended) : String10;  
    {-Convert an extended to a string}  
  
function BcdToKey; (var B) : String10;  
    {-Convert a BCD real to a string}
```

Purpose

Convert numeric types to key strings.

Description

Each of these routines converts a numeric variable of the specified type to a string. The string is intended to be used as a key for a B-Tree Filer fileblock; it is not meant to be displayed, as the individual characters have values spanning the complete range from 0 to 255. String comparison operators (" $<$ ", " $>$ ", etc.) will return the expected results when comparing two converted strings.

Each returned string has a length equal to the maximum allowed for its type. For example, RealToKey always returns strings that are 6 characters long. For numeric variables of type ShortInt, Integer, Word, LongInt, and Real, the entire string must be stored in order to preserve the meaning of the number.

ExtToKey is provided only if the NUMKEYS unit is compiled with the {\$N+} (numeric coprocessor) option enabled. This one routine can be used to convert IEEE floating point numbers of type Single, Double, Extended, and Comp to strings. Not all characters of the returned string are significant for the less precise types. You can use any of the following string types to hold the results of ExtToKey without losing any precision:

Single	: String5 (min) - String10 (max)
Double	: String9 (min) - String10 (max)
Extended	: String10 (min/max)
Comp	: String10 (min/max)

Slightly shorter strings (one less than the recommended minimum) can be used for Singles, Doubles, and Extendeds if you are willing to sacrifice some precision. It is strongly recommend that you always use a String10 for Comps.

The IEEE floating point standard embodied in the 8087 chip and its descendants defines several special classes of numbers that aren't normally part of formal mathematics, but that are useful in the context of numerical analysis. Examples of such numbers are NaN (Not a Number), INF (positive infinity), and -0 (zero with a negative flavor).

When the floating point processor (or emulator) deals with these special numbers, it applies special rules. For example, it returns True when testing the assertion ($0 = -0$), and the result of adding NaN to any normal number is NaN.

When such numbers are converted to strings, the special meaning is lost. NUMKEYS converts 0 to a different string than it does -0. Hence, although a numeric comparison of 0 and -0 returns

Numbers to Keys

equality, a string comparison does not. The string comparison does return the arguably logical result that -0 is less than 0. Note that NaNs cannot be ordered; comparing two NaNs only produces the values equal or not equal, a NaN cannot be said to be greater or less than another NaN. The converted NaN strings are sortable, probably with results that you do not expect. So you will probably not want to use NaNs for keys.

These considerations should not pose a concern in normal situations. The special meanings are restored by the reverse conversion routines.

The parameter passed to BcdToKey must be a variable of type BCD as defined by the TPBCD and OPBCD units from Turbo Professional and Object Professional, respectively.

Example

Consider the PersonDef record described in 4.C and assume that you want an index for the Age field of the record, which is of type Integer. This index would have a maximum key length of 2 and would allow duplicates. To add a key for this index, make the following call:

```
BTAddKey(PF, KeyNr, RefNr, IntToKey(P.Age));
```

To find a key based on age, we'd also need to convert an integer variable containing the age to search for into a string by calling IntToKey.

Keys to Numbers

Syntax

```
function KeyToShort; (S : String1) : ShortInt;  
    {-Convert a string to a shortint}  
  
function KeyToByte; (S : String1) : Byte;  
    {-Convert a string to a shortint}  
  
function KeyToInt; (S : String2) : Integer;  
    {-Convert a string to an integer}  
  
function KeyToWord; (S : String2) : Word;  
    {-Convert a string to a word}  
  
function KeyToLong; (S : String4) : LongInt;  
    {-Convert a string to a longint}  
  
function KeyToReal; (S : String6) : Real;  
    {-Convert a string to a real}  
  
function KeyToExt; (S : String10) : Extended;  
    {-Convert a string to an extended}  
  
procedure KeyToBcd; (S : String10; var B);  
    {-Convert a string to a BCD real}
```

Purpose

Convert a string back to a numeric type.

Description

These routines undo the original conversion to a key string. As long as the full length of the converted string was stored, the numeric value is restored exactly.

KeyToExt is provided only if the NUMKEYS unit is compiled with the {\$N+} (numeric coprocessor) option enabled.

A variable of type BCD must be passed in the untyped parameter of KeyToBcd.

Pack Key Strings

Syntax

```
function Pack4BitKey;(Src : String; Len : Byte) : String;
    {-Pack Src into sequences of 4 bits}

function Pack5BitKeyUC;(Src : String; Len : Byte) : String;
    {-Pack Src into sequences of 5 bits. Alphas converted to upper
    case.}

function Pack6BitKeyUC;(Src : String; Len : Byte) : String;
    {-Pack Src into sequences of 6 bits. Alphas converted to upper
    case.}

function Pack6BitKey;(Src : String; Len : Byte) : String;
    {-Pack Src into sequences of 6 bits}
```

Purpose

Pack a string into less space.

Description

These routines all return a packed key string in which each character in the original string (Src) is represented in 4, 5 or 6 bits rather than 8. The packed key string is therefore roughly 25-50% smaller than the original.

Len is the length of the packed string to be returned. The proper value for this parameter depends on (1) the maximum length of the unpacked string passed as the Src parameter and (2) the number of bits used for packing. The formulas used to calculate what Len should be are:

```
4-bits: (MaxLength) div 2 + Ord(MaxLength mod 2 <> 0)
5-bits: (MaxLength*5) div 8 + Ord((MaxLength*5) mod 8 <> 0)
6-bits: (MaxLength*6) div 8 + Ord((MaxLength*6) mod 8 <> 0)
```

For example, if you are using Pack5BitKeyUC (which uses 5 bits) and the maximum length of an unpacked key string is 50, then Len is calculated as follows:

```
Len = (50*5) div 8 + Ord((50*5) mod 8 <> 0)
     = 250 div 8 + Ord(250 mod 8 <> 0)
     = 31 + Ord(2 <> 0)
     = 31 + 1
     = 32
```

Although this computation seems overly complicated, keep in mind that the compiler can make the necessary calculations for you at compile time. For example:

```
const
    UnpackedLen = 50;
    PackedLen = (UnpackedLen*5) div 8 + Ord((UnpackedLen*5) mod 8 <>
    0);
```

Each of these routines makes important assumptions about what kinds of characters are in Src. You must therefore be certain that you choose the routine which is most appropriate for a given situation. The table below shows the set of characters that each routine can represent in the bits allotted it. In each case the numbers on the right indicate the actual values used to represent each character.

Pack4BitKey		Pack5BitKeyUC	
-----		-----	
'('..'')'	1-5	'A'..'Z'	1-26
'0'..'9'	9-15	'a'..'z'	1-26
all others	0	all others	0
Pack6BitKeyUC		Pack6BitKey	
-----		-----	
'!'..' ' '	1-63	'0'..'9'	1-10
'A'..'Z'	33-58	'A'..'Z'	11-36

Pack Key Strings

'a'..'z'	33-58
all others	0

'a'..'z'	37-62
all others	0

As you can see, each packing scheme has distinct advantages and disadvantages:

Pack4BitKey is best for strings that contain only numbers plus the common punctuation marks associated with numbers.

Pack5BitKeyUC can represent only upper case alphabetic characters. (Lower case alphabetics are automatically converted to upper case.)

Pack6BitKeyUC is similar to Pack5BitKeyUC, but it can represent numeric characters ('0'..'9') and punctuation marks (except "'") as well as upper case alphabetic characters. (Lower case alphabetics are automatically converted to upper case.) Although it can represent more characters than Pack5BitKeyUC, Pack6BitKeyUC provides a lower degree of compression (roughly 25%).

Pack6BitKey gives the same degree of compression as Pack6BitKeyUC, but it can represent only numeric characters ('0'..'9') and alphabetics, not punctuation marks. Its chief advantage over the other two routines is that it can distinguish between upper and lower case alphabetic characters. In situations where case sensitivity is important, it is thus the only one of the three routines that can be used.

Unpack Key Strings

Syntax

```
function Unpack4BitKey; (Src : String) : String;  
    {-Unpack a key created by Pack4BitKey}  
  
function Unpack5BitKeyUC; (Src : String) : String;  
    {-Unpack a key created by Pack5BitKeyUC}  
  
function Unpack6BitKeyUC; (Src : String) : String;  
    {-Unpack a key created by Pack6BitKeyUC}  
  
function Unpack6BitKey; (Src : String) : String;  
    {-Unpack a key created by Pack6BitKey}
```

Purpose

Unpack key strings.

Description

These routines unpack key strings created by Pack4BitKey, Pack5BitKeyUC, Pack6BitKeyUC, and Pack6BitKey respectively.

The returned strings are not necessarily identical to those originally passed to the packing routines. Any characters that the packing routines could not store in the allotted bits are returned as spaces, and lower case alphabets ('a'..'z') are returned as upper case characters unless Pack6BitKey was used.

The length of the unpacked string may also be greater than that of the original (if the original was shorter than the maximum length used to calculate the Len parameter passed to the packing routine); if so, the extra characters are all blanks.

Convert C Keys

Syntax

```
function CStyleNumKey;(S : String) : String;  
    {-Convert Pascal-style numeric key into a C-style one}  
  
function PascalStyleNumKey;(S : String) : String;  
    {-Convert C-style numeric key into a Pascal-style one}  
  
function CStyleDescendingKey;(S : String) : String;  
    {-Convert S to a descending key using C-style algorithm}
```

Purpose

Convert to/from C-Style keys.

Description

These routines enable the practical use of C-style keys to enhance B-Tree Filer's compatibility with B-Tree Filer for C. Generally you will need to use them only if you are using B-Tree Filer with the ASCIIZeroKeys define set.

The need for these routines arises because C-style keys (also known as ASCIIZ strings) cannot have embedded null characters, since the null character is used as a terminator. The normal NUMKEYS routines generate strings with embedded nulls, and hence if you are using C-style keys, you must convert them somehow.

CStyleNumKey converts a string returned from XxxToKey into a form without embedded nulls, suitable for using in an index file with C-style keys.

PascalStyleNumKey does the opposite conversion, creating a string that can be used in the KeyToXxx routines. Use this routine as a first step to convert a numeric C-style key read with a routine such as BTNextKey into a numeric value again.

Whenever you plan to create a fileblock that will be accessible from B-Tree Filer for C, you should either avoid using any function from the NUMKEYS unit, or convert the string to and from a C-style key by using these two functions; CStyleNumKey for adding the key or searching for the key and PascalStyleNumKey for converting a searched key back again.

CStyleDescendingKey creates a descending key suitable for a C-style key.

The C-style strings created using these routines are somewhat longer than their equivalent Pascal-style strings, because the C-style string uses only 7 out of each 8 bits for data storage. The eighth bit is always set to 1 to guarantee that each byte is non-zero. This is the reason that there are no equivalent routines to PackXxxKey or UnpackXxxKey.

This is a collection of miscellaneous routines to extend the FILER unit, which for various reasons were felt to be unsuitable for inclusion in the FILER unit itself. Here's a brief overview of the routines interfaced by ISAMTOOL:

```
ExtendHandles
  Increase the number of file handles available to an application

InvertString
  Return a string that will sort in descending order

IsamErrorMessage
  Return a text string describing the specified error code
```

IsamErrorMessage can return strings in either English or German. By default it returns English strings. To enable German strings, edit ISAMTOOL.PAS and activate the GermanMessage conditional define. You can enable both EnglishMessage and GermanMessage if you like. Also change the typed constant UsedErrorMessage (which defaults to English) to German to complete the language switchover. If you enable both languages, you can switch between them at runtime.

ExtendHandles

Syntax

```
procedure ExtendHandles; (NumHandles : Word);
```

Purpose

Increase the number of usable file handles.

Description

ExtendHandles depends on a DOS function available for the first time with DOS 3.3. It exits with IsamError set to 10190 if the DOS version is older.

NumHandles contains the desired number of file handles. The maximum number of handles is 255, but a bug in DOS 3.3 limits the actual maximum to 254. (The NumHandles parameter is of type Word for compatibility with the Microsoft Windows 3.0 function SetHandleCount that extends the number of handles.)

The CONFIG.SYS file on the system where the application runs must have a FILES= entry that is at least as large as the value of NumHandles. Because of a bug in DOS 3.3, the FILES= entry should generally be set to at least NumHandles+1. Note however that other drivers and TSRs on the system may have file handles allocated out of this system-wide pool, so it is best to set the FILES= parameter generously.

For workstations running under Novell NetWare, there is another configuration parameter that limits the number of available file handles to a default of 40. For recent versions of NetWare (2.1 or higher), the FILE HANDLES= setting in SHELL.CFG (or NET.CFG) specifies the maximum number of open files for a given workstation. See your NetWare Supervisor Reference for additional information. For earlier versions of NetWare, the network shell must be patched to allow more open handles. Contact Novell for further information about the patch.

A new Handle table is allocated when ExtendHandles is called. The amount of memory required is NumHandles+47 bytes. To obtain this memory in a real mode program, ExtendHandles either uses existing free DOS memory or shrinks the Turbo Pascal heap to make some DOS memory available. In a protected mode program, ExtendHandles uses existing DOS memory. In Windows, a special Windows API call is made. This memory is automatically deallocated by the operating system when the program ends.

ExtendHandles transfers any open files into the new handle table. Nevertheless, this routine should be called at the beginning of the program, preferably before any files are opened.

Note that DOS reserves the first five handles for standard devices. Hence, when using the standard 20 entry handle table, only 15 handles are available for other uses. B-Tree Filer needs two file handles for every single user fileblock opened in normal mode. In save mode, or when opened for network access, a fileblock requires three file handles.

ExtendHandles uses a documented DOS call supported only in versions 3.3 and later. For applications that must run under earlier versions of DOS, use the public domain unit EXTEND, available on many bulletin boards and on the BPASCAL forum of CompuServe. EXTEND works for DOS versions 2.0 and later by using different techniques depending on the DOS version.

Example

Assume that you want to have 10 network fileblocks open at the same time, and still have 3 handles available for other uses. The number of required handles is therefore $10 \times 3 + 3 + 5 = 38$. The FILES= statement in CONFIG.SYS should be set to at least 39 (but preferably more).

After a successful call to the following code, the program will be able to open 38 or 39 handles:

```
ExtendHandles(38);
if not IsamOK then begin
  {Error handling}
end;
```

InvertString

Syntax

```
{IFDEF LengthByteKeys}
procedure InvertString; (var Dest : String; Source : String; MaxLen :
Byte);
{IFDEF ASCIIZeroKeys}
procedure InvertString (var Dest; var Source; MaxLen : Byte);
```

Purpose

Invert a string to generate a descending key string.

Description

This procedure has two properties that are useful in manipulating strings for fileblock indexes:

1. Given two strings A and B such that $A < B$, InvertString returns strings such that $\text{InvertString}(A) > \text{InvertString}(B)$.
2. Applying InvertString twice returns the original string, i.e., $\text{InvertString}(\text{InvertString}(A)) = A$.

By taking advantage of these two properties, you can easily create fileblock indexes that are stored in descending order and also allow the original ASCII strings to be recovered.

MaxLen specifies the maximum length of a string in the index, typically the value passed in the IsamIndDescr variable when the fileblock was created. If the actual length of Source is larger than MaxLen, InvertString returns the empty string.

The strings returned by one call to InvertString are not intended for display. Call InvertString a second time before viewing a key string.

There are two versions of this routine, one for Pascal-style strings, the other for C-style strings.

The version that is compiled and used depends on which of the two compiler defines (LengthByteStrings or ASCIIZeroStrings) is enabled in BTDEFINE.INC.

Example

```
{IFDEF LengthByteStrings}
BTAddRec(PF, RefNr, PersonRec);
{Add a key to index 1, which is stored in descending order}
DKey := BuildKey(PersonRec, 1);
BTAddKey(PF, 1, RefNr, InvertString(DKey, MaxKeyLen));
...
Write('Enter key to search for: ');
ReadLn(DKey);
DKey := InvertString(DKey, MaxKeyLen);
BTSearchKey(PF, 1, RefNr, DKey);
if IsamOK then
    Writeln('The closest key found is ', InvertString(DKey,
MaxKeyLen));
```

This example sketches the creation of a descending index for a fileblock. The normal ASCII key (a last name, for example) returned by the user-defined BuildKey routine is inverted before the key is added to the index file with BTAddKey. It is assumed that index 1 of fileblock PF is allowed to store key strings up to MaxKeyLen characters long. Before searching for a key string, it is inverted. When a match is found, the resulting string is inverted before it is displayed.

IsamErrorMessage

Syntax

```
function IsamErrorMessage, (ErrorNr : Integer) : String;
```

Purpose

Return a message for the given error code.

Description

The value passed in ErrorNr is typically the value in the global variable IsamError after detecting that IsamOK is False.

IsamErrorMessage does not incorporate the file name of the associated fileblock. It is the application's responsibility to do so (see function BTDataFileName in the FILER unit). The messages returned by IsamErrorMessage match those found in Appendix A.

Be warned that calling IsamErrorMessage links about 2000 bytes of strings and code into your application.

Note the comments at the beginning of this section regarding support for different languages.

Example

```
S := IsamErrorMessage(10190);
```

Assigns the following text to the string S: 'Extend handle function requires DOS 3.3 or later'.

Although sorting is a common database application, the supplied sort units are not an integral part of the FILER unit in the same way that VREC or the other utility units are. The sort units can be used to sort any kind of data, including B-Tree Filer data files. Some care is necessary when sorting data files, however. See "Sorting B-Tree Filer fileblocks" later in this section for more details.

The virtual sort units, MSORT and MSORTP, are an implementation of the "sort" algorithm. A merge sort allows you to sort more items than will fit in RAM at once, by sorting manageable portions of the input first, then merging these pre-sorted lists to form the final output. In theory, MSORT can sort two billion records at once. In practice, the number of records is limited by available disk, heap, or expanded memory () capacity, and the time you're willing to wait.

MSORT and MSORTP are used in a manner similar to the SORT unit of the Borland Database Toolbox, the TPSORT unit of Turbo Professional and the OPSORT unit of Object Professional. If you've used any of these units before, you'll be quick to understand the methodology of MSORT and MSORTP.

The main difference between these two implementations of a merge sort lies in the targets they have been aimed for. MSORT was designed and optimized for real mode. It uses EMS for storing the partially sorted merge files, or disk as a last resort. MSORTP was expressly written for a protected mode environment (DOS or Windows), and uses the capabilities of that environment to optimize the sort process. MSORT can only be compiled for real mode, whereas MSORTP can be compiled by Borland Pascal to all three targets (real, protected, and Windows). If MSORTP is compiled for real mode, it uses heap memory (not expanded or extended memory) and the disk, and hence is slower than MSORT when run on a machine that has EMS memory.

MSORT exports several routines: AutoSort, DoSort, PutElement, GetElement, and AutoSortInfo. AutoSort is a high level sorting routine that automatically calculates various parameters needed by the sort system. DoSort is a lower level routine that gives you full control over all sort parameters. In most cases, you will use AutoSort, since it calculates these parameters and calls DoSort. AutoSort requires several parameters, as described in detail in the following sections. Two of the parameters tell it about the elements to be sorted--their size and the number to expect. One parameter specifies a pathname for any temporary files created by the sort process. Three parameters provide addresses of routines that MSORT calls during the course of execution--one routine that loads elements into MSORT's data structures, another that MSORT calls to compare pairs of elements, and a final one to return the sorted elements to your program.

The routines you provide to load and retrieve elements call two of the other functions exported by MSORT. For each element to be sorted, your load routine calls PutElement to enter the data. For each sorted element retrieved, your routine calls GetElement.

The following small example shows how to organize an application that uses MSORT.

```

program Example;
uses
  MSORT;
var
  Status  : MSortStatus;
  NumToDo : Word;

  {Routines called by MSORT must be both far and global}
  {$F+}
  procedure InsertElements;
    {-Pass every element to be sorted into the Msort unit}
  var
    W, I : Word;
  begin
    for I := 1 to NumToDo do begin
      W := Random(60000);
      if not PutElement(W) then begin
        Writeln('PutElement Error. ');
        Halt;
      end;
    end;
  end;

  procedure ExtractElements;
    {-Return each sorted element from the Msort unit}
  var
    W : Word;
  begin
    while GetElement(W) do
      Writeln(W);
  end;

  function Less(var X,Y) : Boolean;
    {-Compare elements being sorted}
  begin
    Less := Word(X) < Word(Y);
  end;
  {$F-}

```

```

begin
  Write('Enter number of Integers to sort: ');
  ReadLn(NumToDo);
  Status := AutoSort(NumToDo, SizeOf(Integer), '@InsertElements,
@Less,
                                @ExtractElements);
  case Status of
    MSortSuccess      : Writeln('Success');
    MSortOutOfMemory  : Writeln('Insufficient memory');
    MSortDiskError    : Writeln('Disk Error: ', MSortIOResult);
    MSortOutOfDisk    : Writeln('Insufficient disk space for
merge');
    MSortEMSError     : Writeln('EMS error');
    else               : Writeln('Unknown error: ', Ord(Status));
  end;
end.

```

Note that the InsertElement, ExtractElements, and Less routines must be declared both far and global. Not observing this requirement will probably lead to a program crash.

Sorting B-Tree Filer Fileblocks

In some cases, you may want to arrange the data records of a B-Tree Filer fileblock in physically sorted order. Typically you wouldn't want to arrange the records in the same order that an index already provides, since the index allows you to scan the fileblock in sorted order already. In some cases, it may be convenient to place the data file in physical order. Also, it can be faster to add keys to an index when the records are processed in sorted order.

If you do decide to sort a fileblock, you must consider some additional issues. The first issue is this: each deleted record within the data file contains a special tag used to manage a linked list of free space within the data file; sorting the data file corrupts this linked list. Of course, sorting the data file also invalidates the index file, which refers directly to the record numbers. And finally, the first record in the data file contains internal system information; it must remain first even after the sort. Hence, when sorting a B-Tree Filer fileblock, you must take special precautions, as shown in the following example.

```

type
  MyRecType =
    record
      Dele : LongInt;           {Deletion tag}
      {Other fields}
    end;
const
  DataName = 'MYDATA';        {Name of fileblock to
sort}
  NumberOfKeys = 2;           {Number of keys in
fileblock}
{$F+}
procedure InsertElements;
  {-Pass every element to be sorted into the Msort unit}
var
  RecNum : LongInt;
  IFBPtr : IsamFileBlockPtr;
  Rec : MyRecType;
begin
  BTOpenFileBlock(IFBPtr, DataName,
                  False, False, False, False); {Open only if closed}

```

```

    for RecNum := 1 to BTFFileLen(IFBPTr) do begin
        BTGetRec(IFBPTr, RecNum, Rec, False);    {Read each record}
        if Rec.Dele = 0 then                      {Make sure it's not
deleted}
            if not PutElement(Rec) then begin      {Pass record to MSORT
unit}
                BTCloseFileBlock(IFBPTr);          {Close up if sort
error}
                Exit;
            end;
        end;
        BTCloseFileBlock(IFBPTr);                {Close up}
    end;

procedure ExtractElements;
    {-Return each sorted element from the Msort unit}
var
    RecNum : LongInt;
    KeyNum : Integer;
    KeyStr : IsamKeyStr;
    IFBPTr : IsamFileBlockPtr;
    IID : IsamIndDescr;
    Rec : MyRecType;
begin
    {Initialize index descriptor, IID}
    ...
    {Recreate the fileblock}
    BTCreateFileBlock(DataName, SizeOf(MyRecType), NumberOfKeys,
IID);
    BTOpenFileBlock(IFBPTr, DataName, False, False, False, False);
    while GetElement(Rec) do begin                {Get each sorted
element}
        BTAddRec(IFBPTr, RecNum, Rec);            {Add the record}
        for KeyNum := 1 to NumberOfKeys do begin
            KeyStr := ...                          {Build the key
string}
        BTAddKey(IFBPTr, KeyNum, RecNum, KeyStr); {Add the key}
        end;
    end;
    BTCloseFileBlock(IFBPTr);                      {Close up}
end;

function Less(var X,Y) : Boolean;
    {-Compare elements being sorted}
begin
    {Return True if MyRecType(X) < MyRecType(Y)}
end;
{$F-}

```

InsertElements skips record 0, which holds internal data describing the fileblock. It reads all the remaining records, but does not pass deleted records to MSORT, since sorting deleted records would be a waste of time. ExtractElements creates a new fileblock, in this case overwriting the old one (which probably isn't a good idea in a real application). It then gets each record back in sorted order from MSORT and adds it and its keys to the new fileblock. The Less function works like any other MSORT Less function, since deleted records have already been filtered out.

Although this example shows an outline of the required steps, it does not include error checking. Each call to a B-Tree Filer routine should be followed by a check of IsamOK.

This method of adding the keys to the fileblock is not the most efficient. It would be faster to cycle through the records adding all the keys for index 1, cycle through all the records again adding all the keys for index 2, and so on. Or you could use the REINDEX unit to do this process for you.

Memory Management

MSORT makes use of a special unit called TPALLOC, supplied with B-Tree Filer. TPALLOC contains heap management routines that allow the allocation of data structures greater than 64KB. Structures this large may be needed by the sort system to manage internal data. Specifically, MSORT uses one structure called the "buffer" for an in-memory sort of the largest group of elements that will fit at one time.

By default, the run buffer is allocated as a large contiguous array (which can easily exceed 64KB). This speeds up the allocation and deallocation of this buffer, and simplifies the code required to manage it. There is one potential disadvantage to this technique: if the heap is fragmented, there might not be a single block of heap space large enough to allocate the run buffer contiguously.

Should this fragmentation issue become a problem, you can change MSORT's memory allocation behavior. To do so, remove the compilation directive BigHeap; found near the top of MSORT.PAS and recompile the unit. Fragmentation will no longer be an issue, but a new constraint will apply. Because it can take a lot of time to deallocate potentially thousands of pointers, MSORT uses a special kind of Mark and Release. This technique works correctly only if two conditions are met:

1. the caller of the MSORT routines does not perform heap allocation and deallocation while the sort is in progress--that is, within the user-defined procedures of the sort.
2. the heap is not fragmented or Turbo Pascal 6 is not used. The special Release routine used within MSORT will not work with the Turbo Pascal 6 heap manager when the heap is fragmented.

If the application needs to allocate heap space while the sort is in progress, it must take one precaution in addition to leaving the BigHeap directive defined. The MaxHeapToUse typed constant described below informs MSORT of how much heap space it can use while sorting. The default value allows MSORT to use all available heap space. You must set MaxHeapToUse to a smaller value prior to calling DoSort or AutoSort to guarantee that heap space will be available during the sort.

MSORT also uses memory if it is available and needed. (See AutoSort for more information on how it determines whether EMS will be used.) If an application uses EMS memory itself, it should allocate any pages it needs prior to calling the sort routines. The user-defined Get, Put, and Less routines can use EMS themselves, even transferring data from EMS directly to MSORT. MSORT also sets its mapping context every time it reads or writes EMS, so the user routines need not save and restore this context themselves. MSORT automatically frees any EMS pages it uses upon completion of the sort, or in its exit procedure if a runtime error occurs during the sort process. EMS usage can be disabled by setting the UseEms constant to False.

Disk Usage

When there are too many elements to sort in memory, MSORT creates temporary files to hold the overflow. It stores these files directly in EMS if possible; otherwise, it writes them to disk. A parameter, TempPath, is passed to the sort functions, DoSort and AutoSort, to specify the drive and directory for these files. Of course, disk space sufficient to hold the temporary files must be available on the specified drive.

To help determine the resources required to complete a sort, a procedure called AutoSortInfo is provided. AutoSortInfo is called automatically by AutoSort to determine that enough disk space exists before attempting the merge sort.

MSORT Terminology

Two terms that are used throughout the discussion of MSORT are "" and " order." A run is the number of items that fit into RAM at a time. The merge order is the maximum number of files that are used as input during the " phase." The "merge phase" is a process required only if the elements to be sorted don't fit into a single run.

Declarations

Constants

```
BiggestDataItem; = 65521;
```

The size of the largest single data item Turbo Pascal can handle.

```
MaxHeapToUse; : LongInt = 655210;
```

Specifies the maximum amount of heap space that MSORT can use.

```
MergeOrder = 5;
```

Specifies the number of files open during the merge phase and affects the performance of the sort. In the worst case, there will be MergeOrder files open for input, and one file open for output. The default value of 5 was carefully chosen to balance performance, memory usage, and file handle usage. You can reduce its value to as low as 2.

```
STemp; : String[5] = 'STEMP';
```

Used as the first five characters of each temporary file name created during the merge phase.

```
UseEMS; : Boolean = True;
```

Controls whether the merge sort system will attempt to use EMS if it is available and needed.

Types

```
MSortStatus; = (MSortSuccess,      {Successful sort}
                MSortOutOfMemory, {Insufficient memory}
                MSortDiskError,    {Disk I/O error}
                MSortOutOfDisk,    {Insufficient disk space for
merge}
                MSortEMSError,     {EMS error}
                MSortUserAbort);   {User abort}
```

Both AutoSort and DoSort return a result of type MSortStatus. The calling application should check this result to determine the outcome of the sort. MSortIOResult, described below, provides additional information in case of a disk I/O error.

```
PathName; = String[79];
```

String type used for pathnames.

Variables

```
GRunLength; : Word;
```

The run length is the number of items (records) that can fit in memory at one time. If all the items to be sorted fit into memory, then this will be equal to the total number of items. When using AutoSort, this number is calculated automatically based on available memory.

```
LastFileName; : PathName;
```

Contains the pathname of the last file written to by the merge sort system.

```
MSortIOResult; : Integer;
```

If the sort returns a status of MSortDiskError, then MSortIOResult contains the value of IOResult at the time of the error.

UsingEMS; : Boolean;

This variable is True if the merge sort system is using EMS memory. EMS memory is used only if the typed constant UseEMS is True, if there are enough EMS pages available for use (at least one run's worth), and if a merge phase is required.

AbortSort

Syntax

~~procedure AbortSort;~~

Purpose

Halt a sort prematurely.

Description

Any of the three user routines can call this routine to halt the sort prematurely. DoSort and AutoSort will exit at the next opportunity, returning MSortUserAbort in the function result.

AutoSort

Syntax

```
function AutoSort; (FSizeInRecs : LongInt; RecLength : Word; TempPath  
: PathName;  
GetElements : Pointer; LessFunc : Pointer;  
PutElements : Pointer) : MSortStatus;
```

Purpose

Sort a data set.

Description

AutoSort is the routine you'll probably call to perform a sort. It works at the highest possible level, optimizing the various merge sort parameters to take advantage of system resources. AutoSort is designed to be completely configurable at runtime. Within the limitations of available disk space, it can sort any number of elements of any type, calling user-defined functions for element comparison, input, and output. If possible, all the elements to be sorted are held in memory; otherwise, a merge sort is performed on the elements.

The first two parameters to AutoSort tell it how many elements to expect and the size of each element. AutoSort needs this information to accurately calculate various merge sort parameters. While FSizeInRecs need not be exact, it should be very close to the actual number of records sorted, or the sort might run out of resources. If the number of records to be sorted is not known in advance, use DoSort instead of AutoSort.

TempPath tells AutoSort where to put any temporary files created during the merge sort phase. Specifying the empty string, "", tells AutoSort to use the default drive and directory. Any other string must specify the name of an existing directory. There must be sufficient disk space on the specified drive to accommodate any temporary files created by the sort and merge phase, or AutoSort exits with status MSortOutOfDisk.

The remaining parameters to AutoSort are pointers to routines called during the sort. These routines must be global, must be compiled under the far model, and must have exactly the parameters described below.

GetElements points to a procedure like the following:

```
{ $F+ }  
procedure SendToSortEngine;  
begin  
  { For each element to be sorted do... }  
  if not PutElement(X) then  
    Exit;  
end;  
{ $F- }
```

The routine must obtain each element to be sorted and call PutElement with that element as a parameter. It will generally take the form of a for or while loop.

LessFunc points to a function that returns True if its first element is less than the second, False if the first is greater than or equal to the second. The parameters passed to the Less function are untyped so that they can be treated as variables of a specific type by using typecasting or variables declared absolute on top of the parameters. Actually, X and Y can be declared as parameters of the actual data type as long as they are var parameters.

```
{ $F+ }  
function Less(var X,Y) : Boolean;  
begin  
  { Return True if X < Y, False otherwise }  
end;  
{ $F- }
```

PutElements points to a procedure that retrieves the sorted elements from MSORT using GetElement and makes them available to the program.

AutoSort

```
{ $F+ }  
procedure GetFromSortEngine;  
begin  
    while GetElement(X) do  
        { Do something with X } ;  
end;  
{ $F- }
```

Using these example routines, the call to AutoSort might look as follows:

```
Status := AutoSort(NumItems, RecSize, '', @SendToSortEngine, @Less,  
                  @GetFromSortEngine);
```

AutoSort calls the SendToSortEngine routine after performing preliminary error checking and initialization tasks. SendToSortEngine in turn must call PutElement to put each element into MSORT's internal data structures. When AutoSort needs to compare two elements to place them in sorted order, it calls your Less function. After sorting is complete, AutoSort calls your GetFromSortEngine routine, which must call GetElement to get each element back in sorted order. When GetFromSortEngine returns, AutoSort deallocates the memory it used and returns a sort status to the original calling program. See the MSortStatus type for the codes AutoSort uses.

Example

See the introduction to this section.

See Also

AutoSortInfo

DoSort

AutoSortInfo / DoSort

Syntax

```
function AutoSortInfo; (FSizeInRecs : LongInt; RecLength : Word;
                      var HeapSpace : LongInt; var DiskSpace :
LongInt;
                      var FileHandles : Word; var EMSPages : Word;
                      var RunLen : Word; var FileBufs : Word;
                      var OutFileBufs : Word;
                      var AllInMem : Boolean) : MSortStatus;
```

Purpose

Compute the resources needed for a sort.

Description

AutoSortInfo can be called to determine the best location for temporary files, to help choose customized parameters for DoSort, or to predict whether the sort can succeed given the system's resources. AutoSortInfo will return MSortOutOfMemory if the sort simply cannot be performed.

AutoSortInfo calculates AutoSort's heap space overhead, then checks to see if the amount of free heap space will allow the entire sort to fit in RAM. If not, it chooses the largest possible run size that will fit, and simulates the merge sort process to calculate the disk space, EMS pages, and file handles required during the merge.

HeapSpace returns the peak bytes of heap space required. DiskSpace returns the peak bytes of disk space. FileHandles returns the highest number of file handles. EMSPages is the peak EMS page allocation (each page is 16384 bytes). RunLen is the number of records to be loaded into memory at one time. FileBufs is the number of bytes of heap space used for input file buffers during merging. OutFileBufs is the number of bytes of heap space used for the output buffer during merging. AllInMem is True if the sort can be performed entirely in memory, i.e., no merging is required.

Syntax

```
function DoSort; (RunLength : Word; RecLength : Word; InFileBufMax :
Word;
                OutFileBufMax : Word; TempPath : PathName;
GetElements : Pointer;
                LessFunc : Pointer; PutElements : Pointer) :
MSortStatus;
```

Purpose

Sort a data set with control over low-level configuration.

Description

AutoSort calls this function after it determines the optimum sort parameters. Many parameters to DoSort are the same as those passed to AutoSort, so they aren't described again here. The few that differ are described next.

RunLength is the number of elements to be sorted in RAM at one time. You must assure that at least RunLength*RecLength bytes of heap space are free. (Whether this space must be contiguous or not is determined by the conditional define BigHeap, described earlier.)

InFileBufMax and OutFileBufMax specify buffer sizes to be used while merging. Again, it is your responsibility to assure that sufficient heap space is free for DoSort to allocate these buffers. Passing zero for these parameters disables buffering altogether, at significant cost to performance.

In order for output buffering to make sense, OutFileBufMax should be at least twice RecLength (and hopefully many times more than that). The value specified by InFileBufMax is divided by the MergeOrder, and each input file gets a buffer of that size. Hence, for effective input buffering, InFileBufMax should equal at least twice RecLength*MergeOrder (and hopefully many times more than that).

AutoSortInfo / DoSort

See Also

~~AutoSort~~

GetElement / PutElement

Syntax

```
function GetElement(var X) : Boolean;
```

Purpose

Return the next element in sorted order.

Description

Your application will have the opportunity to call GetElement within the retrieval routine whose address you pass to AutoSort.

Because MSORT does not know the type of your data, GetElement's var parameter is untyped. MSORT knows how many bytes to pass back; you must make sure the variable you pass is large enough.

GetElement returns True until there are no more sorted elements to retrieve. Once it returns False, the parameter X is undefined.

Example

See the examples at the beginning of this section.

Syntax

```
function PutElement(var X) : Boolean;
```

Purpose

Submit an element to the sort system.

Description

Your application will have the opportunity to call PutElement within the loader routine whose address you pass to the AutoSort function.

Because MSORT does not know the type of your data, the parameter X is an untyped var parameter. As such, you cannot pass a constant or expression to PutElement. MSORT knows how many bytes to transfer from X based on the RecLength passed to AutoSort or DoSort.

PutElement returns True when an element is successfully added to the MSORT internal structure. If it returns False, an error occurred (probably due to insufficient memory or disk space), and the application must stop adding elements and exit the loader routine.

Example

See the examples at the beginning of this section.

Rather than tweak the MSORT unit to compile and work for a protected mode environment, MSORTP was written from scratch to take advantage of protected mode features. As a result, it can use the protected mode heap more efficiently than would be the case from a simple porting exercise. It can also take advantage of language features that are guaranteed to exist in protected mode applications, specifically, procedure variables, the Turbo Pascal built-in assembler, and PChar strings. MSORTP can be compiled for all targets, although when used in real mode it won't take advantage of EMS like the MSORT unit does.

You use MSORTP much the same as MSORT, with only a few differences. The following small program shows how to implement a text file sort filter using MSORTP.

```
{ $R-, S-, X+, I+ }
program TextSort;
  {-Protected mode text file sort filter}
uses
  Strings, MSortP;
var
  LineBuf : array[0..1023] of Char;
  MaxHeap : LongInt;
  MinHeap : LongInt;
  Status  : Word;

procedure SendToSortEngine; far;
var
  P : PChar;
begin
  while not Eof do begin
    {Read the next line from the input file as a PChar}
    ReadLn(LineBuf);
    {Don't sort empty lines to avoid StrNew quirk}
    if StrLen(LineBuf) <> 0 then begin
      {Copy string to heap}
      P := StrNew(LineBuf);

      {Store the pointer in the sort engine}
      if not PutElement(P) then
        Exit;
    end;
  end;
end;

function Less(var X, Y) : Boolean; far;
begin
  Less := (StrComp(PChar(X), PChar(Y)) < 0);
end;

procedure GetFromSortEngine; far;
var
  P : PChar;
begin
  {Return each pointer and write the associated string to output}
  while GetElement(P) do
    WriteLn(P);
end;
```

```

begin
  {Determine how much heap space MergeSort can use}
  MaxHeap := MemAvail div 10;
  MinHeap := 4*MinimumHeapToUse(SizeOf(PChar));
  if MaxHeap < MinHeap then
    MaxHeap := MinHeap;

  {Perform the sort}
  Status := MergeSort(MaxHeap, SizeOf(PChar),
    SendToSortEngine, Less, GetFromSortEngine,
    DefaultMergeName);

  if Status <> 0 then
    WriteLn('Sort failed, Status ', Status);
end.

```

This program provides three routines for the use of MSORTP. SendToSortEngine reads lines of text from the standard input device, allocates space for each line on the heap, and passes the string pointer to MSORTP by calling PutElement. Only the pointer is passed to MSORTP, since there is no need for MSORTP to keep a second copy of a string that is already stored in memory.

The Less function compares pairs of elements for MSORTP. This routine simply uses the StrComp function from Borland's STRINGS unit to perform a case-sensitive comparison of two PChars. Note the typecast from the untyped VAR parameters Less receives to the PChar data type stored in the sort engine.

Finally, GetFromSortEngine retrieves the elements in sorted order. In this program, GetElement returns a pointer to the actual string data. When the \$X+ compiler directive is enabled, as it is here, the WriteLn statement writes the string pointed to by each pointer P.

The main block of the program calls the MergeSort function to perform the sort. First it must decide how to partition the heap between the needs of string storage and the needs of MSORTP. MSORTP is given about 10% of the heap and the rest is reserved for text strings. Since the sort engine is storing 4 byte elements, this is a balanced split if the average string length is 40 bytes. Also note that the MinimumHeapToUse function is called to guarantee that the sort engine gets at least 4 times as much heap space as it requires at a minimum. In this way merging is minimized and the sort will always succeed unless there is insufficient disk space or insufficient heap space to hold the input strings.

The second parameter passed to MergeSort is the size of the elements to be sorted, in this case 4 bytes, the size of a PChar. The next three parameters are the routines used to pass elements to the sort engine, to compare elements, and to get sorted elements back from the engine. The final parameter is a routine used to name each merge file.

This program uses the default routine DefaultMergeName provided by MSORTP, which stores the merge files in the current directory.

MergeSort returns a Word result indicating the status of the sort. The result is zero to indicate success, or a non-zero result to explain the reason for a failure.

Non-Callback Sorting

The MSORTP unit also offers you the ability to do "non-callback" sorting. This means that you do not have to provide a SendToSortEngine or a GetFromSortEngine procedure; all sorting occurs "in-line." It is best understood by means of some example code:

```

{$X+}

...

InitMergeSort(HeapToUse, RecLen, LessFunc, DefaultMergeName);
if GetSortStatus = 0 then begin
  while (GetSortStatus = 0) and ...HaveSomeElements... do begin
    ...GetNextElement(Element)...
    PutElement(Element);
  end;
  while (GetSortStatus = 0) and GetElement(Element) do
    ...OutputSortedElement(Element)...
  DoneMergeSort;
end;
if GetSortStatus <> 0 then
  ...ReportError...

```

You must supply the implied routines (those identifiers surrounded by dots).

In non-callback sorting, you call the PutElement and GetElement functions inline in your code. While this doesn't provide any fundamental advantages over the callback method used by MergeSort (and the sort routines in MSORT), it may be more convenient and logical for you. The sorting actually occurs within the calls to PutElement and GetElement. If all elements fit into memory, sorting does not occur until the first call to GetElement. The sort engine calls your LessFunc as needed to compare pairs of elements.

Note the use of the GetSortStatus function to check the current status of the sort. You don't need to use this function when you call MergeSort, since the error detection is handled internally by MergeSort.

MSORTP Terminology

Two terms that are used throughout the discussion of MSORTP are "" and " order." A run is the number of items that fit into memory at a time. The merge order is the maximum number of files that are used as input during the " phase." The "merge phase" is a process required only if the elements to be sorted do not fit into a single run.

Declarations

Constants

```
MaxSelectors; = 256;
```

The maximum number of selectors that will be allocated by the MSORTP sort engine. The MergeSort function allocates memory buffers (using GlobalAlloc) using blocks of up to 65535 bytes, but the size is always an even power of two times the sort record length. As a result the default value of MaxSelectors enables MergeSort to access between 8MB and 16MB of memory. If 8MB is insufficient, you may want to increase MaxSelectors to 512. If 8MB is more than enough, consider decreasing MaxSelectors. Note that the data segment space that MSORTP uses is dominated by the static array that holds these selectors (this array is 2*MaxSelectors bytes in size).

For a real mode target (where selectors do not exist) this constant determines the number of individual allocations made on the heap.

```
MedianThreshold; = 16;
```

The partition length below which the in-memory quicksort simply uses the middle element of the partition for the pivot element. For partition lengths at least this size, MergeSort uses the median of the left, right, and middle elements for the pivot. The median of three algorithm protects the sort against degrading to N*N performance for nearly sorted lists.


```
MergeOrder = 5;
```

Specifies the number of files open during the merge phase and affects the performance of the sort. You can set MergeOrder to any value in the range from 2 to 10 inclusive. However, experimentation indicates that the default value of 5 is optimal under a wide range of conditions.

```
MinRecsPerRun; = 4;
```

Minimum number of records that must fit in memory during a sort. If fewer records fit in memory, MergeInfo and MergeSort return an error code. If even MinRecsPerRun records fit in memory, MergeSort performs merging to complete the sort. Increase this constant if you prefer that the sort fail instead of doing an excessive amount of merging.

```
SwapThreshold; = 64;
```

The record size below which MergeSort swaps complete data records. For records SwapThreshold bytes or larger, MergeSort swaps pointers to records instead of the records themselves. Swapping pointers is the faster approach for large records sorted in memory, but this approach has a memory overhead of 4 bytes per record plus a buffer segment that must be used for a run output buffer. The default of 64 was chosen to keep the typical overhead below 10%. Reducing the default also provides no significant improvement in performance.

Types

```
ElementCompareFunc; = function (var X, Y) : Boolean;
```

Specifies the type of the routine passed as the Less parameter to MergeSort. MergeSort calls this function to compare pairs of elements as needed. It must be declared FAR and must have the form shown here. It should return True if and only if element X is strictly less than element Y. You should typecast the untyped parameters to treat them as elements of the type you are sorting.

```
ElementIOProc; = procedure;
```

Specifies the type of the routine passed as the SendToSortEngine and GetFromSortEngine parameters to MergeSort. These routines must be declared FAR and must have no parameters.

```
MergeInfoRec; = record
  SortStatus      : Word;      {Predicted status of sort, assuming disk
ok}
  MergeFiles      : Word;      {Total number of merge files created}
  MergeHandles    : Word;      {Maximum file handles used}
  MergePhases     : Word;      {Number of merge phases}
  MaxDiskSpace    : LongInt;   {Maximum peak disk space used}
  HeapUsed        : LongInt;   {Heap space actually used}
  SelectorCount   : Word;      {Number of selectors allocated}
  RecsPerSel      : Word;      {Records stored in each selector}
end;
```

Describes the structure returned by the MergeInfo function. This function predicts the status of a sort and its resource usage given certain information about it. See MergeInfo for more information.

```
MergeNameFunc; = function (Dest : PChar; MergeNum : Word) : PChar;
```

Specifies the type of the routine passed as the MergeName parameter to MergeSort. MergeSort calls this function to obtain the name of each merge file when needed.

MSORTP interfaces a default MergeNameFunc that can be used in many circumstances. This routine, DefaultMergeName, returns a name of the form 'SORnnnnn.TMP', where nnnnn is the merge file number given by MergeNum.

A MergeNameFunc must return a unique file name for each value of MergeNum. Typically you specify a non-default MergeNameFunc if you want the merge files to be located on a different drive or directory than the current one. A non-default MergeNameFunc must also be specified if you are writing a network application where multiple users could be sorting at the same time. In that case, the MergeNameFunc must assure that each user has a unique set of merge files, either in separate directories or with unique names based on the user's name or connection number.

AbortSort / DefaultMergeName / DoneMergeSort

Syntax

```
procedure AbortSort;
```

Purpose

Halt a sort prematurely.

Description

Call this routine from your Less, SendToSortEngine, or GetFromSortEngine routines to abort a sort. If the Less function calls AbortSort, it must subsequently return False until the sort completes and MergeSort returns. If you call AbortSort, MergeSort returns a status of 1.

See Also

MergeSort

Syntax

```
function DefaultMergeName;(Dest : PChar; MergeNum : Word) : PChar;
```

Purpose

Return a default name for each merge file.

Description

The default merge name is SORnnnnn.TMP, where nnnnn corresponds to MergeNum. For example, for MergeNum = 1, DefaultMergeName returns 'SOR1.TMP'. For MergeNum = 999, DefaultMergeName returns 'SOR999.TMP'. For MergeNum = 65535 (the largest possible value), DefaultMergeName returns 'SOR65535.TMP'.

DefaultMergeName stores the null-terminated string in the buffer pointed to by Dest, and it returns Dest as the function result. MergeSort provides an 80 character buffer each time it calls the merge name function. This buffer must include the terminating null.

You can write your own merge name function that puts the merge files somewhere besides the current directory. Follow the example of DefaultMergeName and pass your function to MergeSort.

See Also

MergeSort

Syntax

```
procedure DoneMergeSort;;
```

Purpose

Dispose of memory and files allocated by InitMergeSort.

Description

Don't call this function unless you've called InitMergeSort to use the non-callback method of sorting. If InitMergeSort succeeded, you must call DoneMergeSort whether or not the sort itself succeeded.

DoneMergeSort does not change the value returned by GetSortStatus.

See Also

InitMergeSort

MergeSort

GetElement / GetSortStatus / InitMergeSort

Syntax

```
function GetElement(var X) : Boolean;
```

Purpose

Return the next element in sorted order.

Description

Call this routine repeatedly in your GetFromSortEngine routine to retrieve the sorted elements. GetElement returns True until there are no more sorted elements to retrieve. GetElement copies the next element into the variable you pass as the parameter X. Be sure that this variable is large enough to hold an entire record; otherwise GetElement will overwrite memory.

When GetElement returns False, the parameter X is not initialized.

See Also

PutElement

Syntax

```
function GetSortStatus; : Word;
```

Purpose

Return the current sort status value.

Description

This function simply returns the value of an internal status variable used by MSORTP. You don't need to call it unless you're using the non-callback method of sorting. Here is a list of the values that can be returned by GetSortStatus:

0	success
1	user abort (AbortSort was called)
8	insufficient memory to sort
106	invalid input parameter (RecLen zero, MaxHeapToUse too small)
204	invalid pointer returned by GlobalLock, or SelectorInc <> 8
213	no elements available to sort
214	more than 65535 merge files
else	DOS or Turbo Pascal I/O error code

See Also

InitMergeSort

MergeSort

Syntax

```
procedure InitMergeSort; (MaxHeapToUse : LongInt; RecLen : Word;  
                          Less : ElementCompareFunc; MergeName :  
                          MergeNameFunc);
```

Purpose

Initialize the merge sort data structures for non-callback sorting.

Description

You can use this routine together with DoneMergeSort as an alternative to MergeSort. Don't call it if you are calling MergeSort.

InitMergeSort starts a sorting process that is referred to as "non-callback" sorting. See "Non-Callback Sorting" earlier in this section.

The parameters passed to InitMergeSort have the same meanings as the parameters of the same name passed to MergeSort, and hence are not discussed here.

GetElement / GetSortStatus / InitMergeSort

See Also

~~DoneMergeSort~~
MergeSort

~~GetSortStatus~~

MergeInfo

Syntax

```
procedure MergeInfo; (MaxHeapToUse : LongInt; RecLen : Word;  
                     NumRecs : LongInt; var MI : MergeInfoRec);
```

Purpose

Predict the status and resource usage of a merge sort.

Description

MaxHeapToUse is the maximum number of bytes of heap space the sort should use. MergeInfo actually allocates heap space up to this amount; if there is less heap space available, the MergeInfo results apply only to the available heap space.

RecLen is the size in bytes of each record to be sorted. NumRecs is the total number of records to be sorted (or a close approximation).

MI returns information about the proposed sort. MI.SortStatus is zero if the sort is predicted to succeed. MergeInfo assumes that there is sufficient disk space and that no disk errors will occur.

MI.MergeFiles is the total number of merge files that will be created.

MI.MergeHandles is the total number of file handles used. This will always be in the range of 0 to MergeOrder+1 inclusive.

MI.MergePhases is the number of merge phases. A value of 0 indicates that the sort can be done completely in memory. 1 indicates that MergeOrder or fewer merge files are created and will be merged in one pass. Higher values mean that multiple merge passes are required, with the output from earlier passes feeding the input of later passes.

MI.MaxDiskSpace is the peak disk space required. Since merge files are deleted as soon as they are used, the disk space used in a merge sort grows and shrinks. All merge files are deleted when the sort is complete. MaxDiskSpace is always smaller than $2 * \text{RecLen} * \text{NumRecs}$. The analysis that MergeInfo performs to determine MaxDiskSpace requires that MI.MergeFiles be smaller than 16384, and that $4 * \text{MI.MergeFiles}$ bytes of heap space be free when MergeInfo is called. If these requirements aren't met, MergeInfo returns -1 for MI.MaxDiskSpace.

MI.HeapUsed is the number of bytes of heap space the sort will actually use. This is always less than or equal to MaxHeapToUse.

MI.SelectorCount is the number of selectors (and memory blocks) that the sort will allocate in protected or Windows mode; in real mode it is the number of blocks of heap space the sort will allocate.

MI.RecsPerSel is the number of records stored in each memory block. This is always a power of two.

See Also

MinimumHeapToUse

OptimumHeapToUse

MergeSort

Syntax

```
function MergeSort, (MaxHeapToUse : LongInt; RecLen : Word;  
                    SendToSortEngine : ElementIOProc; Less :  
                    ElementCompareFunc;  
                    GetFromSortEngine : ElementIOProc;  
                    MergeName : MergeNameFunc) : Word;
```

Purpose

Sort a set of elements.

Description

MaxHeapToUse specifies the maximum number of bytes of heap space the sort will use. It is not an error for MaxHeapToUse to exceed MemAvail; MergeSort will use whatever is available. If you know in advance how many records will be sorted, it is a good idea to pass the result returned by OptimumHeapToUse for this parameter.

RecLen is the number of bytes in each record to be sorted. You can sort actual records, pointers to records already loaded into memory, or indexes of records stored on disk.

SendToSortEngine is a procedure that you provide. It passes the sort elements to the sort engine. Your procedure must call PutElement for each element to be sorted. If PutElement returns False, an error occurred (usually out of disk space) and you should exit your SendToSortEngine procedure.

Less is another function that you provide. It compares pairs of elements. This function must return True if and only if element X (the first parameter) is strictly less than element Y (the second parameter).

GetFromSortEngine is also a procedure that you provide. It retrieves the sorted elements from the sort engine. Your procedure must call GetElement to retrieve each element in sorted order. When GetElement returns False, all elements have been retrieved or an error occurred.

MergeName is a function that provides a name for each merge file. You can often pass DefaultMergeName for this parameter. DefaultMergeName puts all of the merge files in the current directory.

MergeSort returns a status code in its function result. It can return the following values:

0	success
1	user abort (AbortSort was called)
8	insufficient memory to sort
106	invalid input parameter (RecLen zero, MaxHeapToUse too small)
204	invalid pointer returned by GlobalLock, or SelectorInc <> 8
213	no elements available to sort
214	more than 65535 merge files
else	DOS or Turbo Pascal I/O error code

The most common Turbo Pascal error code returned by MergeSort is 101, which means that there was insufficient disk space to store the merge files.

If you want to predict whether a sort can succeed before actually attempting it, use the MergeInfo procedure.

See Also

DefaultMergeName
MergeInfo

GetElement
PutElement

MinimumHeapToUse / OptimumHeapToUse / PutElement Syntax

```
function MinimumHeapToUse; (RecLen : Word) : LongInt;
```

Purpose

Return the minimum heap space that allows MergeSort to succeed.

Description

Given the size of each record (RecLen), MinimumHeapToUse returns the smallest amount of heap space that will allow a sort to succeed. You can pass this value to MergeSort to sort a group of elements using the smallest amount of memory. Note that the value returned by MinimumHeapToUse is often very small and can cause a significant amount of merging, so it's generally better to multiply the result by a reasonable factor (say 2-4) even if you want to minimize heap usage of a sort.

Because of a quirk in the BP7 DPMI heap manager, MinimumHeapToUse adds 2048 bytes to the actual computed heap requirement in a protected mode application. This is important because sometimes the DPMI heap manager consumes about 2000 bytes of heap space for its internal use.

See Also

OptimumHeapToUse

Syntax

```
function OptimumHeapToUse; (RecLen : Word; NumRecs : LongInt) :  
LongInt;
```

Purpose

Return the smallest heap space for a sort with no merging.

Description

Given the size of each record (RecLen) and the number of records to be sorted (NumRecs), OptimumHeapToUse returns the amount of heap space needed to perform the sort entirely in memory. Additional heap space does not help the sort. Less heap space causes merging.

Because of a quirk in the BP7 DPMI heap manager, OptimumHeapToUse adds 2048 bytes to the actual computed heap requirement in a protected mode application. This is important because sometimes the DPMI heap manager consumes about 2000 bytes of heap space for its internal use.

See Also

MinimumHeapToUse

Syntax

```
function PutElement (var X) : Boolean;
```

Purpose

Submit an element to the sort system.

Description

Call this function in your SendToSortEngine routine for each element to be sorted. Pass the element to be sorted in the untyped parameter X. PutElement returns True if the element is successfully processed by MergeSort. It returns False if an error occurred; do not continue to call PutElement in this case.

See Also

GetElement

B-Tree Filer includes a conversion unit that enables you to import data from a dBase file into a Filer-type fileblock, and to export data from a Filer fileblock into the dBase format.

The DBIMPEXP unit implements the routines that help convert dBase III or dBase IV databases to B-Tree Filer fileblocks, and vice versa. Support is provided for converting both dBase data and memo files. When converting from dBase format to B-Tree Filer format there is an option to generate a Pascal source code include file defining the record layout of the newly created Filer fileblock. The conversion routine also accepts a programmer-written filter function which is used to decide whether a given record will be transferred or not.

The routines in the DBIMPEXP unit convert data files only, support is not provided to convert index files in either direction.

The DBIMPEXP unit uses the CARRCONV unit to convert Pascal field types to dBase field types and vice versa. The routines of this unit are not documented here as they are only used internally by DBIMPEXP. For more information please browse the CARRCONV.PAS file.

Two example programs demonstrate how to use DBIMPEXP. DB2ISAM converts any dBase III or IV format data file to a B-Tree Filer file (generating a Pascal include file to describe the record format of the created B-Tree Filer file if required). ISAM2DB converts the ADDRESS.DAT file from the NETDEMO.EXE example program to the dBase format.

Programming with DBIMPEXP

A routine which converts a dBase database to a B-Tree Filer fileblock should (in general) take the following steps in the order shown. A fixed length record B-Tree Filer fileblock is created (variable length records are not supported). The example program DB2ISAM.PAS takes this approach.

1. Call BTInitIsam to start the B-Tree Filer environment.
2. Use CreateListHeaderUseDBaseFiles to open the dBase data file (and, if required, the dBase memo file), read the dBase record structure from the data file (dBase data files hold their record structure in the file header), and create a list of the field definitions as a linked list of node records, each node containing the definition of a single field.
3. Call CompleteDBaseList to determine the nearest Pascal data type that corresponds to each dBase field type in the field definition list. For example, for a dBase character field type the nearest Pascal data type equivalent is either a Char (if the dBase field width is 1) or a string (if greater than one). It also calculates the offsets of each of the resulting fields in the Filer record.

You now have a complete field description of the dBase record format, and also a complete, internally defined, B-Tree Filer record structure for the new fileblock.

4. Call DBaseImport to convert all the records in the dBase file(s) to the B-Tree Filer format, using the field definition list created in steps 2 and 3. The new Filer fileblock is created and opened before the conversion process begins and is closed afterwards.
DBaseImport accepts various user-defined routines to alter its behavior. You can provide a filter function to accept or reject input records for inclusion into the final Filer fileblock; provide a status function to display a progress report as the records are converted; and provide a routine to generate a Pascal include file that describes the record structure being created.
5. Use DBaseCloseFiles to close the dBase files.
6. Free the memory occupied by the field definition list by calling FreeListHeader.

7. Use `BTExitIsam` to close the B-Tree Filer environment.

Conversion of memo fields is treated slightly differently than 'normal' fields. You must specify the maximum number of characters to convert from the memo text (this will be the field size for all fields created from dBase memo fields in the Filer record). The memo text is always converted to a null-terminated string. The fields in the B-Tree Filer record are moved so that fields converted from dBase memo fields are grouped at the end of the record. This facilitates the creation of a new variable length record fileblock. DBIMPEXP does not support variable length records, but when the variable length fields are placed at the end of the record, you can then use `FixToVar` (see `_6.F`) to convert to variable length records.

The export process (B-Tree Filer fileblock to dBase file) follows a similar approach. However, because B-Tree Filer fileblocks do not have an built-in record definition, there is an extra step to define the Filer fields. The example program `ISAM2DB.PAS` uses this approach.

1. Call `BTInitIsam` to start the B-Tree Filer environment.
2. Use `CreateListHeaderOpenFileBlock` to open the B-Tree Filer fileblock and create an empty field definition list.
3. For each field of the fileblock record structure that you want to convert, call `AddFieldNode`. It defines the field's B-Tree Filer characteristics (type, length, offset) and dBase characteristics (name, type, width, decimal places). `AddFieldNode` creates a new field node record to hold this information and adds it to the field definition list created by step 2. Note that you must call `AddFieldNode` at least once (i.e. you must be converting at least one field per record).
4. Call `CompleteIsamList` to determine the nearest dBase field type that corresponds to each Pascal data type in the field definition list. It completes any field definitions required.
You now have a complete field description of every B-Tree Filer record, and also a complete, internally defined, record structure for the new dBase data file.
5. Call `DBaseExport` to convert all the records in the B-Tree Filer fileblock to the dBase file format, using the field definition list created in steps 2, 3, and 4. The new dBase data file is created and opened before the conversion process begins and is closed afterwards. A dBase memo file is also created and opened, if and only if there are memo fields defined within the field definition list.
`DBaseExport` accepts various user-defined routines to alter its behavior. You can provide a filter function to accept or reject input records for inclusion into the final dBase data file and provide a status function to display a progress report as the records are converted.
6. Use `ClosIsamFiles` to close the B-Tree Filer file.
7. Free the memory occupied by the field definition list by calling `FreeListHeader`.
8. Use `BTExitIsam` to close the B-Tree Filer environment.

Compiler directives

The `DBaseErrorMessage` function converts DBIMPEXP error codes into error message strings. To offer a broader support for different languages, you can specify which languages will be available at execution time by compiling with one (or both) of the following:

```
{ $DEFINE DBaseEnglishMessage }  
{ . $DEFINE DBaseGermanMessage }
```

The default is `DBaseEnglishMessage`. Remove the period before the `$DEFINE` for `DBaseGermanMessage` to enable support for German messages. The value of the typed constant

DBUseErrorMessage determines which of these languages is actually used when BBaseErrorMessage is called.

Declarations

Constants

```
DBDataExtension  : String [3] = 'DBF';
DBMemoExtension  : String [3] = 'DBT';
DumpExtension    : String [3] = 'DMP';
PasIncExtension  : String [3] = 'INC';
```

The standard file name extensions of dBase data files, dBase memo files, conversion log files (dump files), and Pascal include files.

```
DBUseErrorMessage : DBaseUsedErrorMessages =
{$IFDEF DBaseEnglishMessage}
    DBEnglish;
{$ELSE}
    {$IFDEF DBaseGermanMessage}
        DBGerman;
    {$ELSE}
        DBNoMsg;
    {$ENDIF}
{$ENDIF}
```

This typed constant determines the language in which error message strings are produced by DBBaseErrorMessage. If both the English and German languages are defined through the compiler directives DBaseEnglishMessage and DBaseGermanMessage, then the initial value of this constant is DBEnglish. If only one of the languages is so defined, then the initial value of DBUseErrorMessage is that language. If neither of the two symbols is defined, then DBUseErrorMessage is set to DBNoMsg. The setting of this typed constant can be changed at execution time. See DBBaseErrorMessage for more information.

```
DBVersion3X = $0300;
DBVersion4X = $0400;
```

The defined values for the DBaseVersion type (see the DBaseVersion type for more information).

```
DCWrite = 0;
DCSkip  = 1;
DCAbort = 2;
```

The defined values for the DecideCase type (see the DecideCase type for more information).

```
DEZERO  = 0;      {No error}
DEEOF   = 9011;   {End of file reached}
DEOOM   = 9012;   {Not enough free memory}
DEBV    = 9013;   {dBase version not supported or not a dBase file}
DECMF   = 9014;   {Format of the Memo file is invalid or the file is
corrupt}
DERSTL  = 9015;   {Record is too large}
DEWCT   = 9016;   {Invalid CType (data field type)}
DEEWTD  = 9017;   {Error when writing the type definition file}
DEECF   = 9018;   {Field could not be converted}
DELNHI  = 9019;   {ListHeader is not or is incorrectly initialized}
DETMF   = 9020;   {Too many fields}
DEWFT   = 9021;   {Wrong field type}
DEFWTL  = 9022;   {Field length too large}
DETMMD  = 9023;   {Too many decimal places}
DEFTVC  = 9024;   {Field type is not compatible with the dBase
version}
```

```

DEARFNA = 9025; {Auto-relational fields are not allowed at this
point}
DEFCNMF = 9026; {File contains no Memo fields}
DEEODF = 9027; {Error when opening the Dump file}
DEEWDF = 9028; {Error when writing the Dump file}
DEECDF = 9029; {Error when closing the Dump file}
DEPE = 9030; {Programming error}
DEFNAE = 9031; {Field name already exists}
DENFD = 9032; {No Field defined}
DELAST = 9033; {Last error constant, no error}

```

The defined values for the DBaseErrorNr type (see the DBaseErrorNr type for more information).

```

ERAbort = 0;
ERIgnore = 1;

```

The defined values for the ErrorReaction type (see the ErrorReaction type for more information).

```

ProcErrorHandler; : VoidFct_ErrorHandler;

```

Points to a routine that is called if an error occurs during a conversion process. By default this global variable is set to the NoErrorHandler procedure which is a do-nothing routine. You can write another error handling procedure (of type VoidFct_ErrorHandler) and point ProcErrorHandler to this new routine to handle any conversion errors that may occur.

```

ReservedCType = 0;
BooleanCType = 1;
CharCType = 2;
ByteCType = 3;
ShortIntCType = 4;
IntegerCType = 5;
WordCType = 6;
LongIntCType = 7;
CompCType = 8;
RealCType = 9;
SingleCType = 10;
DoubleCType = 11;
ExtendedCType = 12;
StringCType = 13;
ArrayCType = 14;
AZStringCType = 15;
DateCType = 16;
TimeCType = 17;

```

The data field types that the conversion routines recognize. These constants are explicitly used for specifying the types of B-Tree Filer fields with the AddFieldNode function when preparing for exporting data to a dBase file, and are used internally for the opposite conversion.

ReservedCType defines the first 4 bytes of the B-Tree Filer record (a LongInt). This field is usually reserved by B-Tree Filer fileblocks to determine whether a record is deleted (the value is non-zero) or not (the value is zero).

BooleanCType through ExtendedCType are used to convert between Pascal data types and dBase data types.

StringCType is the Pascal string type (a length-byte string).

ArrayCType is not used at this time.

AZStringCType is the C string type (a null-terminated string).

DateCType represents a date field. dBase date fields are converted to Pascal LongInt fields with a Julian date value (i.e. a number of days from the base date) between 0 representing January 1, 1600 and \$D6025 representing December 31, 3999.

TimeCType represents a time field. Pascal LongInt fields containing the elapsed time since midnight in seconds are converted to dBase character fields (8 spaces in size) containing the time as a string. Times range from 0 (00:00:00) to 86399 (23:59:59).

```
StartAutoRel; : LongInt = 0;
```

The start value for an auto-incrementing field.

```
WSInit = 0;
WSWork = 1;
WSExit = 2;
```

The defined values for the WorkStatus type (see the description of the WorkStatus type for more information).

Types

```
DBaseErrorNr; = Integer;
```

A variable of this type is passed to the current error handler. Its value defines the error that just occurred. The codes are the constants that are prefixed 'DE', and were chosen so that they do not clash with the set of error codes that IsamError can take.

```
DBaseFileName; = IsamFileName;
```

A string defining the drive, path, and name (without extension) of a set of files to be used by the DBIMPEXP unit. The various extensions defined above are suffixed to variables of this type to get the full path names. All file names passed to the DBaseImport, DBaseExport, CreateListHeaderUseDBaseFiles, and CreateListHeaderOpenFileBlock functions must be of this type.

```
DBaseUsedErrorMessages; = (DBNoMsg, DBGerman, DBEnglish);
```

Defines the different values that the typed constant DBUseErrorMessage can have.

```
DBaseVersion; = Integer;
```

A parameter of this type is passed to CompleteIsamList, and defines whether DBaseExport is to create a dBase III or a dBase IV compatible file. The two values that can be used are DBVersion3X and DBVersion4X.

```
DecideCase; = Integer;
```

This type is returned by a record filter function which is of type EnumFct_DecideWrite. The intended values for a DecideCase variable are the constants prefixed with 'DC'. DCWrite means accept the record; DCSkip means reject the record; DCAbort means cancel the entire conversion.

```
EnumFct_DecideWrite; = function(LHPtr : PListHeader; Errors :
Integer;
                                var BTBuf, DBBuf) : DecideCase;
```

A function of this type is passed as a parameter to the DBaseImport and DBaseExport functions. It is called after converting every input record, and prior to writing the output record. The function must decide whether the converted record is to be written, and must return DCWrite, DCSkip, or DCAbort. LHPtr defines the field definition list. Errors is the total number of fields in this record that could not be converted. BTBuf contains the B-Tree Filer record and DBBuf contains the dBase record.

```
ErrorReaction; = Integer;
```

A variable of this type is passed to the current error handler (whose address is in ProcErrorHandler). The variable indicates the severity of the error. ERAbort is a severe error, the program cannot recover and is about to be aborted. ERIgnore is non-fatal, the program can continue and ignore the error (apart from reporting it to the error handler).

```
ListHeader; = Record
    DBSource      : Boolean;
    ListPtr       : PFieldNode;
    DBHeaderPtr   : PDBaseHeader;
    BTHHeaderPtr  : PIsamHeader;
```

```
End;
PListHeader = ^ListHeader;
```

ListHeader is the central type for the dBase import and export routines. It holds the complete set of information about the conversion to be done; most importantly it holds the field definition list. A variable of type PListHeader is allocated and returned by the CreateListHeaderUseDBaseFiles and CreateListHeaderOpenFileBlock functions.

DBSource indicates whether the ListHeader was created for conversion from a dBase file (True) or to a dBase file (False). ListPtr points to a simple linked list that contains the field definition information. Just prior to the conversion, each node of this list contains full information about a field, both in terms of the dBase and of the B-Tree Filer implementations. The list is initially set up in part by CreateListHeaderUseDBaseFiles and/or CreateListHeaderOpenFileBlock. For a conversion to a dBase file, AddFieldNode adds nodes that define the Filer fields to be converted. The list is checked and completed by CompleteDBaseList and/or CompleteIsamList.

DBHeaderPtr and BTHHeaderPtr contain information about the dBase and B-Tree Filer files.

```
IntFct_ReXUser; = function(Status : WorkStatus; LHPtr :
PListHeader;

                                ReadRecs, WriteRecs, ErrorRecs : Longint;
                                var DatSBuf) : Integer;
```

A function of this type is passed as a parameter to the DBaseImport and DBaseExport functions. It is used to allow the display of status information to the user. It is called every time through the read, convert, and write loop of the conversion routine, just after the output record is written (or would have been written).

If Status is WSInit, the conversion process is about to start. ReadRecs contains the total number of records in the source file, and WriteRecs holds the record length of the source file. ErrorRecs is zero, and DatSBuf is undefined.

If Status is WSWork or WSExit, the conversion process just completed reading a record, converting it, and (possibly) writing it to the output file. ReadRecs contains the number of records read and WriteRecs contains the number of records written. ErrorRecs is the total number of records to date that had errors during the conversion. DatSBuf contains the newly formed record.

WSExit is called only once, just for the last time through the loop after the last record was converted and written. It allows the program to clean up the screen by removing the status display.

The function should return zero.

```
IntFct_WriteTDef; = function(LHPtr : PListHeader;
                                IFName : IsamFileName) : Integer;
```

A parameter of this type is passed to DBaseImport. Its function is to write an include file that defines the B-Tree Filer record format for the dBase file that is to be converted. LHPtr holds the field definition list and IFName is the name of the fileblock that is about to be created. Two predefined functions of this type are provided: WriteNoTypeDef (which does nothing at all), and WritePascalTypeDef (which creates a simple include file for Pascal programs). If you write your own function, use the source for WritePascalTypeDef as a template.

```
VoidFct_CharArrConvert; = procedure(CArrPtr : Pointer; Len : Word);
```

A parameter of this type (ProcCArrConv) is passed to the DBaseImport and DBaseExport functions. It allows an array of characters to be further modified before writing to the output file. The modifications must not alter the length of the array, and generally are limited to modifying the case of the characters, or translating them between character sets. Len is the number of characters, and CArrPtr is a pointer to the character string (which can be assumed to be an array [1..Len] of Char).

```
VoidFct_ErrorHandler; = procedure(Reaction : ErrorReaction;
IsamError : Integer;

                                DBaseError : DBaseErrorNr);
```

The global variable ProcErrorHandler is a variable of this type. You can write a procedure of this type and install it in ProcErrorHandler, and it is called whenever the conversion process gets an error.

Reaction indicates the severity of the error. ERAbort indicates that the program is about to be terminated. ERIgnore indicates that the error should be ignored. IsamError is non-zero if the error occurred in a B-Tree Filer routine, DBaseError is non-zero for any other type of error.

```
WorkStatus; = Integer;
```

A variable of this type is passed to a status display function, which is of type IntFct_ReXUser. The intended values for a WorkStatus variable are the constants prefixed with 'WS'. WSInit means that the conversion process is about to start, it allows the status display routine to build its screen. WSExit means that the conversion process has completed, the status routine can clear its screen. WSWork is passed at all other times, and indicates that a record was just converted.

AddFieldName

Syntax

```
function AddFieldName; (var LHPtr : PListHeader; Name :
DBaseFieldNameStr;
                        CType, BufSize, Offset : Word;
                        Width, Decimals : Integer) : Boolean;
```

Purpose

Create a new field descriptor and add it as a node to the field list.

Description

This function is used during the preparation stage for a conversion from a B-Tree Filer fileblock to a dBase database. It allocates a memory block for a field node, fills it with the passed information and appends it to the end of the field definition list LHPtr^.ListPtr.

As far as B-Tree Filer is concerned, the records it manages are typeless: they have size but no structure. A program which uses Filer imposes its own structure on the records by defining them as variables of some record type. Because of this, the DBaseExport function cannot deduce the structure of your Filer records, you must inform it about the fields that make up a record. You do this by calling AddFieldName once for every field that you want to convert to dBase format.

Name is the dBase field name; this is automatically converted into upper-case letters. CType indicates the data type of the B-Tree Filer field. BufSize contains the size of the field in bytes. Offset indicates the start position of the field from the beginning of the record (the first field has offset zero).

The next two parameters are to further define the resulting dBase field. Width determines its width and Decimals is the number of decimal places (specified if and only if the field is numeric). You may pass the value -1 for either of these parameters, and the function CompletelsamList will calculate the actual values from the values passed originally for CType and BufSize.

You do not have to call AddFieldName for every field in the Filer record structure, nor do you have to add the nodes in offset sequence. The sequence of calls to AddFieldName just gives the order of the fields in the final dBase record.

AddFieldName does check to see whether a field to be converted falls entirely within the Filer record.

The only field that B-Tree Filer takes for granted in a Filer record is the first four bytes of the record. They are used as a deletion marker (non-zero if the record is deleted, zero if the record is active). If you want to, you can convert these 'deleted' records as well by calling AddFieldName with CType equal to ReservedCType, BufSize equal to 4 (the size of a LongInt), and Offset equal to 0. If DBaseExport finds such a field node in the field definition list, it converts the 'deletedness' of a Filer record. If the first four bytes are zero, the converted dBase record is marked active, otherwise it is marked as deleted. If such a field node is not added, all records are copied and marked active.

If AddFieldName encounters an error, all the open files are closed, the memory allocated for LHPtr and its nodes is freed, and the function returns False. If the function succeeds, it returns True.

Example

```
var
  LHPtr : PListHeader;
begin
  ...
  LHPtr := CreateListHeaderOpenFileBlock('TEST');
  if (LHPtr = Nil) then
    { error processing }
  if not AddFieldName(LHPtr, DelMarkName, ReservedCType, 4, 0, -1,
-1 ) then
    { error processing }
  if not AddFieldName(LHPtr, 'NAME', StringCType, 16, 20, -1, -1 )
```

AddFieldNode

```
then  
    { error processing }  
...  
...
```

Prepares a new field list for a file block called TEST by calling CreateListHeaderOpenFileBlock, and then starts adding field definitions to it. Two fields are shown being added. The first is a deletion marker. It has the predefined name DelMarkName, is of type ReservedCType, is 4 bytes long (the size of a LongInt), and starts at offset 0 in the record. The second field is a string: the dBASE field is called "NAME", is of type StringType, is 16 bytes long, and starts at offset 20 in the B-Tree Filer record. In both cases the Width and Decimals parameters to AddFieldNode are passed -1, a call to CompleteSamList will calculate and fill these in later.

See Also

CompleteSamList

CloseDBaseFiles / CloseIsamFiles

Syntax

```
function CloseDBaseFiles;(LHPtr : PListHeader) : Integer;
```

Purpose

Close the dBase file(s).

Description

This function must be called after DBaseImport to close the dBase data and memo files opened by CreateListHeaderUseDBaseFiles.

If CloseDBaseFiles succeeds, it returns zero, otherwise it returns an error code (see the DEXxx constants earlier in this section).

Example

```
var
  LHPtr : PListHeader;
begin
  ...
  ErrCode := DBaseImport(LHPtr, ...);
  if (ErrCode <> 0) then
    { error processing }
  ErrCode := CloseDBaseFiles(LHPtr);
  if (ErrCode <> 0) then
    { error processing }
  FreeListHeader(LHPtr);
  ...
```

DBaseImport is called to convert a dBASE data file to B-Tree Filer format. This routine automatically closes the new fileblock. The call to CloseDBaseFiles closes the dBASE files. The call to FreeListHeader deallocates the heap memory maintained in LHPtr.

Syntax

```
function CloseIsamFiles;(LHPtr : PListHeader) : Integer;
```

Purpose

Close the B-Tree Filer file.

Description

This function must be called after DBaseExport to close the B-Tree Filer source fileblock opened by CreateListHeaderOpenFileBlock.

If CloseIsamFiles succeeds, it returns zero, otherwise it returns an error code (see the DEXxx constants earlier in this section).

Example

```
var
  LHPtr : PListHeader;
begin
  ...
  ErrCode := DBaseExport(LHPtr, ...);
  if (ErrCode <> 0) then
    { error processing }
  ErrCode := CloseIsamFiles(LHPtr);
  if (ErrCode <> 0) then
    { error processing }
  FreeListHeader(LHPtr);
  ...
```

CloseDBaseFiles / CloseIsamFiles

DBaseExport is called to convert a B-Tree Filer fileblock to a dBASE data file. This routine automatically closes the new dBASE data file. The call to CloseIsamFiles closes the fileblock.

The call to FreeListHeader deallocates the heap memory maintained in LHPtr.

CompleteDBaseList

Syntax

```
function CompleteDBaseList; (LHPtr : PListHeader;  
                           AZStrs, AutoRel : Boolean) : Integer;
```

Purpose

Complete the partially set up field definition list to prepare for DBaseImport.

Description

This function is used as the final preparatory step towards the actual conversion of a dBase database to a B-Tree Filer fileblock. It takes a field definition list LHPtr, created by CreateListHeaderUseDBaseFiles and completes the definition of each field node by determining the Filer data type for each dBase field type.

If AZStrs is True, dBase character arrays (of length greater than 1) are converted to null-terminated strings (C style). If it is False, they are converted to length-byte strings (Pascal style).

If AutoRel is True, an extra LongInt field node is added after the deletion field node (the first node). During the conversion this field acts as an auto-incrementing field, the first record converted and written will have the value given by the global variable StartAutoRel, the next record will have the next value (StartAutoRel+1), and so on.

If CompleteDBaseList succeeds, it returns zero, otherwise it returns an error code (see the DEXxx constants earlier in this section).

Example

```
var  
  LHPtr : PListHeader;  
begin  
  ...  
  ErrCode := CreateListHeaderUseDBaseFiles(LHPtr, 'TEST', 0);  
  if (ErrCode <> 0) then  
    { error processing }  
  ErrCode := CompleteDBaseList(LHPtr, False, False);  
  if (ErrCode <> 0) then  
    { error processing }  
  ErrCode := DBaseImport(LHPtr, ...);  
  if (ErrCode <> 0) then  
    { error processing }
```

First a new dBASE field list is created for the dBASE file TEST.DBF. Because dBASE data files have an internal record description, there is no need to add field nodes explicitly, CreateListHeaderUseDBaseFiles can do it automatically. Then a call to CompleteDBaseList completes the field list by calculating all the B-Tree Filer record offsets and widths. Finally, a call to DBaseImport converts the dBASE record to B-Tree Filer format.

CompleteIsamList

Syntax

```
function CompleteIsamList(LHPtr : PListHeader; DBVer : DBaseVersion) : Integer;
```

Purpose

Complete the partially set up field definition list to prepare for DBaseExport.

Description

This function is used as the final preparatory step towards the actual conversion of a B-Tree Filer fileblock to a dBase database. It takes a field definition list LHPtr, created by CreateListHeaderOpenFileBlock and added to by several calls to AddFieldNode, and completes the definition of each field node by determining the dBase field type for each Filer data type.

The DBVer parameter sets an internal flag that determines whether a dBase III or dBase IV compatible data file is created by DBaseExport.

If CompleteIsamList succeeds, it returns zero, otherwise it returns an error code (see the DEXxx constants earlier in this section).

Example

```
var
  LHPtr : PListHeader;
begin
  ...
  LHPtr := CreateListHeaderOpenFileBlock('TEST');
  if (LHPtr = nil) then
    { error processing }
  ..several calls to AddFieldNode..
  ErrCode := CompleteIsamList(LHPtr, DBVersion4X);
  if (ErrCode <> 0) then
    { error processing }
  ErrCode := DBaseExport(LHPtr, ...);
  if (ErrCode <> 0) then
    { error processing }
```

First a new field list is created for the fileblock TEST. AddFieldNode is called for each B-Tree Filer field that needs to be converted. Then a call to CompleteIsamList completes the field list by calculating all the dBASE field offsets, widths, and types. DBVersion4X is passed as the dBase version so that a dBase IV format file is created. Finally, a call to DBaseExport converts the B-Tree Filer data file to dBase format.

CreateListHeaderOpenFileBlock

Syntax

```
function CreateListHeaderOpenFileBlock; (BTFileName :  
IsamFileBlockName)  
: PListHeader;
```

Purpose

Open the B-Tree Filer fileblock and create an empty field definition list.

Description

This function is used in the preparation for a conversion of a Filer fileblock to a dBase file (or files). It opens the fileblock (data and index files) whose name (without extension) is given by BTFileName. It allocates memory for and builds an empty field definition list, and the address of the list is returned as the function result.

After calling this function, you must call AddFieldNode at least once (the field list must define at least one field to be converted) before the list can be 'completed' by calling CompleteIsamList, and then used by DBaseExport.

If an error occurs, CreateListHeaderOpenFileBlock returns nil and the fileblock remains closed.

Example

```
var  
  LHPtr : PListHeader;  
begin  
  ...  
  LHPtr := CreateListHeaderOpenFileBlock('TEST');  
  if (LHPtr = nil) then  
    { error processing }  
  ...  
  ..several calls to AddFieldNode..
```

The call to CreateListHeaderOpenFileBlock opens the fileblock called TEST, and allocates a new field definition list for it. The fields in the fileblock are then described by making several calls to AddFieldNode.

CreateListHeaderUseDBaseFiles

Syntax

```
function CreateListHeaderUseDBaseFiles; (var LHPtr : PListHeader;  
DBFileName : DBaseFileName;  
MaxMemoSize : Word) : Integer;
```

Purpose

Open the dBase file(s) and create the partial record definition list.

Description

This function is used in preparation for a conversion of a dBase data file (and possible memo file) to a B-Tree Filer fileblock.

DBFileName is the complete path of the dBase file(s) without any extension (the constants DBDataExtension and DBMemoExtension are suffixed to produce the full path names of the data and memo files respectively). This routine then opens the dBase file(s) ready for reading the database structure and for the data conversion.

MaxMemoSize indicates the maximum number of bytes that will be converted from any memo fields found. The number of memo fields which can be converted is not limited, however the final size of the converted Filer record must be less than 64KB. If 0 is passed in the MaxMemoSize parameter, the dBase memo file is not opened, and no memo fields are converted.

CreateListHeaderUseDBaseFiles reads the dBase database format from the dBase data file and builds a field definition linked list from the information. Each node in the linked list is the definition of a single field. The list is allocated on the heap, and its address is returned in LHPtr. This field list is reorganized internally by this function so that all memo field nodes appear at the end of the list. The effect is to force memo fields to be written at the end of the Filer record, thus simplifying the task of creating Filer variable length records later.

The field list always starts with a deletion field node because dBase records have a hidden delete marker as the first field.

Before using the list pointed to by LHPtr in the DBaseImport routine, it must be 'completed' by the CompleteDBaseList function.

If CreateListHeaderUseDBaseFiles succeeds it returns zero as a function result, otherwise it returns a non-zero value and the dBase database is closed.

Example

```
var  
  LHPtr : PListHeader;  
begin  
  ...  
  ErrCode := CreateListHeaderUseDBaseFiles(LHPtr, 'TEST', 0);  
  if (ErrCode <> 0) then  
    { error processing }  
  ErrCode := CompleteDBaseList(LHPtr, False, False);  
  if (ErrCode <> 0) then  
    { error processing }  
  ...
```

The call to CreateListHeaderUseDBaseFiles opens the dBASE file called TEST.DBF, reads the record definition embedded in it, and allocates a new field definition list for the field definitions. The last parameter of 0 means that no dBASE memo fields are to be converted. Once the list is prepared, a call to CompleteDBaseList completes it by calculating all the resulting B-Tree Filer field types, widths and offsets.

DBaseErrorMessage

Syntax

```
function DBaseErrorMessage; (ErrorNr : DBaseErrorNr) : String; ————
```

Purpose

Return a message string for the specified error code.

Description

This function returns a string describing the error code ErrorNr. It can return the string in different languages. The language used depends on the value of the typed constant DBUseErrorMessage.

DBaseExport

Syntax

```
function DBaseExport (LHPtr : PListHeader; DBFName : DBaseFileName;  
KeyNr : Word; FuncReXUser : IntFct_ReXUser;  
ProcCarrConv : VoidFct_CharArrConvert;  
FuncDecideWrite : EnumFct_DecideWrite) :  
  
Integer;
```

Purpose

Convert a B-Tree Filer data file to a dBase database.

Description

The DBaseExport function does the actual conversion of the B-Tree Filer data file to the dBase data and memo files. A dBase database with the name given by DBFName is created. If these files already exist, they are overwritten.

The conversion uses the information stored in the field definition list LHPtr^.ListPtr. This linked list contains a node for each field to be converted, and each node record contains the data required for the conversion of that field. The field list is created by CreateListHeaderOpenFileBlock, fields are added by AddFieldNode, the list is completed by CompletelsamList, and the memory for the list is deallocated by FreeListHeader after DBaseImport finishes.

If the field list contains one or more nodes that define a memo field (i.e., a null-terminated string whose length is greater than 255 bytes) then a dBase memo file is created in addition to the data file. If there are no such fields, a memo file is not created--all of the converted data ends up in the dBase data file.

KeyNr defines the index to use for reading the records from the B-Tree Filer fileblock. By using an index, you can create a sorted dBase data file (note, do not take 'sorted' to mean 'indexed'). If you want to use a purely sequential order and are not concerned with the final order of the dBase records, pass 0 as the KeyNr parameter.

FuncReXUser defines a routine that is called by the conversion process for every record read. The function of this routine is to display some kind of status or progress information to the user. If such a routine is not required, pass the predefined routine NoReXUser. The routine you write should return zero. If, however, it returns a non-zero result, the conversion process is aborted and DBaseExport terminates immediately.

ProcCarrConv defines a routine that is called after each conversion of a string field (both length-byte strings and null-terminated strings) or a character field. This allows you to do some further manipulation of the converted string, for example forcing upper-case, or translating from ANSI to OEM characters. If no further conversion is required for these character type fields, use the predefined function NoCarrConv.

FuncDecideWrite defines a routine that is called after each record is converted, and must decide whether the new record is written to the dBase file(s) or not. There is one predefined example function, StdDecideWrite, that skips all records that are marked as deleted (i.e., that have a non-zero value in the first four bytes of the Filer record).

Currently, the DBIMPEXP unit does not support variable length records. However, if you can guarantee that all of the data you convert is contained within the *first* section of your variable length Filer record, you can convert the data with DBaseExport. The reason this works is that the second and subsequent sections of a variable length record have their first byte set to \$01, and hence StdDecideWrite would view them as being deleted and not write them to the dBase file.

Another way of converting a variable length record file is to read it via an index. In this situation, the only records read are the initial sections of each variable length record.

If DBaseExport succeeds, it returns zero, otherwise it returns an error code (see the DEXxx constants earlier in this section).

If errors occur during the conversion, the numbers of the records and the field names of the fields that were not converted are written in a file. This file is given the same name as the source file, but with the extension DumpExtension.

DBaseExport

The following table shows the defaults for converting Pascal data types to dBase field types.

B-Tree Filer	DBIMPEXP	dBase		
Pascal data type	CType	Field type	Length (n)	
Decimals				
LongInt 0	DateCType	'D'	8	
LongInt 0	TimeCType	'C'	8	
String[n] 0	StringCType	'C'	<= 255	
Array[0..n] of Char 0	AZStringCType	'C'	<= 255	
0		'M'	> 255	
Char 0	CharCType	'C'	1	
Boolean 0	BooleanCType	'L'	1	
		(note 1)		
Byte 0	ByteCType	'N'	3	
ShortInt 0	ShortIntCType	'F'	4	
		(note 2)		
Word 0	WordCType		5	
Integer 0	IntegerCType		6	
LongInt 0	LongIntCType		11	
Comp 4	CompCType	'N'	14	
Real 4	RealCType	'F'	14	
		(note 2)		

DBaseExport

Single 4	SingleCType		14	
Double 4	DoubleCType		14	
Extended 4	ExtendedCType		14	

Notes:

1. The Boolean value True is converted to 'T', False to 'F'.
2. The 'F' data type is only valid in dBase IV.

Example

```
var
  LHPtr : PListHeader;
begin
  ...
  LHPtr := CreateListHeaderOpenFileBlock('TEST');
  ..several calls to AddFieldNode..
  ErrCode := CompleteIsamList(LHPtr, DBVersion4X);
  ...
  ErrCode := DBaseExport(LHPtr, 'DBTEST', 1, UserInfo,
                        NoCArrConv, StdDecideWrite );
  if (ErrCode <> 0) then
    { error processing }
```

Creates a new field list for the fileblock TEST by calling CreateListHeaderOpenFileBlock. Several calls to AddFieldNode define the B-Tree Filer fields to be converted, and then the field list is completed by a call to CompleteIsamList (the second parameter specifies that a dBASE IV file will be created). Finally there is a call to DBaseExport passing the complete field list LHPtr.

The new dBASE file will be called 'DBTEST.DBF' (the extension is added automatically) and the sequence of records in that data file will be the same as the TEST fileblock's index number 1. UserInfo is a progress display routine (the code for that routine is in the demonstration program ISAM2DB.PAS). NoCArrConv is a routine that does no extra character conversions. StdDecideWrite is a routine that filters out all B-Tree Filer deleted records (all those whose first 4 bytes are non-zero). After DBaseExport completes successfully, the dBASE data file DBTEST.DBF will have been created.

DBaseImport

Syntax

```
function DBaseImport; (LHPtr : PListHeader; IFBName :  
IsamFileBlockName;  
  
FuncWriteTypeDef : IntFct_WriteTDef;  
FuncReXUser : IntFct_ReXUser;  
ProcCArrConv : VoidFct_CharArrConvert;  
FuncDecideWrite : EnumFct_DecideWrite) :  
  
Integer;
```

Purpose

Convert a dBase database to a B-Tree Filer data file.

Description

The DBaseImport function does the actual conversion of dBase data and memo fields to the B-Tree Filer format. A fileblock with the name given by IFBName is created. If one already exists, it is overwritten.

The conversion uses the information stored in the field definition list LHPtr^.ListPtr. This linked list contains a node for each field to be converted, and each node record contains the data required for the conversion of that field. The field list is created by the function CreateListHeaderUseDBaseFiles, it is completed by CompleteDBaseList, and the memory for the list is deallocated by FreeListHeader after DBaseImport finishes.

FuncWriteTypeDef must point to a function that creates some kind of record type definition file (a Pascal 'include' file) corresponding to the format of the records converted by DBaseImport. Two example functions are provided: WriteNoTypeDef and WritePascalTypeDef. The former is a do-nothing function and can be used if no include file needs to be generated. The latter creates a Pascal record type definition as an include file. If you need another type of record definition file, use WritePascalTypeDef as a template to write a unique function.

FuncReXUser defines a routine that is called by the conversion process for every record read. The function of this routine is to display some kind of status or progress information to the user. If such a routine is not required, pass the predefined routine NoReXUser. The routine you write should return zero. If, however, it returns a non-zero result, the conversion process is aborted and DBaseImport terminates immediately.

ProcCArrConv defines a routine that is called after each string field or character field is converted. This allows you to do some further manipulation of the converted string, for example forcing upper-case, or translating from ANSI to OEM characters. If no further conversion is required for character type fields, use the predefined function NoCArrConv.

FuncDecideWrite defines a routine that is called after each record is converted, and must decide whether the new record is written to the Filer fileblock or not. There is one predefined example function, StdDecideWrite, that skips all records that were marked as deleted in the dBase file.

If DBaseImport succeeds, it returns zero, otherwise it returns an error code (see the DEXxx constants earlier in this section).

If errors occur during the conversion, the numbers of the records and the field names of the fields that were not converted are written in a file. This file is given the same name as the source file, but with the extension DumpExtension.

The following table shows the defaults for converting dBase field types to Pascal types.

DBaseImport

dBase			DBIMPEXP	B-Tree Filer
Field type	Length (n)	Decimals	CType	Pascal data type
'C' Of Char	= 1	0	CharCType	Char
	> 1	0	AZStringCType	Array [0..n]
			or StringCType (note 1)	String [n]
'D' (note 2)	8	0	DateCType	LongInt
'L'	1	0	BooleanCType	Boolean
'N' 'F' (note 3)	< 3	0	ShortIntCType	ShortInt
	< 5	0	IntegerCType	Integer
	< 10	0	LongIntCType	LongInt
	< 12	0	RealCType	Real
	< 16	0	DoubleCType	Double
	>= 16	0	ExtendedCType	Extended
	< 8	> 0	SingleCType	Single
	< 12	> 0	RealCType	Real
	< 16	> 0	DoubleCType	Double
	>= 16	> 0	ExtendedCType	Extended
'M' Of Char	<= 65000	0	AZStringCType	Array [0..n]

Notes:

1. The actual type depends on the parameter AZStrs in the function CompleteDBaseList.
2. The resulting LongInt value varies from 0 (1-Jan-1600) to \$D6025 (31-Dec-3999).
3. The 'F' data type is only valid in dBase IV.

Example

```

var
  LHPtr : PListHeader;
begin
  ...
  ErrCode := CreateListHeaderUseDBaseFiles(LHPtr, 'DBTEST', 0);
  if (ErrCode <> 0) then
    { error processing }
  ErrCode := CompleteDBaseList(LHPtr, False, False);
  if (ErrCode <> 0) then
    { error processing }
  ErrCode := DBaseImport(LHPtr, 'TEST', WritePascalTypeDef,
    UserInfo,
                                NoCArrConv, StdDecideWrite );
  if (ErrCode <> 0) then
    { error processing }

```

DBaseImport

Creates a new dBASE field list for the dBASE file DBTEST.DBF. Because dBASE data files have an internal record description, there is no need to add field nodes explicitly, CreateListHeaderUseDBaseFiles can do it automatically. The call to CompleteDBaseList completes the field list by calculating all the B-Tree Filer record offsets and widths. The call to DBaseImport passing the completed list converts the dBASE records to B-Tree Filer format.

The new fileblock is called TEST. WritePascalTypeDef writes out a Pascal-style include file using the information in the field list. UserInfo is a progress display routine (the code for it is in DB2ISAM.PAS). NoCArrConv is a routine that does no character conversions. StdDecideWrite filters out all the deleted dBASE records from the data file (all those whose first 4 bytes are non-zero). After DBaseImport completes successfully, the fileblock TEST has been created.

FreeListHeader / NoErrorHandler / NoReXUser

Syntax

```
procedure FreeListHeader; (var LHPtr : PListHeader);
```

Purpose

Release a field definition list.

Description

This routine deallocates all memory used in LHPtr, including all the field nodes. It also functions correctly when LHPtr has been only partially set up (if it has not yet been passed to CompleteSamList or CompleteDBaseList). LHPtr is set to nil.

Example

See the examples for CloseDBaseFiles and CloseSamFiles.

Syntax

```
procedure NoErrorHandler; (Reaction : ErrorReaction; IsamError :  
Integer;  
DBaseError : DBaseErrorNr);
```

Purpose

A default error function that does nothing.

Description

By default, the global variable ProcErrorHandler is initialized with this function.
ProcErrorHandler is called by all the routines of DBIMPEXP when an error occurs.
NoErrorHandler is a do-nothing procedure that ignores all errors.

Syntax

```
function NoReXUser; (Status : WorkStatus; LHPtr : PListHeader;  
ReadRecs, WriteRecs, ErrorRecs : Longint;  
var DatSBuf) : Integer;
```

Purpose

A default progress routine that does nothing.

Description

This routine is a standard function that can be passed as the FuncReXUser parameter for the DBaseImport and DBaseExport functions. NoReXUser does nothing, and returns immediately.

StdDecideWrite / WriteNoTypeDef / WritePascalTypeDef

Syntax

```
function StdDecideWrite; (LHPtr : PListHeader; Errors : Integer;  
var BTBuf, DBBuf ) : DecideCase;
```

Purpose

Standard record filter function.

Description

This routine is a standard function that can be passed as the FuncDecideWrite parameter for the DBaseImport and DBaseExport functions. StdDecideWrite skips deleted records, so they are not written to the output file.

A deleted B-Tree Filer record is assumed to be one with a non-zero value in the first four bytes.

A deleted dBase record is assumed to be one with an asterisk (*) in the first byte.

Example

See the examples for DBaseExport and DBaseImport.

Syntax

```
function WriteNoTypeDef; (LHPtr : PListHeader; IFName : IsamFileName)  
: Integer;
```

Purpose

A type definition writing routine that does nothing.

Description

This routine is a standard function that can be passed as the FuncWriteTypeDef parameter for the DBaseImport function. WriteNoTypeDef is a do-nothing function that does not write any type definition file (include file).

Syntax

```
function WritePascalTypeDef; (LHPtr : PListHeader;  
IFName : IsamFileName) : Integer;
```

Purpose

A type definition writing routine that creates a Pascal include file.

Description

This routine is a standard function that can be passed as the FuncWriteTypeDef parameter for the DBaseImport function. WritePascalTypeDef creates an include file that contains a Pascal record type definition for the record format of the dBase file being converted.

Example

See the example for DBaseImport.

There are two example programs that show the use of the DBIMPEXP unit. DB2ISAM converts any dBase file to a B-Tree Filer fileblock. ISAM2DB converts the example ADDRESS.DAT data file to dBase format.

DB2ISAM is a demo program that demonstrates the use of the DBIMPEXP unit. It converts a dBase III or dBase IV DBF data file, and if one is present the attached DBT memo file, to a B-Tree Filer fileblock format.

The dBase file can contain any number of memo fields of any size; however, the final size of the Filer record cannot be greater than 64KB. If required, a Pascal include file can also be generated during the conversion.

DB2ISAM expects a command line of the following form:

```
DB2ISAM /D dBaseFile [options]
```

dBaseFile is the name for the data file (.DBF), memo file (.DBT), and conversion error log file (.DMP).

The following options can be specified:

```
/I FilerFile   Name for the .DAT and .INC files
/Mn           Maximum size of a Memo field (0 <= n <= 65000)
/S+          Convert dBase character fields to Pascal strings
/T+          Create a Pascal include file
/A-          Insert an auto-incrementing field into the record
```

Note that '-' can be used instead of '/' when specifying command line options.

The /I option allows you to specify the name for the B-Tree Filer data file and, if required, the include file. If no /I parameter is specified, the dBase file name is used. The data file is created with extension '.DAT' and the include file with '.INC'. If the Filer data file already exists, you are prompted whether to overwrite it.

/Mn defines the maximum size of a memo field. If a memo field is larger than this, the extra characters are not converted and the memo is truncated. The default for this option is 512 bytes. Memo fields are converted to null-terminated strings with this maximum size and are always placed at the end of the converted B-Tree Filer record. The values for n can be between 0 (i.e., no memo fields to be converted) and 65000 bytes. Remember that a single Filer record cannot be larger than 65527 bytes. If you attempt to convert a dBase record (with memo fields) that is larger than this, an error message is generated.

The /S option defines whether dBase character fields of length greater than 1 should be converted to null-terminated (C style) or length byte (Pascal style) strings. The default is S+, which forces character fields to be converted to Pascal style strings. This option does not affect memo fields, which are always converted to null-terminated strings.

/T defines whether a Pascal include file containing a record type definition should be created. The default is T+, which causes an include file to be created before the actual conversion takes place. Set this option off (/T-) to suppress the writing of this file.

The /A option defines whether an automatically incremented field is included in the B-Tree Filer record. The default is A-, which means that the additional field is not created. If the option is set to A+, an additional field is inserted into the Filer record, just after the deletion marker field. In the first record converted, this new field is set to 1, in the second record it is set to 2, and so on.

Running DB2ISAM

Before the conversion starts, the number of records in the dBase file is shown. As the conversion progresses, the program displays the number of records converted as a percentage.

To interrupt the conversion process, press any key. You are then prompted for confirmation:

```
Terminate conversion process? (Y/N):_
```

If you enter 'Y', the program terminates immediately after closing all files. The converted Filer file will contain a partial set of converted records in this case. If you enter 'N', the conversion process continues.

After the conversion is complete, the program displays a line similar to the following:

```
Results: 187 records read; 171 records written
```

A difference between the records read and written is due to records that were skipped because they were marked as deleted in the dBase data file or a data conversion error occurred. In the latter case, dBaseFile.DMP contains details about the error, and the program displays:

```
Data conversion errors were found in 16 records.  
The conversion error log file has the details.
```

If a severe error occurs during the execution of the program, it aborts after displaying an error message.

ISAM2DB is a demo program that demonstrates the use of the DBIMPEXP unit. It converts a B-Tree Filer data file to a dBase III or dBase IV file.

Because B-Tree Filer fileblocks do not have any built-in field definitions, any program that wants to convert data from a B-Tree Filer fileblock must have the field definition compiled into the program. ISAM2DB.EXE only converts data from NETDEMO.EXE's ADDRESS.DAT file. Therefore ISAM2DB cannot function in the same generic manner as DB2ISAM, which can convert nearly all dBase files to B-Tree Filer data files.

DB2ISAM expects a command line of the following form:

```
ISAM2DB /I FilerFile [options]
```

FilerFile is the name for the B-Tree Filer data file (.DAT) and index file (.IX). If the program was not written for this file (no record type definition exists), the program terminates. If data conversion errors occur, an error log file (.DMP) containing details (record number and field name) about the errors is produced.

The following options can be specified:

```
/D dBaseFile   Name for the .DBF and .DBT files
/Vn           Type of dBase file to create (3 for dBase III or 4
for dBase IV)
/Kn           Key number for reading the B-Tree Filer fileblock (0
to read in    sequential order, 1 to use the first index, etc)
```

Note that '-' can be used instead of '/' when specifying command line options.

The /D option allows you to specify the family name for the dBase data file and, if required, the memo file. If no /D option is specified, the B-Tree Filer file name is used. The data file is created with extension '.DBF' and the memo file with extension '.DBT'. If the dBase data file already exists, you are prompted whether to overwrite it.

The /V option indicates which format the dBase files should have. To create dBase III compatible files, use /V3. For dBase IV files, use /V4. The default is dBase III format.

/K determines the order in which the B-Tree Filer records are read. Specify /K0 to read the Filer records in sequential order (as they appear in the data file). Specify /Kn, where n is the number of an index for the Filer fileblock, to read the records in key sequence. It is an error to specify a value for n that is greater than the number of indexes for the fileblock. The default is to read the records in sequential order.

Running ISAM2DB

ISAM2DB first compares the name of the B-Tree Filer file to the expected name. It can only convert ADDRESS.DAT from the NETDEMO example program because it must know the Filer record structure. If the file names do not match, the help display is shown and the program is terminated.

Before the conversion starts, the number of records in the B-Tree Filer file is shown. As the conversion progresses, the program displays the number of records converted as a percentage.

To interrupt the conversion process, press any key. You are then prompted for confirmation:

```
Terminate conversion process? (Y/N) : _
```

If you enter 'Y', the program terminates immediately after closing all files. The converted dBase file will contain a partial set of converted records in this case. If you enter 'N', the conversion process continues.

After the conversion is complete, the program displays a line similar to the following:

```
Results: 187 records read; 171 records written
```

A difference between the records read and written is due to records that were skipped because they were marked as deleted in the Filer data file or a data conversion error occurred. In the latter case, FilerFile.DMP contains details about the error, and the program displays:

```
Data conversion errors were found in 16 records.  
The conversion error log file has the details.
```

If a severe error occurs during the execution of the program, it aborts after displaying an error message.

7. Browsers

When a database is used in an interactive application, it's almost always necessary to provide a capability for visually browsing the database. This allows the user to see the relationships among records and select a particular record for updating, deletion, etc.

Unfortunately, the design of a database browser requires the otherwise quite portable code of B-Tree Filer to become more specific to a particular operating system platform and user interface toolbox. For example, the design of a browser that works in graphics mode in a Windows program based on Borland's ObjectWindows Library is quite different from that for a program running in text mode DOS using Object Professional.

Many programmers expect to use a facility built into their interface toolbox when it comes time to browse a database. For example, they might expect to use a PickList object from Object Professional, or a list box from ObjectWindows Library or Turbo Vision. Generally, this is not a good idea. Such built-in list browsers are usually limited in the number of items they can display (the Windows list box can hold at most about 8000 items). And the number of items is often limited to what will fit into memory. Worse yet, there's no efficient way to map an item number needed by the list box to an exact key in the sorted order provided by a database index. As a result, the list box naturally displays items in the physical order of the data file; to do better, some kind of intermediate data structure must be built to get the records in index order before the list box is displayed.

The most important reason why a list box is not appropriate for browsing a database appears in multi-user applications. The list box has no way of reacting to changes made by other workstations.

To solve these problems, B-Tree Filer provides a set of database browsers. These browsers automatically present records in index order, buffer a small set of the records in memory so that capacity is not limited by free RAM, and automatically react to changes made in a multi-user environment.

Specialized browsers are provided for the most popular application frameworks, and an "abstract" browser that you can customize for almost any need is also supplied. Following is a list of the browser modules:

BROWSER

An simple browser that is not specific to any user interface toolbox. You provide function pointers to implement various platform-specific features such as writing each formatted record to the screen.

LOWBROWS, MEDBROWS, HIBROWS

An object-oriented abstract browser used by the following framework-specific browsers. With the exception of a few concepts and type definitions, you don't need to use the objects defined in these modules.

OPBROW

A browser for Object Professional DOS text mode applications, in real mode or protected mode.

TVBROWS

A browser for Turbo Vision DOS text mode applications, in real mode or protected mode.

WBROWSER

A browser for ObjectWindows Library Windows applications.

These browsers share a number of the same capabilities. For example, your application always provides a function that converts the data record passed to it into a string that will be displayed on the screen. All browsers support fixed length records and variable length records. The framework-specific browsers integrate well into the hierarchies of each supported framework; they all provide mouse support, scrollbars, resizeable windows, and so on.

The following sections describe each browser in more detail.

The browser provided by this unit is a simple-to-use utility used to examine a selection of data records from a fileblock. The data records are displayed on-screen using one row per record, in a format that you define. A highlight bar can be positioned to select a particular record. The browser does not require any particular user interface library, but for various reasons and assumptions in the code it is not usable under Windows.

The BROWSER unit is extensively reconfigurable:

- browser input comes from any fileblock, using fixed length or variable length records, in a single user or network environment
- browser output can be positioned within any desired window
- the order in which records are displayed can be based on any index of the fileblock
- low and high key values specify the range of records to display, and a filtering function can be used to select records meeting additional criteria
- formatting and display of records are handled by user-defined procedures whose addresses are passed to the browser at runtime
- commands for vertical and horizontal scrolling of the records are provided
- the keyboard can be remapped and custom exit keys can be defined
- for multi-user applications, a hook allows the browser to automatically update the display whenever another workstation modifies the fileblock
- mouse and scroll bar support are available to users of Turbo Professional or Object Professional

To use the BROWSER unit in your program, just add it to the Uses statement. BROWSER also depends on the following units: CRT, DOS, FILER, and VREC. If you're using Turbo Professional, be sure to edit BTDEFINE.INC and activate the conditional define UseTPCRT. If you're using Object Professional, be sure to activate UseOPCRT in BTDEFINE.INC. In these cases, the setting for UseMouse in TPDEFINE.INC or OPDEFINE.INC respectively determines whether the browser supports the mouse.

The BROWSER unit does not fit into the Object Professional window hierarchy. If you'd like an object-oriented browser, see OPBROW in _7.C, or dearchive FBROWSE.LZH (the installation program put it in the \FILER\BONUS directory) and read FBROWSE.DOC.

The command handling behavior of BROWSER is implemented just like it is in the Turbo Professional units TPPICK, TPENTRY, and others. A configurable command table is defined to map keystrokes to logical commands like "move to next record." Although this table is initialized to reasonable default values, an application can modify it at runtime for special capabilities. See "Keyboard Handling" later in this section for more details.

For correct behavior of the browser, the first four bytes of each data record must be reserved for B-Tree Filer's use, and initialized by the application to zero whenever a record is added to the fileblock.

BROWSER Example

To introduce how the browser works, the example in _4.C can be expanded. First, you must add the CRT (or TPCRT or OPCRT) and BROWSER units to the uses statement.

A minimum of two user-defined routines are needed to control how the browser displays each record: one to create a text string that contains the visible contents of each record, and another to write each such string to the screen at the desired location in the desired attributes. The browser calls the first routine when it reads a new record from the fileblock, and it calls the second when it updates the screen. The second routine is called more frequently, so the separation of the two routines leads to better performance.

The first routine is called BuildARow. It must be global (not nested), it must be compiled with the far model, and it must have exactly the parameters specified. For example:

```
{F+}
procedure BuildARow(var RR : RowRec; KeyNr : Integer;
                   var DatS; DatLen : Word);
  {-Return one row to the browser}
begin
  with RR, PersonDef(DatS) do begin
    if Ref <> -1 then
      Row := FirstName+ ' '+LastName+ ' '+City
    else
      {Record is locked, indicate it on screen}
      Row := '****';
      Row := Pad(Row, 80);
    end;
  end;
end;
```

RowRec is a type interfaced by BROWSER. It contains three fields: IKS, the key string for a record; Ref, the data reference number for the record; and Row, a string that is to be initialized by BuildARow. The RR.IKS and RR.Ref fields are already set when BuildARow is called. The parameter DatS contains the actual data record and DatLen contains the number of bytes in the data record (which is particularly useful for variable length records).

When RR.Ref is set to -1, the browser was unable to read this record because it was locked. BuildARow should set RR.Row to a distinctive string, perhaps a series of asterisks, in this case. If RR.Row contains any other value, BuildARow should combine RR.Row, RR.IKS, and the fields of DatS in any fashion to build a display string. This example just concatenates the FirstName, LastName, and City fields. It is necessary to typecast the untyped DatS parameter to the actual record type to access the data fields.

KeyNr is the key number the browser is using to determine the display order. You can use this parameter to select different display formats for different keys.

The second routine, DisplayARow, is responsible for writing the RowRec to the screen. Again, it must be global, compiled far, and have exactly the specified parameters, as follows:

```
{F+}
procedure DisplayARow(var RR : RowRec; KeyNr, RowNr, StartRow :
Integer;
                   HighLight : Boolean; var HorizOfs : Integer);
  {-Display one row for the browser}
begin
  if HighLight then
    TextAttr := $70
  else
    TextAttr := $07;
  GotoXY(1, StartRow+RowNr-1);
```



```

    Write(RR.Row);
end;

```

All values passed to DisplayARow are initialized by the browser. RR is a RowRec previously set up by the BuildARow routine. Although RR is a Var parameter, DisplayARow should not change it. KeyNr is again the index number being used to determine browsing order. RowNr is the relative row number within the browse window for the record to be displayed. RowNr is in the range 1..NrOfRows, where NrOfRows is the total browsing rows passed to the Browse function. StartRow is the first actual screen row used for the browser, again based on a parameter passed to Browse. Highlight is True if the record has the "highlight bar" (i.e., is the current record). This parameter is often used to select a video attribute. HorizOfs specifies the amount of horizontal scrolling; it can often be ignored, as it is here.

The example routine chooses a video attribute of \$70 (reverse video) when Highlight is True, or \$07 (gray on black) when it's not. It then moves the CRT unit's cursor to the left edge of the screen on the appropriate row and writes the record's previously computed string to the screen. The row strings were previously padded to the maximum screen width. If that is not done, DisplayARow must clear out the end of each line.

The following example shows the variables and main code block needed to activate the browser.

```

var
  PF : IsamFileBlockPtr;
  Pages : LongInt;
  BrowseStatus : Integer;
  P : PersonDef;
  Len : Word;
  RefNr : LongInt;
  KeyStr : IsamKeyStr;
  ExitCmd : BKType;

begin
  Pages := BInitIsam(NoNet, 40000, 0);
  if not IsamOK then begin
    {Error handling}
    Halt;
  end;
  BOpenFileBlock(PF, 'TEST', False, False, False, False);
  if not IsamOK then begin
    {Error handling}
    Halt;
  end;

  RefNr := 1;
  KeyStr := '';
  ExitCmd := BKNone;
  BrowseStatus :=
    Browse(PF,           {Open fileblock pointer}
           False,        {Variable length records?}
           1,            {Index for browse order}
           '',           {Lowest key to display}
           #255,         {Highest key to display}
           2,            {Screen row for first row of browser}
           20,           {Number of rows for browser}
           P,            {Data record buffer for selected record}
           Len,          {Length of returned record}
           RefNr,        {Record number of selected record}
    );

```

```

        KeyStr,          {Key string for selected record}
        ExitCmd,         {Command used to exit browser}
        Nil,             {Special task hook}
        @BuildARow,      {User routine to build each row string}
        @DisplayARow); {User routine to display each row string}

ClrScr;
case BrowseStatus of
  0 : Writeln('Exited on record ', RefNr, ' with exit command ',
ExitCmd);
  1 : Writeln('No keys available in specified range');
  2 : Writeln('FILER error ', IsamError, ' while browsing');
end;
BTExitIsam;
end.

```

The first several lines of code are standard preparation: the FILER unit is initialized via `BTInitIsam` and a fileblock is opened with `BTOpenFileBlock`.

The call to `Browse` activates the browser. Its parameters are discussed in detail later in this section. For now, note that we pass it the addresses of the `BuildARow` and `DisplayARow` routines. `Browse` draws a screen full of records and then takes control of the keyboard to allow the user to scroll around and make a selection.

`Browse` returns two status variables that the application must check. The function result `BrowseStatus` contains overall status from the browser; any value other than zero indicates that an error occurred and therefore no selection could be made. If `BrowseStatus` is zero, then `ExitCmd` contains a command number used to exit the browser. This is typically `BKenter` (user pressed <Enter>) or `BKquit` (user pressed <Esc>). Additional program-defined exit commands are possible as well. At this point, the parameters `RefNr`, `KeyStr`, `Len`, and `P` are also initialized to contain the current record values.

Record Filtering

You may not want to display all the records in a database while browsing. There are three ways to do this:

- when you call `Browse`, specify `LowKey` and `HighKey` values to limit the displayed records to a particular range
- build an index that contains keys for only those records you wish to display
- take advantage of the browser's record filtering hooks

The first approach should be chosen whenever a simple range selection is adequate.

The second approach can be implemented in several ways. It works best when you reserve an index slot for the "browsing index" when the fileblock is created, and add and delete appropriate keys for this index whenever records are added and deleted. In this way, there's no delay to generate the index when you're ready to start the browser.

Sometimes you cannot generate the browsing index in advance, perhaps because the selected records are based on a user-defined specification. You can still create a temporary index just before calling the browser, although your user will have to wait while that index is generated. If you reserved an index slot for this temporary use, you can clear the index by calling `BTDeleteAllKeys` and then add keys for just the selected records to that index. See `BTDeleteAllKeys` in Chapter 5 for an example.

You can even generate a temporary index if you haven't reserved a slot for it when the fileblock was created. The following code fragment shows how.

```

var
  PF : IsamFileBlockPtr;
  TPF : IsamFileBlockPtr;
  IID : IsamIndDescr
  RefNr : LongInt;
  P : PersonDef;
...
{Initialize temporary index descriptor}
IID[1].KeyL := 20;
IID[1].AllowDupK := False;
{Create temporary fileblock}
BTCreateFileBlock('TEMP', SizeOf(PersonDef), 1, IID);
{Open a fileblock using the real data file and a temporary index
file}
BTOpenFileBlock(TPF, 'TEST;TEMP');

{Add the keys to the temporary index}
for RefNr := 1 to BTFileLen(TPF) do begin
  BTGetRec(TPF, RefNr, P, False);
  if SomeConditionIsMet(RefNr, P) then
    BTAddKey(TPF, 1, RefNr, TempBuildKey(1, P));
end;

{Browse using the temporary index}

RefNr := 1;
KeyStr := '';
ExitCmd := BKNONE;
BrowseStatus :=
  Browse(TPF,           {Open fileblock pointer}
        False,         {Variable length records?}
        1,             {Index for browse order}
        '',            {Lowest key to display}
        #255,          {Highest key to display}
        ...);

{Close the temporary fileblock and delete temporary files}
BTCloseFileBlock(TPF)
BTDeleteFileBlock('TEMP');

```

This technique takes advantage of the fact that you can separately specify the data and index file names for a fileblock. In this case the data file from the real fileblock PF is used, but a new index file that exists only temporarily for the use of the browser is created. The technique must be used carefully--if at all--in a network setting; if another workstation modifies the data file, it doesn't know to update the temporary index file and the browser could get dangerously out of sync with reality.

Another point to bear in mind with this technique is that the number of indexes in the temporary fileblock must be equal to the number of indexes in the original fileblock. This is because B-Tree Filer verifies that the number of indexes field in the data file's system record is equal to the number present in the index file when the fileblock is opened. In the above example, the TEST fileblock is assumed to have one index. If it had more, you would have to specify more elements in the IID array. Since the extra indexes would not be used, setting the values to a key length of 1 would be satisfactory.

Because of the disadvantages of the preceding techniques, record filtering hooks are provided. These hooks allow you to filter the desired records in real time using any index.

Suppose you want to display records in last name order, for only those records whose zip code starts with '9'. To do so, you write a record validation function, which the browser calls to determine whether each record meets the filtering requirements. The record validation function can perform whatever logical tests that it requires, including a) reading the data record associated with the key and checking other fields, or b) checking the data in related fileblocks.

Filtering can slow browser operations substantially. Imagine that you have a fileblock containing 100,000 records and that your filtering function accepts only 100 of these records, spread equally throughout the key sequence. In this case, the browser would need to scan 1,000 records for every record that it eventually displayed on the screen.

Use the following routines to control record filtering:

```
procedure EnableFiltering(ValidateFunc : Pointer);
{-Enables Browser filtering. ValidateFunc is a pointer to a user-
defined
function that determines whether a record should be displayed
in the
Browser}

procedure DisableFiltering;
{-Disables Browser filtering. Has no effect if filtering is not
enabled}

function IsFilteringEnabled : Boolean;
{-Returns True if Browser filtering is enabled}
```

ValidateFunc must be the address of a far, non-nested function declared as follows:

```
{ $F+ }
function ValidateARecord(IFBPtr : IsamFileBlockPtr; KeyNr :
Integer;
                                Ref : LongInt; var KeyStr : IsamKeyStr;
                                NetUsed : Boolean) : Boolean;
```

The record validation function returns True if the indicated record should be displayed by the Browser, False if not. When the validation function is called, the browser has not read the data record by calling BTGetRec. That saves time in many applications where the filtering decision can be made simply by checking the key string (KeyStr) or the data reference number (Ref). The validation function can, of course, read the record or perform other tests before making its decision. NetUsed is True if IFBPtr is a network fileblock; this parameter remains for historical reasons (even though network status can be determined from IFBPtr).

SIMPDEMO and NETDEMO provide working examples of record filtering. The F6 command lets you specify fields of the address record to be used as filter masks. Once enabled, the browser displays only those records consistent with the filter masks.

Automatic Screen Refresh

In a network environment, several users can browse and modify a fileblock at the same time. The BROWSE unit is designed to deal with this situation automatically. By default, changes made on another station do not appear on the screen until a keystroke that reloads the page buffers is executed. For example, pressing PgDn causes the browser to read a new set of keys and records from disk; any changes made by other stations appear at that time.

The browser also includes a "refresh function" hook which allows changes to appear in real time, without waiting for a special keystroke. The refresh function's purpose is to detect that another workstation modified the fileblock and signal the browser to update the screen. When activated, the refresh function is called just prior to getting each keyboard command while the browser is active. A refresh function is activated by pointing BROWSER's global typed constant RefreshFunc to a global, far routine declared as follows:

```
{ $F+ }
function RefreshFunction(IFBPtr : IsamFileBlockPtr; KeyNo :
Integer) : Boolean;
```

When no refresh function is desired, RefreshFunc should be set equal to Nil, as it is by default. The Refresh function should return True if a screen refresh is required, and False if not.

BROWSER interfaces two predefined refresh functions, RefreshAtEachCommand and RefreshPeriodically. The first signals for a screen refresh if no keystrokes are pending and the fileblock was modified since the last refresh. The second checks every RefreshPeriod clock ticks to see if the fileblock was modified. If a key is pressed prior to detecting a modification, RefreshPeriodically exits with a False result; otherwise it returns True. The global typed constant RefreshPeriod defaults to 90 clock ticks (about 5 seconds). RefreshPeriodically usually generates less network traffic because it checks for a modification less frequently; however, screen updates won't occur as quickly after other stations modify the data.

The two predefined refresh functions will serve in most applications. However, if your application needs to perform other tasks while waiting for a keypress in the browser, you should use the provided functions as models for routines that perform the additional tasks as well.

SIMPDEMO and NETDEMO provide working examples of the refresh function when dealing with network fileblocks. When these demos are used in a NetWare environment, they activate a special refresh function that uses NetWare semaphore services to detect fileblock modification with very low overhead.

Mouse Support

To use the browser's mouse support, you must have either Turbo Professional or Object Professional and you must activate either UseTPCRT or UseOPCRT in BTDEFINE.INC. Additionally, UseMouse must be defined in TPDEFINE.INC or OPDEFINE.INC. BROWSER uses the TPCRT and TPMOUSE units from Turbo Professional, or OPCRT and OPMOUSE from Object Professional. You can still use the browser *without* mouse support if you just have Borland's CRT unit. If a mouse is present, call EnableBrowseMouse to enable browser mouse support, and DisableBrowseMouse if needed to disable it later.

When the mouse is enabled, clicking over a particular record highlights it; clicking again selects that record (causing Browse to exit the same as if Enter were pressed). The browser optionally displays a vertical scroll bar, including a 'thumb' (known as a slider in Turbo Professional or Object Professional) that indicates the approximate relative position of the highlighted record among all the keys of the browser index. The scroll bar works like one in a Turbo Professional pick list: clicking on the bar scrolls the display to the indicated relative location; clicking on the up arrow of the scroll bar moves the highlight bar up by one row or one page; clicking on the down arrow moves the bar down in the same fashion. If the mouse is outside of the browser screen region when the mouse is clicked, an optional user routine is called.

The following typed constants control the browser's mouse support:

```

AutoScaleMouse; : Boolean = True;
  {Adjust bar for LowKey and HighKey?}
BrowseMouseEnabled; : Boolean = False;
  {True if mouse is enabled}
BrowseMousePage; : Boolean = False;
  {True to scroll by one page per click}
MouseDnMark; : Char = #25;
  {Arrow character at bottom of bar}
MouseUpMark; : Char = #24;
  {Arrow character at top of scroll bar}
MouseX1; : Byte = 1;
  {Left margin for mouse select}
MouseX2; : Byte = 79;
  {Right margin for mouse select}
ScrollBarAttr; : Byte = $07;
  {Video attribute for scroll bar}
ScrollBarAutoSize; : Boolean = True;
  {Match bar to window height?}
ScrollBarCol; : Byte = 80;
  {Absolute column for scroll bar}
ScrollBarHt; : Byte = 18;
  {Height of bar, excluding the arrows}
ScrollBarUp; : Byte = 1;
  {Relative location of bar's up arrow}
ScrollMark; : Char = #178;
  {Character for slider}
ScrollVertChar; : Char = #176;
  {Character for scroll bar background}
SliderAttr; : Byte = $0F;
  {Video attribute for slider}
UserMousePtr; : Pointer = Nil;
  {Address of mouse user routine}
UseScrollBar; : Boolean = True;
  {Display scroll bar?}

```

ScrollBarAttr and SliderAttr are the video attributes used when displaying a scroll bar.

MouseUpMark and MouseDnMark are the characters used for the up and down arrows of the scroll bar. ScrollMark is the character used for the "slider." ScrollVertChar forms the background for the scrollbar.

UserMousePtr, if not Nil, points to a routine of the following form (a global routine compiled under the far model):

```

{$F+}
procedure MouseHotSpotHandler(X, Y : Byte; var Cmd : BKtype);
begin
  ...
end;
{$F-}

```

X and Y are the absolute coordinates of the mouse cursor when the mouse was clicked somewhere outside the browse window. Cmd is the command to be executed next, if any, based on the position

of the mouse. The application should set Cmd to BKNONE if no browser command will be executed as a result of the mouse click.

BrowseMousePage, if True, indicates that clicking on the up/down arrows should scroll the display by a full page.

AutoScaleMouse indicates whether the range of the scroll bar should extend from LowKey to HighKey (the default), or from the lowest possible key to the highest possible key. In the latter case, if LowKey and HighKey specify a constrained range of key values, then the slider is constrained to fall within a corresponding range of the scroll bar.

UseScrollBar indicates whether or not a scroll bar is desired; it is forced to False if a mouse is not present. If UseScrollBar is True, ScrollBarAutoSize indicates that the size and vertical position of the scroll bar should be calculated automatically based on the height and location of the browse window. If UseScrollBar is False, ScrollBarUp and ScrollBarHt indicate the top row and the height of the scroll bar. ScrollBarCol is the absolute screen column in which the scroll bar should appear.

Keyboard Handling

The browser offers a number of hooks for customization of keyboard operations. These features are modeled after those first introduced with Turbo Professional 5.0, so if you're familiar with keyboard handling in TPENTRY or TPEDIT, the browser's features will be easy to use. If not, read on.

There are three kinds of hooks that fall into the category of keyboard handling. The first is the "installable keyboard hook." BROWSER defines a set of logical commands that are tied to a particular set of keystrokes, and it stores the links between commands and keystrokes in an array that can be easily modified. The second is the "background task hook." BROWSER allows the programmer to specify a function that returns a keystroke, and that function is free to perform whatever tasks it wishes while waiting for a key to be pressed. The third is the "context-sensitive help hook." This hook facilitates the creation of context-sensitive help systems by providing a means of invoking a programmer-defined help routine whenever a particular command is issued.

BROWSER's installable keyboard hooks are accessed by using the function AddBrowseCommand. This function reads and modifies a typed constant array of bytes that holds the mapping between logical commands and keystroke sequences. BROWSER's array is called BrowseKeySet, and it can hold up to 201 bytes of mapping information. By default, it is configured to support all the obvious cursor keys for paging through a list of records, and also the traditional WordStar "magic diamond." See AddBrowseCommand for instructions on how to modify this command list while your program runs. For substantial modifications to the list, you have two other options. The first is simply to modify BROWSER's source code to specify the desired keys. The second is to write an installation program that finds the key list by means of an identification string conveniently placed just before the table. Writing such an installation program isn't discussed here; there is an example provided with Turbo Professional (TPKEYS, one of the BONUS units).

Background task hooks allow a program to get some work done while waiting for keyboard input. The most common application of this technique is to display the current time. In a network application, there may be other needs, such as polling for messages from other workstations or attempting to obtain file locks. BROWSER's background task hook makes this easy to do.

To define a background task, you must declare a function of the following form:

```
{ $F+ }  
function MyReadKey : Word;
```

```

begin
  while not KeyPressed do begin
    inline($CD/$28); {Give TSR's a chance to pop up}
    {** Perform background task here **}
  end;
  MyReadKey := BrowseReadKey;
end;
{$F-}

```

The function must be compiled with the far model and must be global (not nested within any other routines). The KeyPressed loop polls for waiting keystrokes and provides continuing opportunities for a background task to execute. The inline statement is optional, but it is sometimes necessary to allow TSRs to pop up while you're waiting for keyboard input. Of course, the background task itself must not take too long, or keyboard response will become sluggish.

To activate the background task, assign the address of your function to BROWSER's variable BrowseKeyPtr, like so:

```
BrowseKeyPtr := @MyReadKey;
```

If you use the automatic screen refresh function RefreshPeriodically, you'll find that the background task facilities described here are effectively disabled. That's because the checking performed for automatic screen refresh is itself a background task. Although the routine you specify with BrowseKeyPtr is still called, it isn't called until a keystroke is already pending and hence your background task won't get any time. If you need to use both RefreshPeriodically and your own background tasks in one program, you must write an automatic screen refresh function that combines both functions.

Activating a context sensitive help hook involves a similar process. When the <F1> key is pressed, BROWSER checks to see if the BrowseHelpPtr pointer is Nil. If it is, the <F1> key is ignored. If not, the routine at the specified address is called. Of course, the <F1> key mapping can be changed using the installable keyboard hook.

The routine called when the help key is pressed must take a very specific form:

```

{$F+}
procedure HelpRoutine(UnitCode : Byte; IdPtr : Pointer; HelpIndex :
Word);
begin
end;
{$F-}

```

The routine must be compiled with the far model, and it must be global. The parameters must match those shown in both type and order. The general form used here is just like that in Turbo Professional, which defines such a routine for a number of units, including those for line editing, data entry screens, and menus. The parameters allow the help routine to determine what unit called it, so that it can provide help in an appropriate form.

When such a routine is called from BROWSER, the parameters take on a special meaning. The first parameter has the value HelpForBrowse, which is a constant declared by BROWSER. This indicates that the call for help came from the database browser. The second parameter, IdPtr, is set to the fileblock pointer being browsed. The third parameter, HelpIndex, is assigned the value BrowseHelpIndex, which is a variable that the application should set just prior to calling Browse. The help hook routine can then decide what help is appropriate to display based on these

parameters. Displaying the help is the application's responsibility; if you have Turbo Professional, the TPHELP unit is ideal for the task.

To activate the help hook, assign the address of your procedure to BROWSER's variable BrowseHelpPtr, like so:

```
BrowseHelpPtr := @HelpRoutine;
```

Another nice feature for browsers is incremental searching. The user must be able to start typing the value of the key that the browser should be positioned at and the browser should automatically do so, moving the highlight with every new keystroke typed. For example suppose the key the user wants is 'SMITH'. Typing 'S' positions the highlight at the start of the keys beginning with 'S'. Typing 'M' positions the highlight at the start of the keys beginning with 'SM', and so on.

To program this kind of interface enhancement requires some work on your part. First you need to declare a global key string to hold the current search string. Initialize it to the null string before starting the browser. Change your current background task routine so that alphabetic characters are converted to a \$FFFF keystroke, and add a command to the browser's command table that maps a \$FFFF keystroke onto bkUser9, for example. In your background task routine, when you get an alphabetic character add it to your global search string. You could even map the backspace keystroke in this way as well, by deleting the last character in your search string if there was one. The browser exits with command bkUser9, at which time you could immediately call BrowseAgain to restart the browser, but pass the your search string as the KeyStr parameter. If the keystroke received by the background task routine was not a alphabetic character, then you set the search string to null.

Here is one implementation of this idea:

```
var
  SearchString : IsamKeyStr;
{$F+}
function MyReadKey : Word;
var
  KeyStroke : Word;
begin
  while not KeyPressed do begin
    inline($CD/$28); {Give TSR's a chance to pop up}
    {** Perform background task here **}
  end;
  KeyStroke := BrowseReadKey;
  if (char(lo(KeyStroke)) in [^H, 'A'..'Z']) then begin
    if (char(lo(KeyStroke)) = ^H) then begin
      if (length(SearchString) > 0) then
        Delete(SearchString, length(SearchString), 1);
    end else
      SearchString := SearchString + char(lo(KeyStroke));
    KeyStroke := $FFFF;
  end else
    SearchString := '';
  MyReadKey := KeyStroke;
end;
{$F-}
...
begin
  BrowseKeyPtr := @MyReadKey
```

```
AddBrowseCommand(bkUser9, 1, $FFFF, 0);
{..start up a browse loop..}
```

Declarations

Constants

A number of typed constants are used for mouse control. These typed constants are declared only if the UseMouse conditional is defined and other conditions are met. See "Mouse Support" earlier in this section for a description of these constants.

```
BKnone      = 00; {Not a command}
BKchar      = 01; {Regular character--not a command}
BKenter     = 02; {Select}
BKquit      = 03; {Escape}
BKfirstRec  = 04; {Cursor to first record}
BKlastRec   = 05; {Cursor to last record}
BKleft      = 06; {Cursor left one column}
BKright     = 07; {Cursor right one column}
BKup        = 08; {Cursor up one row}
BKdown      = 09; {Cursor down one row}
BKpageUp    = 10; {Cursor up one page}
BKpageDown  = 11; {Cursor down one page}
BKplus      = 12; {Reread current record}
BKhelp      = 13; {Invoke help routine}
BKredraw    = 14; {Redraw the browse screen}
BKprobe     = 15; {Signals a mouse event}
BKrowEnd    = 16; {Go to end of row}
BKrowBegin  = 17; {Go to start of row}
BKtask0     = 18; {User-defined task commands}
...
BKtask9     = 27;
BKuser0     = 28; {User-defined exit commands}
...
BKuser9     = 37;
```

These are the codes for each of BROWSER's cursor movement and selection commands. Most of them are self-explanatory. The BKhelp command is activated whenever <F1> is pressed. If a user-defined help routine was specified, it is called at this time.

BKredraw (which is not mapped to any keystroke) is used internally for mouse support. BKprobe is triggered whenever the left mouse button (by default) is pressed. BROWSER then classifies the mouse position and acts accordingly.

By default, neither BKrowEnd nor BKrowBegin is mapped to a keystroke sequence. When BKrowEnd is executed, the browser sets its internal horizontal scrolling offset to 32767 and passes this value to the user DisplayARow routine, which can interpret the value to mean "scroll horizontally as far to the right as possible." The DisplayARow routine must then clip HorizOfs to a realistic value so that subsequent BKleft and BKright commands work correctly. BKrowBegin resets the horizontal offset to zero. NETDEMO and SIMPDEMO demonstrate the use of these commands, mapping them to the End and Home keys respectively.

BKtask0..BKtask9 can be assigned to keystroke sequences that cause BROWSER to call the user-defined special task routine. Keystrokes assigned to BKuser0..BKuser9 causes the browser to exit. The application program can then inspect the returned parameter ExitKey to determine what action to take.

```
BrowseKeyMax = 200;
BrowseKeyID : String[17] = 'browser key array';
BrowseKeySet; : array[0..BrowseKeyMax] of Byte = (...);
```

Used to identify and manage the keystroke to command mapping table. See "Keyboard Handling" earlier in this section for details.

```

{$IFDEF UseTPCRT}
HelpForBrowse; = TpCrt.HelpForXXXX2; {= 7}
{$ELSE}
{$IFDEF UseOPCRT}
HelpForBrowse = 99;
{$ELSE}
HelpForBrowse = 7;
{$ENDIF}
{$ENDIF}

```

Defines the unit index passed to a context sensitive help routine. See "Keyboard Handling" earlier in this section for details.

```
MaxCols = 128;
```

Specifies the maximum width of a one-line formatted record to be displayed by the browser. This can be larger than the width of the physical screen provided that horizontal scrolling is supported by the user-defined routines. It cannot be greater than 255. This constant, in combination with MaxRows, determines the dominant stack space usage of the Browse function. With default values, Browse uses somewhat less than 4000 bytes of stack space.

```
MaxRows; = 20;
```

Specifies the tallest window allowed by the browser. If the actual window (specified in a call to Browse) is taller, BROWSER limits the height to MaxRows. This constant determines the size of a variable local to the Browse function.

```
MinRows; = 4;
```

Specifies the smallest window height allowed by the browser. If the actual window (specified in a call to Browse) is shorter, BROWSER increases the height to MinRows. BROWSER isn't designed to work with a smaller value than 4 for MinRows, so the only option is to make it larger.

```
NoNetMode; : Boolean = False;
```

In a network environment, the browser must read from the disk more often than for a single user application. For example, another workstation might delete the first record currently displayed by the browser, requiring a complete screen update. When NoNetMode is False, the browser rebuilds each display page from scratch after most commands. When it is True, the browser assumes that no records are deleted or modified until the browser exits; in that case, browser performance is much snappier.

```
ReadDataRecord; : Boolean = True;
```

When this constant is set to False, BROWSER does not read the data record associated with each key to display in the browse screen. As a result, the DatS parameter passed to the user BuildARow routine is uninitialized and should not be referenced. Setting ReadDataRecord False improves performance when the key string alone provides adequate information to create the display string, or the information to be displayed will be obtained by indirect means.

```
RefreshFunc; : Pointer = Nil;
```

The address of a user-defined routine that detects fileblock modification by other workstations. See "Automatic Screen Refresh" earlier in this section for more information.

```
RefreshPeriod; : Word = 90;
```

Number of clock ticks that the RefreshPeriodically function waits before checking whether another workstation modified the fileblock. There are approximately 18.2 clock ticks per second, so the default delay is about 5 seconds. If you reduce RefreshPeriod, the browser detects changes made by other workstations more quickly, but file and message activity on the server increases proportionally.

```
RetriesOnLock; : Integer = 50;
```

Number of retries in case of a lock error that occurs when the browser calls BTFindKeyAndRef, BTNextKey, BTPrevKey, BTRedLockFileBlock, or BTSearchKey.

```
RowsToJump; : Integer = 0;
```

Determines how many rows the display scrolls when the highlight bar is at the bottom of the screen and the down arrow is pressed (or conversely, when the bar is at the top and up arrow is pressed). The default value, 0, means that the display scrolls by half of the window height. A common value for RowsToJump is 1, which generates smooth scrolling, at the cost of performance, especially in multi-user environments.

```
UseReadLock; : Boolean = False;
```

When UseReadLock is set to True, BROWSER places a read lock on the fileblock as it builds each page of the browse display. If it can't obtain the read lock in RetriesOnLock attempts, Browse exits and returns the function result 2. A read lock is useful in this situation to prevent another workstation from placing a fileblock lock during the display update, which would cause the browser to pass a Ref number of -1 to the BuildARow routine for one or perhaps all the records.

Types

```
BKtype; = BKnone..BKuser9;
```

BKtype defines the class of actions performed by the browser. It includes such actions as cursor up and down, item select, and browser exit. See the constants above for further details.

```
RowRec =
  record
    IKS : IsamKeyStr;           {Record key}
    Ref : LongInt;             {Record reference number}
    Row : String[MaxCols];     {String to display based on record}
  contents}
end;
```

BROWSER uses this type to manage the record display. The Browse function declares an array of RowRec, portions of which it initializes itself and other parts which are initialized by a routine the application supplies. In particular, Browse initializes the IKS and Ref fields. Then it calls a user-defined routine, pointed to by ProcBuildARow, which generates the Row string. Finally, Browse calls another user-defined routine, pointed to by ProcDisplayARow, which writes the Row string to the screen.

Variables

```
BrowseHelpIndex; : Word;
```

Contains the topic number to be passed to the help routine when it is activated. If help is activated, the application should initialize this variable just prior to each call to Browse.

```
BrowseHelpPtr; : Pointer;
```

Points to a routine that is called when the help key is pressed. By default, this pointer is Nil and no action occurs.

```
BrowseKeyPtr; : Pointer;
```

Points to a routine that reads each keystroke while in the browser. By default, this points either to BrowseReadKey or to ReadKeyWord in TPCRT or OPCRT, routines that return both a scan code and an ASCII character. The application can point this to another routine that returns a key and perhaps performs some background task while awaiting input.

AddBrowseCommand

Syntax

```
function AddBrowseCommand; (Cmd : BKtype; NumKeys : Byte;  
                           Key1, Key2 : Word) : Boolean;
```

Purpose

Modify a keystroke assignment.

Description

You can make three kinds of changes to the key assignments. First, you can add new keystrokes for exit or special task commands. Second, you can disable a particular command. And third, you can change a key assignment.

Cmd is the browser command number to change. NumKeys is 1 if Key1 contains keystroke values, or 2 if Key1 and Key2 contain keystroke values. The key parameters contain key codes consistent with those returned by BrowseReadKey. However, the high byte of each word (the scan code) is ignored unless the low byte (the ASCII character) is zero.

Assume that you want to let a user browse through a database and press to specify that a particular record should be deleted. To implement this, you need to map the key to one of the exit commands:

```
Status := AddBrowseCommand(BKuser0, 1, $5300, 0);
```

The first parameter is the command code. BKuser0 is the first available user-defined exit command, so it is selected. When the user presses , Browse returns the value BKuser0 in its ExitKey parameter.

The second parameter specifies how many keys follow. Acceptable values here are limited to 1 or 2. You would only specify 2 if you wanted the user to press something like <CtrlQ><CtrlR>, two separate keystrokes for a WordStar-like command. In the example, is just one key so 1 is specified. The scan code and ASCII character for follow in the next parameter. The final parameter is ignored in this case, because NumKeys is 1. Another character code would go here if needed.

AddBrowseCommand returns False for three reasons: if the table used to hold the commands is full (the size of the table is 200 bytes for all the commands together); if the NumKeys value is anything but 1 or 2; or if the new keystroke sequence would lead to an ambiguous interpretation of an existing command. Once your program is checked out, you can reasonably expect AddBrowseCommand to return True.

The same technique would be used to enable a special task routine for Browse. In that case, you would choose a command code in the range BKtask0..BKtask9.

Now assume that you want to disable a command. You might want to disable the <Up> and <Down> arrow keys and restrict the user to moving a page at a time. The following sequence does that:

```
Status := AddBrowseCommand(BKnone, 1, $4800, 0); {Disable <Up>}  
Status := AddBrowseCommand(BKnone, 1, $5000, 0); {Disable <Down>}
```

The command code BKnone is a special value that means "no action should occur when this key is pressed," (i.e., the keystroke should be ignored). Again, there is one keystroke for each command.

Finally, assume that you want to change an existing key assignment. For example, maybe you want <CtrlPgDn> to do the same thing as <PgDn>, and <CtrlPgUp> to do the same thing as <PgUp>:

```
Status := AddBrowseCommand(BKpageDown, 1, $7600, 0);  
Status := AddBrowseCommand(BKpageUp, 1, $8400, 0);
```

When AddBrowseCommand scans the key table, it sees that <CtrlPgDn> is already assigned to BKlastRec (move to last record) and simply changes its assignment to BKpageDown. Similarly, <CtrlPgUp> is assigned to BKpageUp rather than BKfirstRec. In both cases, the amount of space remaining in the table for expansion is unchanged.

AddBrowseCommand

See Also

~~[BrowseReadKey](#)~~

Browse

Syntax

```
function Browse; (IFBPtr          : IsamFileBlockPtr;
                  VarRec           : Boolean;
                  KeyNr            : Integer;
                  LowKey           : IsamKeyStr;
                  HighKey          : IsamKeyStr;
                  StartScreenRow   : Integer;
                  NrOfRows        : Integer;
                  var DatS;
                  var DatLen       : Word;
                  var Ref          : LongInt;
                  var KeyStr       : IsamKeyStr;
                  var ExitKey      : BKtype;
                  ProcSpecialTask  : Pointer;
                  ProcBuildaRow    : Pointer;
                  ProcDisplayaRow  : Pointer) : Integer;
```

Purpose

Display fileblock records and allow the user to scroll through them.

Description

IFBPtr points to an open fileblock. The fileblock can be opened for single user, network, and/or variable length modes. VarRec must be True if the fileblock contains variable length records, otherwise it must be False. KeyNr specifies which of the indexes to use for ordering the records displayed by the browser.

The browser displays all keys in the range from LowKey to HighKey, inclusive. Note how HighKey is interpreted. If, for example, HighKey is 'SMITH', then all of the following records would appear in the browse window: 'SMIT', 'SMITH', 'SMITHERS', 'SMITHSON'. Specifying 'SMITH'#0 for HighKey would prevent the last two from appearing.

The combination of KeyStr and Ref specifies the first record to be displayed and highlighted. If KeyStr is less than LowKey, then the first record greater than or equal to LowKey is displayed. If KeyStr is greater than HighKey, then the largest acceptable key is highlighted first.

DatS is a buffer at least as large as the largest (fixed or variable length) record to be displayed. DatS returns the selected record when the browser exits, but is also used as a temporary buffer while browsing. If variable length records are browsed, DatLen returns the length of the selected record.

StartScreenRow specifies the screen row where the first record will appear, and NrOfRows is the number of records visible at once. If NrOfRows is less than MinRows, it is increased to that value. Similarly, if NrOfRows exceeds MaxRows, it is reduced to that value.

ProcBuildaRow and ProcDisplayaRow are pointers to procedures used to create and display a row, respectively. You must provide these procedures and they *must* have a procedure header that matches the example declaration described later in the reference section. Furthermore, these procedures must be declared with the far compilation model, and they must not be nested inside of other procedures. The far model is activated with the {SF+} compiler directive or the far keyword.

The procedure whose address is passed in ProcSpecialTask is called when keys assigned to command codes BKtask0 through BKtask9 are pressed. If no special task is desired, pass Nil for this parameter. ProcSpecialTask, if used, must also point to a procedure whose header exactly matches the one described under SpecialTask, is global, and is compiled under the far model.

The command code that caused Browse to exit is returned in ExitKey. The first action that Browse is to take is also specified in this parameter. If no action is to occur, ExitKey must be initialized to BKnone (= 0).

Browse

See the constants earlier in this section for descriptions of each action that is possible while within the browser. The user remains within Browse until one of the exit keys is pressed, after which control is returned to the calling program.

Browse can return any of three function results:

- 0 Success
- 1 No keys found in the range LowKey..HighKey
- 2 FILER error of class 2 or higher (check IsamError)

When Browse returns the success code of zero, the value of ExitKey can be checked to determine what action to take next. If ExitKey \neq BKquit (any exit key but Escape was pressed), then DatS contains the selected data record, Ref its record number, and KeyStr its key string. When ExitKey = BKquit, all of these parameters are left undefined.

Browse neither clears the screen on entry nor restores it on exit. These actions are the responsibility of the calling program.

Example

See the introduction to this section for a simple example of using Browse. Also see NETDEMO.PAS and SIMPDEMO.PAS for more thorough examples.

See Also

BrowseAgain
DisplayARow

BuildARow
SpecialTask

BrowseAgain

Syntax

```
function BrowseAgain; (IFBPtr          : IsamFileBlockPtr;
                      VarRec           : Boolean;
                      KeyNr            : Integer;
                      LowKey           : IsamKeyStr;
                      HighKey          : IsamKeyStr;
                      StartScreenRow,  : Integer;
                      NrOfRows         : Integer;
                      var HighlightedRow : Integer;
                      var HorizOfs      : Integer;
                      var DatS;
                      var DatLen        : Word;
                      var Ref            : LongInt;
                      var KeyStr        : IsamKeyStr;
                      var ExitKey       : BKtype;
                      ProcSpecialTask   : Pointer;
                      ProcBuildaRow     : Pointer;
                      ProcDisplayaRow   : Pointer) : Integer;
```

Purpose

Browse a fileblock, returning to same location as previous call.

Description

This routine works just like Browse, but it is often more suitable for applications that call the browsing routine repeatedly in a loop. BrowseAgain can retain the exact contents of the browse window, including the position of the highlight bar, from one call to the next. By contrast, Browse recomputes the relative position of the records within the browse window for each call, which can lead to disconcerting visual effects when the routine is called within a loop.

BrowseAgain takes two parameters in addition to those of Browse: HighlightedRow, which specifies the relative screen row where the highlighted bar should appear, and HorizOfs, which specifies the amount of horizontal scrolling. If you maintain the values of these parameters between successive calls to BrowseAgain, the browse display will remain the same when BrowseAgain is recalled, instead of scrolling to values that it computes internally.

Example

```
var
  PF : IsamFileBlockPtr;
  Pages : LongInt;
  BrowseStatus : Integer;
  P : PersonDef;
  Len : Word;
  RefNr : LongInt;
  KeyStr : IsamKeyStr;
  ExitCmd : BKType;
  CurrentRow : Integer;
  HorizOfs : Integer;
...
  CurrentRow := 1;
  HorizOfs := 0;
  RefNr := 1;
  KeyStr := '';
  ExitCmd := BKNONE;
  repeat
    BrowseStatus :=
      BrowseAgain(PF, False, 1, '', #255, 2, 20,
                  CurrentRow, HorizOfs,
                  P, Len, RefNr, KeyStr,
```

BrowseAgain

```
                ExitCmd, Nil, @BuildARow, @DisplayARow);  
if BrowseStatus <> 0 then begin  
    {Error handling}  
end else begin  
    case ExitCmd of  
        {Handle various exit commands}  
    end;  
end;  
until ExitCmd = BKquit;
```

Shows how to initialize the CurrentRow and HorizOfs variables prior to a repeat loop that calls BrowseAgain. The case statement typically handles commands to add a new record, delete a record, search, and so on. See SIMPDemo and NETDemo for complete examples.

See Also

Browse

BrowseReadKey / BuildARow / DisableBrowseMouse

Syntax

```
function BrowseReadKey; : Word;
```

Purpose

Wait for a key to be pressed and return its key code.

Description

For normal keystrokes (the letter 'a', for example), BrowseReadKey returns the ordinal value of the ASCII character (\$61 for 'a'). For extended keystrokes, it returns the scan code of the key in the high byte, and zero in the low byte (e.g., \$4800 for <Up> arrow). These return values are consistent with the way that Browse expects to read keys.

This function is interfaced by BROWSER so that programmer-defined background tasks can call it for basic keyboard input. See "Keyboard Handling" earlier in this section for more information.

Syntax (name may be chosen as desired)

```
procedure BuildARow; (var RR : RowRec; KeyNr : Integer; var DatS;  
DatLen : Word);
```

Purpose

Model routine called by Browse to convert a record to a display string.

Description

The BROWSER unit does not interface a routine by this name; you must declare and implement a routine that matches the one shown here. The routine must be global and compiled with the far model.

On entry to BuildARow, the field RR.IKS is initialized with the key string of the current record, and the field RR.Ref contains the current record number. The parameter DatS contains the complete data record. If RR.Ref equals -1, RR.IKS and DatS are uninitialized because the corresponding record was locked. If the global typed constant ReadDataRecord is False, DatS and DatLen are not initialized.

Prior to exiting, BuildARow must initialize the field RR.Row. This field is a string no longer than MaxCols that contains information to be displayed for the record. The string may be wider than the physical screen, as long as the DisplayARow routine is prepared to display a portion of the string and support horizontal scrolling.

Example

See the introduction to this section for an example.

See Also

Browse

DisplayARow

Syntax

```
{ $IFDEF UseMouse }  
procedure DisableBrowseMouse;;
```

Purpose

Disable mouse control of the browser.

Description

Call this routine to temporarily disable mouse control within the BROWSER unit. This routine points BrowseKeyPtr to the ReadKeyWord routine from Turbo Professional or Object Professional and disables mouse event handling in TPMOUSE or OPMOUSE.

See Also

EnableBrowseMouse

DisableFiltering / DisplayARow / EnableBrowseMouse

Syntax

```
procedure DisableFiltering;;
```

Purpose

Disable record filtering.

Description

The next call to Browse will display all records whose keys fall in the range from LowKey to HighKey.

See Also

EnableFiltering

Syntax (name may be chosen as desired)

```
procedure DisplayARow; (var RR : RowRec; KeyNr : Integer; RowNr : Integer;
                        StartRow : Integer; HighLight : Boolean;
                        var HorizOfs : Integer);
```

Purpose

Model routine called by Browse to display a row string.

Description

The BROWSER unit does not interface a routine by this name; you must declare and implement a routine that matches the one shown here. The routine must be global and compiled with the far model.

The field RR.Row has a string formatted by BuildARow that is to be displayed by DisplayARow. StartRow contains the first row of the browser display, and RowNr reflects the relative position of the row. The absolute row where the record should be written is then StartRow+RowNr-1.

When Highlight is True, the record is currently selected. This should be indicated by the use of reverse video or some other special color.

HorizOfs is used to control horizontal scrolling. If the entire display string fits within the screen, this parameter may be ignored. Otherwise, note that Browse increments this parameter for every press of the Right arrow key and decrements it for every press of Left. The DisplayARow routine may modify the value of HorizOfs to constrain horizontal scrolling to a desired range.

Example

See the introduction to this section. SIMPDEMO and NETDEMO show how to implement horizontal scrolling.

See Also

Browse

BuildARow

Syntax

```
{IFDEF UseMouse}
procedure EnableBrowseMouse;;
```

Purpose

Enable mouse control of the browser.

Description

This routine points BrowseKeyPtr to the ReadKeyOrButton routine from Turbo Professional or Object Professional and enables mouse event handling in TPMOUSE or OPMOUSE. See "Mouse Support" earlier in this section for more information.

DisableFiltering / DisplayARow / EnableBrowseMouse
See Also
DisableBrowseMouse

EnableFiltering / IsFilteringEnabled / SpecialTask

Syntax

```
procedure EnableFiltering;(ValidateFunc : Pointer);
```

Purpose

Enable record filtering.

Description

ValidateFunc is the address of a global, far function whose declaration matches the following prototype:

```
{ $F+ }  
function ValidateARecord(IFBPtr : IsamFileBlockPtr; KeyNr :  
Integer;  
                        Ref : LongInt; var KeyStr : IsamKeyStr;  
                        NetUsed : Boolean) : Boolean;
```

Before accepting any record, the browser calls the validation function to determine whether it meets the filtering criteria. The validation function should return True to accept the record for display; False otherwise. Note that the record has not been read from the data file when the validation routine is called. See "Record Filtering" earlier in this section for more information.

See Also

DisableFiltering

Syntax

```
function IsFilteringEnabled; : Boolean;
```

Purpose

Returns True if a record filtering function is enabled.

Description

This routine is primarily for status reporting.

See Also

EnableFiltering

Syntax (name may be chosen as desired)

```
procedure SpecialTask;(IFBPtr : IsamFileBlockPtr; var DatS; Ref :  
LongInt;  
                    IKS : IsamKeyStr; KeyNr : Integer; var Command  
: BKType;  
                    var ExitCode : Integer; DatLen : Word);
```

Purpose

Model routine called by Browse when a special task key is pressed.

Description

The BROWSER unit does not interface a routine by this name; you must declare and implement a routine that matches the one shown here. The routine must be global and compiled with the far model.

IFBPtr, DatS, Ref, IKS, KeyNr, and DatLen describe the current fileblock and record when the special task key was pressed. These can be used to further define the actions of the special task.

Command contains the command code that caused this procedure to be called. It can take on the values BKtask0..BKtask9. There is an important special meaning for this parameter as well. Its value upon return from SpecialTask determines what the browser will do next. For example, if the special task determines that the browser must execute a <PgDn> action after returning, Command should contain BKpageDown upon exit. If no action is required, be sure to set

EnableFiltering / IsFilteringEnabled / SpecialTask

Command to BKnone (= 0) before exiting the special task routine. Even repeated calls to the special task or recursive calls to Browse are possible. If you make a recursive call to Browse remember that each call consumes about 4000 bytes of stack space.

ExitCode is used to let Browse know if an error occurred. Zero means no error. Any other value causes Browse to terminate and return this value as its function result.

This section describes three low level units that implement an abstract browser object upon which platform-specific layers are built. These units are called LOWBROWS, MEDBROWS, and HIBROWS. The abstract browser objects manage a buffer of records that are to be displayed, but they do not actually get input from the user or write output to the screen. Those actions are reserved for the final, platform-specific layer.

Most of the types and members in the abstract objects are for internal use and are not documented here. The compatibility layers for each platform access the internal methods in ways that are documented for each layer. However, the module LOWBROWS declares some constants and types that affect all of the browsers; those are documented here.

Most of the methods in all levels of the browser hierarchy are implemented as functions returning integer results. The result corresponds to a B-Tree Filer error class ranging from 0 to 4 (see BTIsamErrorClass in Chapter 5). All error classes 3 or larger cause the browser to abort its operation immediately. Error classes of 1 or 2 can abort the current operation in some cases.

LOWBROWS Declarations

Constants

```
BRCurrentlyLocked; = -1;
```

The Status field of a variable of type RowRec (see below) contains this value if the corresponding data record is locked. In this case, the record's data cannot be read successfully. The RowRec is passed as a parameter to the browser window's BuildRow virtual method, which must be overridden. Your BuildRow method should detect the BRCurrentlyLocked status and build an appropriate string for display.

```
BRNoFilterResult; = -2;
```

The Status field of a variable of type RowRec contains this value if filtering is active and the filter routine could not decide whether to accept a particular record because the record was locked.

```
BRFilterError; = -3;
```

The Status field of a variable of type RowRec contains this value if an error occurred during the record filter check (apart from the record being locked, for which BRNoFilterResult is used).

```
BRUserStatStart; = -10
```

Values equal to or more negative than this value can be defined for your own purposes and passed in the RowRec variable.

```
MaxCols = 128;
```

The maximum number of columns that can be displayed in a browser window with the assistance of horizontal scrolling. This value can be increased to a maximum of 255.

```
MaxEltsPerRow; = 8;
```

The maximum number of lines that can be displayed for each record displayed by the browser. However, at this time the browser modules support a maximum of one line per record.

```
NoError;      = 0;  
DialogError;  = 1;  
LockError;    = 2;
```

These constants correspond to the three lowest error classes of B-Tree Filer.

Types

```
BRLRowEltString; = String[MaxCols];  
RowString; = BRLRowEltString;
```

RowString is the type used to hold the display string for each record in the browser. It is a field in the RowRec type described below.

```
BRLRowEltStrArr; = Array[1..MaxEltsPerRow] of BRLRowEltString;
```

BRLRowEltStrArr holds the array of lines used to display each record in the browser. However, at this time the browser modules support a maximum of one line per record.

```
GenKeyStr; = IsamKeyStr;
```

GenKeyStr is an alias for IsamKeyStr, used to pass key string parameters to and from the browser methods.

```
RowRec = record  
  {Read only portion}  
  IKS      : GenKeyStr;  
  Ref      : LongInt;  
  Status   : Integer;  
  RowModified : Boolean;  
  RowBuilt  : Boolean;  
  {Portion to be initialized by BuildRow}  
  case Boolean of  
    True   : (Row : RowString);  
    False  : (RowElt : BRLRowEltStrArr);  
  end;  
  RowRecPtr : ^RowRec;
```

A parameter of this type is passed to many methods of the browsers, including particularly the BuildRow and PerformFilter methods which must be overridden in a user application. IKS is the key string of the browsing index of the corresponding record. Ref is the data reference number of the record. Status equals 0 if the data record could be read successfully, BRCCurrentlyLocked if the record was locked, or other values as defined above. RowModified and RowBuilt are flags used internally by the browser modules.

Row is a string that must be filled in by the BuildRow method. RowElt alternatively specifies multiple lines to be displayed for a given record. However, at this time the browser units support a maximum of one line per record.

Relevant Data Fields

The highest object declared by the units LOWBROWS, MEDBROWS, and HIBROWS is BRHBrowser. Each of the compatibility layers derives a new object from BRHBrowser and then stores a pointer to this new object within a window object native to a particular platform. BRHBrowser contains (or inherits) data fields that can be used in a read-only manner after a browser is initialized. The following declarations can all be used in this fashion:

```
CurRow; : Word;
```

Contains the relative row of the currently highlighted record. Always ranges from 1 to NrofRows.

```
DataBuffer; : Pointer;
```

Pointer to a user-supplied data buffer for reading the current record.

```
DelayTimeOnGetRec; : Word;
```

The time (in milliseconds) between attempts to read a data record when the record is locked.

```
KeyNr; : Word;
```

The fileblock index being used to control the browsing order.

```
LastVarRecLen; : Word;
```

Length of the last variable length record read.

```
LowKey; : GenKeyStr;  
HighKey; : GenKeyStr;
```

These strings contain the lowest and highest keys of the browsing index that the browser will display. LowKey will be set to " and HighKey set to MaxKeyLen bytes of \$FF to display all indexed records.

```
NrOfRows; : Word;
```

Contains the current count of browser lines. NrOfRows is changed automatically when the browser window is resized.

```
RetriesOnGetRec; : Word;
```

The number of times to retry reading a locked record.

```
UsedFileBlock; : IsamFileBlockPtr;
```

The fileblock being browsed.

```
VariableRecs; : Boolean;
```

True when variable length records are being browsed.

```
VarRecMaxReadLen; : Word;
```

The maximum number of bytes that the browser reads in each variable length record.

The OPBROW unit implements an Object Professional compatibility layer over the abstract browser objects. The ISBrowser object implemented by OPBROW is still an abstract object; at least one method, BuildRow, must be overridden in an object you derive from ISBrowser. Several other virtual methods, including PreCompletePage, PostCompletePage, ProcessPreCommand, ProcessPostCommand, PerformFilter, ShowErrorOccured (sic), and CharHandler, can be overridden to customize the behavior of the browser.

See the demonstration program OPISDEMO for examples of how to use this object. See _7.B for additional information about the declarations of the lower level browse units.

User Interface Behavior

The ISBrowser Process method offers appropriate commands to navigate the list of records. The following list arranges them by category and shows the keystrokes or mouse actions that activate each command.

ccLeft	<Left>, <CtrlS>
Horizontally scroll left by one character.	
ccRight	<Right>, <CtrlD>
Horizontally scroll right by one character.	
ccHome	<Home>, <CtrlQ><S>
Horizontally scroll to the beginning of the row.	
ccEnd	<End>, <CtrlQ><D>
Horizontally scroll to the end of the row.	
ccWordLeft	<CtrlLeft>, <CtrlA>
Horizontally scroll left by 10 characters.	
ccWordRight	<CtrlRight>, <CtrlF>
Horizontally scroll right by 10 characters.	
ccUp	<Up>, <CtrlE>, <CtrlW>
Move the highlight bar up one row.	
ccDown	<Down>, <CtrlX>, <CtrlZ>
Move the highlight bar down one row.	
ccPageUp	<PgUp>, <CtrlR>
Move the highlight bar up one page.	
ccPageDn	<PgDn>, <CtrlC>
Move the highlight bar down one page.	
ccTopOfFile	<CtrlPgUp>
Move the highlight bar to the first record.	
ccEndOfFile	<CtrlPgDn>, <CtrlQ><C>
Move the highlight bar to the last record.	

ccSelect <Enter>

Select the current row and exit the browser Process method.

ccMouseSel <ClickLeft>

Move the highlight bar to the indicated row. If lwSelectOnClick is on and the click is over the current record, exit the browser Process method.

ccQuit <Esc>, <CtrlBreak>, <ClickRight>

Quit the browser without making a selection.

ccHelp <F1>, <ClickBoth>

Context sensitive help. Passes ucOPBrowse, this, and the window's help index number to the user-supplied context-sensitive help routine.

bcUpdate <+>

Refresh the browser screen. bcUpdate is a command code that equals ccUser0-1.

ISBrowser supports vertical and horizontal scroll bars using the standard Object Professional methodology. It also supports the OPDRAG module when UseDrag is defined in OPDEFINE.H.

Declarations

Constants

```
lwSelectOnClick; = $0001;
lwSuppressUpdate; = $0002;
```

Options that affect the behavior of the browser. If lwSelectOnClick is on, the browser is exited with the command ccSelect when the mouse is clicked on the current record. If lwSuppressUpdate is off, the browser screen is updated periodically (see the description of SetUpdateInterval later in this section).

```
OpBrKeyMax = 220;                               {Last available slot in
key set}
OpBrKeyID : String[12] = "opbrow keys";       {ID string for install
programs}
OpBrKeySet; : Array[0..OpBrKeyMax] =           {Default key assignments}
(...);
OPBrCfgEnd : Byte = 0;
```

OpBrKeyID is an ID string used to mark the beginning of the configuration data area for OPBROW; OpBrCfgEnd marks the end of the configuration data area. In between them is OpBrKeySet, the command table used by OpBrCommands. OpBrKeyMax is the last valid index into the table. These identifiers are declared in the source file OPBROW.ICD in the BROWSE directory.

```
ucOPBrowse; = 99;
```

The OPBROW unit code, which is passed to a help routine when the ccHelp command is triggered.

Types

```
ISBrowser; = object(CommandWindow)
  PBrowser : LowWinBrowserPtr;
  ...
end;
```

You create a descendant of an ISBrowser object to display a browser using the Object Professional window system. ISBrowser is used like any other CommandWindow. PBrowser points to a LowWinBrowser, which handles the platform-independent work of the browser. The

methods of the ISBrowser are described later in this section. ISBrowser also inherits all methods of the Object Professional CommandWindow object.

```
ISBrowserPtr = ^ISBrowser;
LowWinBrowserPtr = ^LowWinBrowser;
LowWinBrowser = object (BRHBrowser)
    Owner : ISBrowserPtr;
    ...
end;
```

The ISBrowser object instantiates an instance of a LowWinBrowser, which performs all file access and keeps most of the browsing data structures up to date. A pointer to a LowWinBrowser is stored in the PBrowser field of ISBrowser. Owner points back to the ISBrowser instance that owns the LowWinBrowser. See [_7.B](#) for information about the fields that LowWinBrowser inherits.

Variables

```
BadOPBrOptions; : Word = 0;
```

Options used internally by a browser. The lwOptionsOn and lwOptionsOff methods will not change these bits of the browser options. (There are currently no such bits.)

```
DefOPBrOptions; : Word = lwSelectOnClick;
```

The default options for a browser.

```
{IFDEF UseDrag}
OpBrCommands; : DragProcessor;
{ELSE}
OpBrCommands : CommandProcessor;
{ENDIF}
```

The default command processor for a browser object. This variable is initialized in the unit's initialization block. See OPBROW.ICD for details of the key assignments.

Syntax

Purpose

Description

Syntax

Purpose

Description

See Also

Syntax

Purpose

Description

See Also

BuildBrowScreenRow

CharHandler / Done / GetCurrentDatRef

Syntax

```
procedure CharHandler;; virtual;
```

Purpose

Called for each ASCII character entered while ISBrowser.Process is active.

Description

The supplied implementation of this method does nothing. Override it to provide different behavior. You could keep an incremental match string and use the SetAndUpdateBrowserScreen method to reposition the highlight bar on a matching record.

The last character entered is available via the GetLastKey method that ISBrowser inherits from the Object Professional CommandWindow object.

See Also

SetAndUpdateBrowserScreen

Syntax

```
destructor Done; virtual;
```

Purpose

Dispose of the browser.

Description

This destructor deallocates all data structures used by the browser, then calls the inherited CommandWindow.Done. The associated fileblock is *not* closed.

See Also

Init

Syntax

```
function GetCurrentDatRef : LongInt;
```

Purpose

Return the record number of the highlighted record.

Description

This method simply returns the data reference number of the highlighted record.

GetCurrentKeyNr / GetCurrentKeyStr / GetCurrentRec

Syntax

```
function GetCurrentKeyNr : Word;
```

Purpose

Return the number of the browsing index.

Description

This method simply returns the number of the index being used for browsing. (Also available in the KeyNr field of the abstract browser.)

Syntax

```
function GetCurrentKeyStr : String;
```

Purpose

Return the key string for the highlighted record.

Description

This method simply returns the key string associated with the browsing index of the highlighted record.

Syntax

```
function GetCurrentRec(var Match : Boolean) : Integer;
```

Purpose

Read the highlighted record into the record buffer.

Description

This method is typically called after the Process method of ISBrowser returns. It reloads the record that was highlighted by the user into the buffer provided when the browser was initialized.

The boolean variable Match is set to True if the string generated from the newly read record equals the string currently displayed. If the strings are not equal, Match is set to False. This gives a simple check to find out whether the record was changed by another workstation.

An error class is returned as the function result. Any non-zero value indicates that the record could not be read.

See Also

GetThisRec

GetThisRec / Init

Syntax

```
function GetThisRec(var RR : RowRec) : Integer;
```

Purpose

Read the specified record into the record buffer.

Description

RR contains fields named IKS and Ref which describe the key string and reference number of a record. GetThisRec is usually called within a PerformFilter method in order to get complete record data for making a filter decision. The function result is the error class obtained when reading the record.

See Also

PerformFilter

Syntax

```
constructor Init(X1, Y1, X2, Y2 : Byte; AFileBlockPtr :  
  IsamFileBlockPtr;  
                ANumberOfEltsPerRow : Word; ANumberOfRows : Word;  
  AKeyNr : Word;  
                ALKey, AHKey : GenKeyStr; AHeader, AFooter :  
  BRLRowEltString;  
                var ADatS; AIsVarRec : Boolean);
```

Purpose

Initialize a browser window.

Description

This constructor instantiates a browser window using the default window options (DefWindowOptions), the default color set (DefaultColorSet), and the default browser options (DefOPBrOptions). See InitCustom for more details.

InitCustom Syntax

```
constructor InitCustom, (X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;  
AFileBlockPtr : IsamFileBlockPtr;  
ANumberOfEltsPerRow : Word; ANumberOfRows :  
Word;  
AKeyNr : Word; ALKey, AHKey : GenKeyStr;  
AHeader, AFooter : BRLRowEltString; var ADatS;  
AIsVarRec : Boolean; WinOptions : LongInt);
```

Purpose

Initialize a browser window with custom options.

Description

InitCustom first calls CommandWindow.InitCustom, passing it the size indicated by X1, Y1, X2, Y2, the colors indicated by Colors, and the window options WinOptions.

InitCustom allocates a dynamic instance of type LowWinBrowser and stores its address in the field PBrowser.

AFileBlockPtr is the address of an IsamFileBlock, already opened.

ANumberOfEltsPerRow is currently ignored. At some point it will specify the number of screen lines displayed for each browsed record. Currently the number of screen lines per record is always 1.

ANumberOfRows specifies the maximum number of record elements buffered by the browser. The number of rows displayed by the browser window must always be less than or equal to ANumberOfRows.

AKeyNr is the fileblock index number used for selecting and ordering the records displayed by the browser. AKeyNr can be 0, in which case an index is not used, the records are just taken in reference number sequence from the data file.

ALKey and AHKey determine the lowest and highest keys displayed in the browser window. All records that begin with AHKey will be displayed. For example, if AHKey = 'bird', the browser displays records having the key 'bird song'. Pass ALKey as a blank string and AHKey as a string of \$FF characters to display all records in the fileblock.

AHeader and AFooter contain the header and footer lines displayed in the browser window. Specify an empty string to leave out either line. AHeader and AFooter are limited to one line each.

ADatS must point to a buffer large enough to hold the largest record in the fileblock. The browser reads each record into this buffer as needed. The GetThisRec and GetCurrentRec methods also read a record into this buffer.

AIsVarRec must be True for browsing a variable length record fileblock, False otherwise.

Calling this constructor does not cause the browser to access the fileblock or build any display pages. This occurs only when Draw, Process, or SetAndUpdateBrowserScreen is called.

See Also

Done

Init

lwOptionsAreOn / lwOptionsOff / lwOptionsOn

Syntax

```
function lwOptionsAreOn; (OptionFlags : Word) : Boolean;
```

Purpose

Return True if all specified options are on.

Description

This function returns True if the specified browser options are currently selected.

See Also

lwOptionsOff

lwOptionsOn

Syntax

```
procedure lwOptionsOff; (OptionFlags : Word);
```

Purpose

Turn options off.

Description

This procedure deactivates the specified browser option(s).

See Also

lwOptionsAreOn

lwOptionsOn

Syntax

```
procedure lwOptionsOn; (OptionFlags : Word);
```

Purpose

Turn options on.

Description

This procedure activates the specified browser option(s).

See Also

lwOptionsAreOn

lwOptionsOff

PerformFilter

Syntax

```
function PerformFilter(var RR : RowRec; var UseIt : Boolean) :  
Integer; virtual;
```

Purpose

Determine whether to display a given record.

Description

The default implementation of this method accepts all records. You must override it to provide different behavior.

On entry to this function, the IKS and Ref fields of the RR parameter are already initialized. If possible, the filter routine should determine whether to accept the given record by using just the values of these members. If the filter routine needs the complete data record to decide whether to accept the record, it should call the method `GetThisRec` (passing it RR) to load the specified record into the record buffer.

To accept the record for display, `PerformFilter` should set `UseIt` to `True`; to filter the record, it should set `UseIt` to `False`. `PerformFilter` should return a function result of 0 to indicate success; otherwise it should return an error class.

If the record is accepted for display, `PerformFilter` can immediately build the row to be displayed by calling `BuildBrowseScreenRow`. This prevents the browser from having to read the data record a second time.

See Also

`BuildBrowseScreenRow`

`GetThisRec`

PostCompletePage / PreCompletePage

Syntax

```
function PostCompletePage : Integer; virtual;
```

Purpose

Execute an operation after constructing each browser page.

Description

By default this method does nothing. Override it to perform a custom action.

This method is called after constructing the browser page. See PreCompletePage for more information.

PostCompletePage can signal an error by returning a non-zero function result. This causes the browser to call the virtual method ShowErrorOccured with the given result.

See Also

PreCompletePage

Syntax

```
function PreCompletePage : Integer; virtual;
```

Purpose

Execute an operation before constructing each browser page.

Description

By default this method does nothing. Override it to perform a custom action.

A browser page consists of an array of information with one element for each row displayed within the browser window. Each element is of type RowRec (described in _7.B). Whenever a browser command is executed, the browser must rebuild the page of elements to display. The page is constructed in two steps. In the first step, the IKS and Ref fields of each element are filled in by scanning the browse index and calling the PerformFilter method. In the second step, the BuildRow method is called for each of the elements to construct a display string for each one. Note, however, that PerformFilter can construct the display string itself, which obviates the need for the second step.

PreCompletePage is called after the first step is complete. PostCompletePage is called after the second step is complete.

PreCompletePage can signal an error by returning a non-zero function result. This causes the browser to call the virtual method ShowErrorOccured with the given result. In this case, the second step is not executed.

See Also

PostCompletePage

ProcessPostCommand / ProcessPreCommand / ProcessSelf

Syntax

```
procedure ProcessPostCommand;; virtual;
```

Purpose

Execute an operation after reading the keyboard.

Description

By default this method does nothing. Override it to perform a custom action. This method is called after ISBrowser.Process reads the keyboard or obtains a mouse event.

You could call GetCurrentRec within ProcessPostCommand to load the highlighted record into the record buffer. You could then display the current record in a separate window, in more detail than the browser screen shows for all the records.

See Also

ProcessPreCommand

Syntax

```
procedure ProcessPreCommand;; virtual;
```

Purpose

Execute an operation before reading the keyboard.

Description

By default this method does nothing. Override it to perform a custom action. This method is called before ISBrowser.Process reads the keyboard or obtains a mouse event. It is the counterpart to ProcessPostCommand.

See Also

ProcessPostCommand

Syntax

```
procedure ProcessSelf;; virtual;
```

Purpose

Process browser commands.

Description

This method follows the general model for CommandWindow.ProcessSelf with the following additions and exceptions.

It handles the commands described in "User Interface Behavior" earlier in this section.

It calls a number of virtual methods. BuildRow must be overridden. Several other virtual methods, including PreCompletePage, PostCompletePage, ProcessPreCommand, ProcessPostCommand, PerformFilter, ShowErrorOccured (sic), and CharHandler, can be overridden to customize the behavior of the browser.

If Process (which calls ProcessSelf) returns with an error, be sure to check IsamError from the FILER unit for detailed information about fileblock errors.

SetAndUpdateBrowserScreen / SetDimAttr

Syntax

```
procedure SetAndUpdateBrowserScreen(NewKeyStr : GenKeyStr; NewRef : LongInt);
```

Purpose

Move the highlight bar to the specified record.

Description

NewKeyStr and NewRef specify the key string and reference number of a record to highlight. A new browser page is built and the screen is updated immediately if the browser window is current.

You can use this routine after a search is performed on the fileblock, or after a new record is added, to position the highlight on a new record.

See Also

CharHandler

UpdateBrowserScreen

Syntax

```
procedure SetDimAttr(Color, Mono : Byte);
```

Purpose

Set the color for unselected browser lines.

Description

By default, unselected browser lines are displayed in the attribute given by the TextColor or TextMono field of the ColorSet passed to the initializing function. Use this method to change the colors. SetDimAttr does not affect the display until the next time the browser window is drawn.

See Also

SetHighLightAttr

SetHeaderFooter / SetHeaderFooterAttr

Syntax

```
procedure SetHeaderFooter(AHeader, AFooter : BRLRowEltString);
```

Purpose

Change the header and footer lines.

Description

Use this method to change the header and footer specified when the browser window was constructed. Specify an empty string to disable the header or footer.

Calling this method causes the browser page to be rebuilt. If the browser window is current, the screen is updated immediately.

See Also

InitCustom

Syntax

```
procedure SetHeaderFooterAttr(Color, Mono : Byte);
```

Purpose

Set the color for the header and footer lines.

Description

By default, the header and footer are displayed in the attribute specified by the HeaderColor or HeaderMono field of the ColorSet passed to the browser's constructor. Use this method to change the colors. SetHeaderFooterAttr does not affect the display until the next time the browser window is drawn.

See Also

InitCustom

SetHighLightAttr / SetKeyNr / SetLowHighKey

Syntax

```
procedure SetHighLightAttr;(Color, Mono : Byte);
```

Purpose

Set the color for the selected browser line.

Description

By default, the selected browser line is displayed in the attribute specified by the HighlightColor or HighlightMono field of the ColorSet passed to the browser's constructor. Use this method to change the colors. SetHighLightAttr does not affect the display until the next time the browser window is drawn.

See Also

SetDimAttr

Syntax

```
procedure SetKeyNr(Value : Word);
```

Purpose

Set the index number used by the browser.

Description

Value should range between 0 and the largest index number of the fileblock being browsed.

SetKeyNr simply stores the new index number in a field of the browser. You must then specify a new current record and update the screen by calling SetAndUpdateBrowserScreen.

Index 0 is defined to be the arrival sequence of the records in the data file (i.e., the reference number sequence).

See Also

SetAndUpdateBrowserScreen

Syntax

```
procedure SetLowHighKey(ALowKey, AHighKey : GenKeyStr);
```

Purpose

Set new key limits for the browser.

Description

ALowKey and AHighKey specify the new low and high key limits. See InitCustom for more information.

SetLowHighKey simply stores the new key limits in fields of the browser. You must update the browser screen by calling UpdateBrowserScreen or SetAndUpdateBrowserScreen (if the current record is outside of the new key range).

See Also

UpdateBrowserScreen

SetUpdateInterval / ShowErrorOccured

Syntax

```
procedure SetUpdateInterval;(IV : Word);
```

Purpose

Set the update interval in milliseconds.

Description

If the browser is used on a network fileblock, the lwSuppressUpdate option is not enabled, and a non-zero update interval is specified, the browser automatically refreshes the display after the specified interval. As a result, the browser screen automatically accounts for changes to the fileblock made by other workstations.

The default update interval is zero milliseconds, so you must call SetUpdateInterval to activate this feature. Do not specify an interval less than 1000 milliseconds or network thrashing will occur.

Syntax

```
procedure ShowErrorOccured(AClass : Integer); virtual;
```

Purpose

Execute an operation when a browser error occurs.

Description

By default this method does nothing. Override it to perform a custom action.

This method is called whenever an error is detected within the browser. The parameter AClass is the B-Tree Filer error class.

UpdateBrowserScreen

Syntax

```
procedure UpdateBrowserScreen;
```

Purpose

Rebuild and redraw the browser screen.

Description

Call this method when the fileblock is changed in a way that affects the browser screen. For example, if you delete a record, call UpdateBrowserScreen to account for it. If you want to move the highlight to a different record, call SetAndUpdateBrowserScreen instead.

See Also

SetAndUpdateBrowserScreen

The TVBROWS unit implements a Turbo Vision compatibility layer over the abstract browser objects. The TBrowserView object implemented by TVBROWS is also an abstract object; at least one method, BuildRow, must be overridden in an object you derive from TBrowserView. Several other virtual methods, including GetPalette, HandleEvent, PerformFilter, PreCompletePage, PostCompletePage, and ShowErrorOccured (sic), can be overridden to customize the behavior of the browser.

The closely related object TBrowserWindow is a complete Turbo Vision window containing a TBrowserView.

See the demonstration program TVISDEMO for examples of how to use these objects. See _7.B for additional information about the declarations of the lower level browse modules.

The following documentation assumes some familiarity with the concepts of Turbo Vision. Refer to Borland's documentation for further background.

User Interface Behavior

TBrowserView and TBrowserWindow handle the following keyboard events to navigate the list of records.

kbLeft

Horizontally scroll left by one character.

kbRight

Horizontally scroll right by one character.

kbHome

Horizontally scroll to the beginning of the row.

kbEnd

Horizontally scroll to the end of the row.

kbCtrlLeft

Horizontally scroll left by 10 characters.

kbCtrlRight

Horizontally scroll right by 10 characters.

kbUp

Move the highlight bar up one row.

kbDown

Move the highlight bar down one row.

kbPgUp

Move the highlight bar up one page.

kbPgDn

Move the highlight bar down one page.

kbCtrlPgUp

Move the highlight bar to the first record.

kbCtrlPgDn

Move the highlight bar to the last record.

If the browser owns horizontal or vertical scroll bars, the mouse can be used with the scroll bars. Clicking on the vertical scroll bar arrows moves the highlight bar up or down one row. Clicking on the horizontal scroll bar arrows scrolls left or right by one character. Clicking in the scroll region of the bar causes the browser to page up, down, left, or right as appropriate. The scroll bar slider can be used to drag the current record or horizontal position.

By clicking in the active region of the view, the highlight bar can also be moved. Double clicking on a record causes the command cmListItemSelected to be broadcast to the owner of the view.

Declarations

Constants

```
CBInterior; = #2#6#7;
```

The color palette for a TBIInterior object, which is used within a TBrowserWindow.

```
CBrowserView; = #29#27#28;
```

The color palette for a TBrowserView object.

Types

```
LowWinBrowserPtr = ^LowWinBrowser;  
LowWinBrowser = object (BRHBrowser)  
  Owner : PBrowserView;  
  ...  
end;
```

The TBrowserView and TBrowserWindow objects instantiate an instance of a LowWinBrowser, which performs all file access and keeps most of the browsing data structures up to date. A pointer to a LowWinBrowser is stored in the PBrowser field of TBrowserView and TBrowserWindow. Owner points back to the TBrowserView or TBrowserWindow instance that owns the LowWinBrowser. See [_7.B](#) for information about the fields that LowWinBrowser inherits.

```
PBrowserScrollBar = ^TBrowserScrollBar;  
TBrowserScrollBar; = object (TScrollBar)  
  function ScrollStep (Part : Integer) : Integer; virtual;  
end;
```

This object provides specialized scroll bar behavior for the browser.

```
PBrowserView = ^TBrowserView  
TBrowserView; = object (TView)  
  PBrowser : PLowWinBrowser;  
  PHScrollBar,  
  PVScrollBar : PBrowserScrollBar;  
  ...  
end;
```

You create a descendant of a TBrowserView object to display a browser as a view, e.g., as a control within a dialog box. TBrowserView is used like any other Turbo Vision view. PBrowser points to a LowWinBrowser, which handles the platform-independent work of the browser. PHScrollBar and PVScrollBar point to browser scroll bars, or are set to nil when there are no scroll bars. The methods of the TBrowserView are described in the reference section that follows. Note that TBrowserView also inherits all methods of the Turbo Vision TView object.

```
PBInterior = ^TBIInterior;  
TBIInterior; = object (TBrowserView)
```

```

    ...
end;

```

TBInterior is used internally by the TVBROWS unit.

```

    PBrowserWindow = ^TBrowserWindow;
    TBrowserWindow = object(TWindow)
        PInterior : PBrowserView;
    ...
end;

```

TBrowserWindow is a complete window containing a TBrowserView. PInterior points to the TBrowserView. Note that the methods of TBrowserWindow are almost identical to those of TBrowserView. The only real difference is in the constructor, which is described in the reference section following. For the following methods of TBrowserWindow, see the corresponding entry for TBrowserView in the reference section.

```

BuildBrowseScreenRow
BuildRow
Done
GetCurrentDatRef
GetCurrentKeyNr
GetCurrentKeyStr
GetCurrentRec
GetThisRec
PerformFilter
PostCompletePage
PreCompletePage
SetAndUpdateBrowserScreen
SetHeaderFooter
SetKeyNr
SetLowHighKey
ShowErrorOccured
UpdateBrowserScreen

```

BuildBrowseScreenRow / BuildRow / ChangeBounds

Syntax

```
function BuildBrowseScreenRow(var RR : RowRec) : Integer;
```

Purpose

Build a RowRec for the record currently in the record buffer.

Description

This function is provided primarily for use in filter routines. The filter method, PerformFilter, should execute the following sequence. First, it should call GetThisRec to load the record being tested into the record buffer. Then it should decide whether to filter or display the record. If it decides to display the record, it should call BuildBrowseScreenRow, which ultimately calls BuildRow to build the display string for the record. Calling BuildBrowseScreenRow in PerformFilter avoids having to construct the RowRec a second time later. BuildBrowseScreenRow returns an error class, which should also be returned by PerformFilter.

See Also

BuildRow

GetThisRec

Syntax

```
function BuildRow(var RR : RowRec) : Integer; virtual;
```

Purpose

Build the display string for the current row. Must be overridden!

Description

You must override this method in an object derived from TBrowserView. When BuildRow is called, the record buffer (supplied by you when you construct the object) contains the current record (or the portion of it specified by VarRecMaxReadLen for variable length records). The function must initialize the RR.Row field with the string to be displayed for this record. The Status field of RR is already initialized when BuildRow is called. If Status equals zero, the record buffer is properly initialized. The Status field can contain other values as shown in _7.B. When Status is non-zero, the record buffer is *not* initialized. BuildRow should construct an appropriate string to indicate the lock or error condition. The IKS and Ref fields of RR are always initialized when BuildRow is called. BuildRow should return zero if it is successful. Otherwise it should return a B-Tree Filer error class.

See Also

BuildBrowseScreenRow

Syntax

```
procedure ChangeBounds;(var Bounds : TRect); virtual;
```

Purpose

Called by Turbo Vision to change the position or size of the view.

Description

This method overrides a standard Turbo Vision virtual method that is called when the position or size of the view is changed. It first calls the inherited TView.ChangeBounds. Then, if the number of rows in the view has changed, the browser page is rebuilt.

Done / Draw / GetCurrentDatRef / GetCurrentKeyNr

Syntax

```
destructor Done; virtual;
```

Purpose

Dispose of the browser view.

Description

This destructor deallocates all data structures used by the browser, then calls TView's destructor. The associated fileblock is *not* closed.

See Also

Init

Syntax

```
procedure Draw;; virtual;
```

Purpose

Display the browser view.

Description

This method overrides the standard Turbo Vision virtual method for drawing a view. It draws all of the browser rows within the view's bounds.

Syntax

```
function GetCurrentDatRef : LongInt;
```

Purpose

Return the record number of the highlighted record.

Description

This method simply returns the data reference number of the highlighted record.

Syntax

```
function GetCurrentKeyNr : Word;
```

Purpose

Return the number of the browsing index.

Description

This method simply returns the number of the index being used for browsing. (Also available in the KeyNr member of the abstract browser.)

GetCurrentKeyStr / GetCurrentRec / GetPalette

Syntax

```
function GetCurrentKeyStr : String;
```

Purpose

Return the key string for the highlighted record.

Description

This method simply returns the key string associated with the browsing index of the highlighted record. The string is returned as the function result.

Syntax

```
function GetCurrentRec(var Match : Boolean) : Integer;
```

Purpose

Read the highlighted record into the record buffer.

Description

This method is typically called in a HandleEvent method that is responding to a user command to edit or delete a record. It reloads the record that was highlighted by the user into the buffer provided when the browser was initialized.

Match is set to True if the string generated from the just read record equals the string currently displayed. If the strings are unequal, Match is set to False. This gives a simple check to find out whether the record was changed by another workstation.

An error class is returned as the function result. Any non-zero value indicates that the record could not be read.

See Also

GetThisRec

Syntax

```
function GetPalette; : PPalette; virtual;
```

Purpose

Return the address of the view's palette.

Description

The default implementation of this method returns a pointer to a constant with the value CBrowserView. Override this method to use a different palette.

GetThisRec / HandleEvent

Syntax

```
function GetThisRec(var RR : RowRec) : Integer;
```

Purpose

Read the specified record into the record buffer.

Description

RR contains fields named IKS and Ref which describe the key string and reference number of a record. GetThisRec is usually called within a PerformFilter method to get complete record data for making a filter decision. The function result is the error class obtained when reading the record.

See Also

PerformFilter

Syntax

```
procedure HandleEvent;(var Event : TEvent); virtual;
```

Purpose

Handle events for the browser view.

Description

HandleEvent first calls the inherited TView.HandleEvent. If the view's state has the sfSelected flag set, HandleEvent then handles scroll bar, mouse click, mouse move, and keyboard events as described in "User Interface Behavior" in the overview of this section. Otherwise, if the view's ofSelectable option is set and the mouse is clicked on one of the view's scroll bars, the view is selected.

Init

Syntax

```
constructor Init(var Bounds : TRect; PHS, PVS : PBrowserScrollBar;
                 AFileBlockPtr : IsamFileBlockPtr;
                 ANumberOfEltsPerRow : Word; ANumberOfRows : Word;
                 AKeyNr : Word;
                 ALKey, AHKey : GenKeyStr; AHeader, AFooter :
                 BRLRowEltString;
                 var ADatS; AIsVarRec : Boolean);
```

Purpose

Initialize a browser view (a TBrowserView object).

Description

This constructor first calls TView's constructor, passing it the rectangle Bounds. It also stores the scroll bars pointed to by PHS and PVS.

TBrowserView's constructor allocates a dynamic instance of type LowWinBrowser and stores its address in the PBrowser field.

AFileBlockPtr is the address of an IsamFileBlock, already opened.

ANumberOfEltsPerRow is currently ignored. At some point it will specify the number of screen lines displayed for each browsed record. Currently the number of screen lines per record is always 1.

ANumberOfRows specifies the maximum number of record elements buffered by the browser. The number of rows in the browser view must always be less than or equal to ANumberOfRows.

AKeyNr is the fileblock index number used for selecting and ordering the records displayed by the browser. AKeyNr can be 0, in which case an index is not used, the records are just taken in reference number sequence from the data file.

ALKey and AHKey determine the lowest and highest keys displayed in the browser window. All records that begin with AHKey will be displayed. For example, if AHKey = 'bird', the browser displays records having the key 'bird song'. Pass ALKey as a blank string and AHKey as a string of \$FF characters to display all records in the fileblock.

AHeader and AFooter contain the header and footer lines displayed in the browser window. Specify an empty string to leave out either line. AHeader and AFooter are limited to one line each.

ADatS must point to a buffer large enough to hold the largest record in the fileblock. The browser reads each record into this buffer as needed. The GetThisRec and GetCurrentRec methods also read a record into this buffer.

AIsVarRec must be True for browsing a variable length record fileblock, False otherwise.

Calling this constructor does not cause the browser to access the fileblock or build any display pages. This occurs only when Draw or SetAndUpdateBrowserScreen is called.

See Also

Done

PerformFilter

Syntax

```
function PerformFilter(var RR : RowRec; var UseIt : Boolean) :  
Integer;
```

Purpose

Determine whether to display a given record.

Description

The default implementation of this method accepts all records. You must override it to provide different behavior.

On entry to this function, the IKS and Ref fields of the RR parameter are already initialized. If possible, the filter routine should determine whether to accept the given record by using just the values of these fields. If the filter routine needs the complete data record to decide whether to accept the record, it should call the method `GetThisRec` (passing it RR) to load the specified record into the record buffer.

To accept the record for display, `PerformFilter` should set `UseIt` to `True`; to filter (or ignore) the record, it should set `UseIt` to `False`. `PerformFilter` should return a function result of 0 to indicate success; otherwise it should return a B-Tree Filer error class.

If the record is accepted for display, `PerformFilter` can immediately build the row to be displayed by calling `BuildBrowseScreenRow`. This prevents the browser from having to read the data record a second time.

See Also

`BuildBrowseScreenRow`

`GetThisRec`

PostCompletePage / PreCompletePage

Syntax

```
function PostCompletePage : Integer;
```

Purpose

Execute an operation after constructing each browser page.

Description

By default this method does nothing. Override it to perform a custom action.

This method is called after constructing the browser page. See PreCompletePage for more information.

PostCompletePage can signal an error by returning a non-zero function result. This causes the browser to call the virtual method ShowErrorOccured with the given result.

See Also

PreCompletePage

ShowErrorOccured

Syntax

```
function PreCompletePage : Integer;
```

Purpose

Execute an operation before constructing each browser page.

Description

By default this method does nothing. Override it to perform a custom action.

A browser page consists of an array of information with one element for each row displayed within the browser window. Each element is of type RowRec (described in _7.B). Whenever a browser command is executed, the browser must rebuild the page of elements to display. The page is constructed in two steps. In the first step, the IKS and Ref fields of each element are filled in by scanning the browse index and calling the PerformFilter method. In the second step, the BuildRow method is called for each of the elements to construct a display string for each one. Note, however, that PerformFilter can construct the display string itself, which obviates the need for the second step.

PreCompletePage is called after the first step is complete. PostCompletePage is called after the second step is complete.

PreCompletePage can signal an error by returning a non-zero function result. This causes the browser to call the virtual method ShowErrorOccured with the given result. In this case, the second step is not executed.

See Also

PostCompletePage

ShowErrorOccured

SetAndUpdateBrowserScreen / SetHeaderFooter

Syntax

```
procedure SetAndUpdateBrowserScreen(NewKeyStr : GenKeyStr; NewRef : LongInt);
```

Purpose

Move the highlight bar to the specified record.

Description

NewKeyStr and NewRef specify the key string and reference number of a record to highlight. A new browser page is built and the screen is updated immediately if the view is visible.

You can use this routine after a search is performed on the fileblock, or after a new record is added, to position the highlight on a new record.

See Also

UpdateBrowserScreen

Syntax

```
procedure SetHeaderFooter(AHeader, AFooter : BRLRowEltString);
```

Purpose

Change the header and footer lines.

Description

Use this method to change the header and footer specified when the browser window was constructed. Specify an empty string to disable the header or footer.

Calling this method causes the browser page to be rebuilt. If the browser window is visible, the screen is updated immediately.

See Also

TBrowserView

SetKeyNr / SetLowHighKey / ShowErrorOccured

Syntax

```
procedure SetKeyNr(Value : Word);
```

Purpose

Set the index number used by the browser.

Description

Value should range between 0 and the largest index number of the fileblock being browsed.

SetKeyNr simply stores the new index number in a field of the browser object. You must specify a new current record and update the screen by calling SetAndUpdateBrowserScreen.

Index 0 is defined to be the arrival sequence of the records in the data file (i.e., the reference number sequence).

See Also

SetAndUpdateBrowserScreen

Syntax

```
procedure SetLowHighKey(ALowKey, AHighKey : GenKeyStr);
```

Purpose

Set new key limits for the browser.

Description

ALowKey and AHighKey specify the new low and high key limits. See the TBrowserView constructor for more information.

SetLowHighKey simply stores the new key limits in fields of the browser. You must update the browser screen by calling UpdateBrowserScreen or SetAndUpdateBrowserScreen (if the current record is outside of the new key range).

See Also

SetAndUpdateBrowserScreen

UpdateBrowserScreen

Syntax

```
procedure ShowErrorOccured(Class : Integer); virtual;
```

Purpose

Execute an operation when a browser error occurs.

Description

By default this method does nothing. Override it to perform a custom action.

This method is called whenever an error is detected within the browser. The parameter Class is a B-Tree Filer error class.

UpdateBrowserScreen

Syntax

~~procedure UpdateBrowserScreen;~~

Purpose

Rebuild and redraw the browser screen.

Description

Call this method when the fileblock is changed in a way that affects the browser screen. For example, if you delete a record, call UpdateBrowseScreen to account for it. If you want to move the highlight to a different record, call SetAndUpdateBrowserScreen instead.

See Also

SetAndUpdateBrowserScreen

Init

Syntax

```
constructor Init(var Bounds : TRect; ATitle : TTitleStr; ANumber : Integer;
                 ADrvOrFileBlockPtr : Pointer; ANumberOfEltsPerRow : Word;
                 ANumberOfRows : Word; AKeyNr : Word; ALKey, AHKey : GenKeyStr;
                 AHeader, AFooter : BRLRowEltString; var ADatS;
                 AIsVarRec : Boolean);
```

Purpose

Initialize a browser window (a TBrowserWindow object).

Description

This constructor first calls TWindow's constructor, passing it the window bounds (Bounds), the title (ATitle), and the window number (ANumber). Then it constructs the interior of the window as a TBrowserView and also creates vertical and horizontal scroll bars for the window.

ADrvOrFileBlockPtr is the address of an IsamFileBlock, already opened.

ANNumberOfEltsPerRow is currently ignored. At some point it will specify the number of screen lines displayed for each browsed record. Currently the number of screen lines per record is always 1.

ANNumberOfRows specifies the maximum number of record elements buffered by the browser. The number of rows in the browser view must always be less than or equal to ANNumberOfRows.

AKeyNr is the fileblock index number used for selecting and ordering the records displayed by the browser.

ALKey and AHKey determine the lowest and highest keys displayed in the browser window. All records that begin with AHKey will be displayed. For example, if AHKey = 'bird', the browser displays records having the key 'bird song'. Pass ALKey as a blank string and AHKey as a string of \$FF characters to display all records in the fileblock.

AHeader and AFooter contain the header and footer lines displayed in the browser window. Specify an empty string to leave out either line. AHeader and AFooter are limited to one line each.

ADatS must point to a buffer large enough to hold the largest record in the fileblock. The browser reads each record into this buffer as needed. The GetThisRec and GetCurrentRec methods also read a record into this buffer.

AIsVarRec must be True for browsing a variable length record fileblock, False otherwise.

Calling this constructor does not cause the browser to access the fileblock or build any display pages. This occurs only when Draw or SetAndUpdateBrowserScreen is called.

The WBROWSER unit implements an ObjectWindows Library (OWL) compatibility layer over the abstract browser objects. The TBrowserWindow object implemented by WBROWSER is also an abstract object; at least one method, BuildRow, must be overridden in an object you derive from TBrowserWindow. Several other virtual methods, including PreCompletePage, PostCompletePage, ShowFilterWorking, PerformFilter, ShowErrorOccured (sic), and HandleChar, can be overridden to customize the behavior of the browser.

TBrowserWindow is designed to act as either the main window of a Windows application (the MainWindow field of the TApplication object), or as a subsidiary window (child window).

See the demonstration programs OWDEMO and BTWDEMO for examples of how to use this object. See _7.B for additional information about the declarations of the lower level browse units.

The following documentation assumes some familiarity with the concepts of OWL. Refer to Borland's documentation for further background.

User Interface Behavior

The following browser actions are associated with the keyboard:

<Up>, <Down>

Move the highlight bar up or down one row.

<CtrlHome>, <CtrlEnd>

Move the highlight bar to the first or last data record.

<PgUp>, <PgDn>

Move the highlight bar up or down one page.

<Left>, <Right>

Scroll horizontally by one column.

<Home>, <End>

Scroll horizontally to the beginning or end of the browser rows.

<CtrlLeft>, <CtrlRight>

Scroll horizontally by one page.

<+>

Refresh the browser display (useful in multi-user or multi-tasking situations).

Alphanumeric keys

Move the highlight bar to the first record whose selected key string begins with the entered character. Note that the entered character is *not* automatically converted to uppercase. The virtual method HandleChar can be overridden to change the behavior of the browser whenever any ASCII key is pressed.

TBrowserWindow contains methods for handling the following Windows messages.

WM_LButtonDown

Handles all mouse clicks in the client area of the window.

WM_HScroll

Handles horizontal scroll bar events for the browser.

WM_KeyDown

Handles keyboard actions as described above and keeps track of shift keys.

WM_KeyUp

Keeps track of shift keys.

WM_MouseMove

Causes the browser highlight bar to be dragged if the left mouse button is pressed.

WM_SetFocus

Rebuilds and redisplay the browser page.

WM_Size

Rebuilds and redisplay the browser page, if the size has changed.

WM_Timer

Rebuilds and redisplay the browser page, unless timer handling is suppressed.

WM_VScroll

Handles vertical scroll bar events for the browser.

When using the mouse, a click on the left mouse button triggers most actions. When the mouse pointer is on a scroll bar, clicking the left button causes a page move in the direction indicated by the pointer's position relative to the scroll bar thumb. Clicking while the pointer is on a scroll bar arrow causes a single row or column scroll. Pressing the left mouse button while on the thumb allows dragging the thumb to a new location, which takes effect when the mouse button is released. Clicking the mouse over a new row of the browser window moves the highlight to that row. Holding down the left button while the pointer is over the browsing window allows dragging the highlight bar to a new location.

Declarations

Constants

```
HardError;    = 4;  
ProgrammingError; = 5;
```

Convenient constants for error classes reported by B-Tree Filer and the WBROWSER module.

Types

```
FontInfo; = record  
    Font      : THandle;  
    ChHeightExtra,  
    ChHeight,  
    ChWidth,  
    ChRefWidth : Word;  
    FixedPitch : Boolean;  
end;
```

A field of type FontInfo stores information about the font being used by the browser. Font is the handle of the font. FixedPitch is True for a fixed pitch font, False for a proportional font.

The width and height fields are measured in the logical units used by the GetTextMetrics Windows API function. ChHeight is the character height plus external leading (tmHeight+tmExternalLeading from the Windows API TEXTMETRIC structure).

ChHeightExtra is any additional leading used by the browser (0 by default). ChRefWidth is the average character width (tmAveCharWidth). ChWidth is the mean of the average width (tmCharWidth) and the maximum width (tmMaxCharWidth). For fixed pitch fonts, ChWidth and ChRefWidth are equal.

By default the browser uses the "System Fixed Font." To use a different font, override the SetTheFont and/or SetCharValues methods of TBrowserWindow.

```
PLowWinBrowser = ^LowWinBrowser;
LowWinBrowser = object (BRHBrowser)
    Owner : PBrowserWindow;
    ...
end;
```

The TBrowserWindow object is connected to an instance of a LowWinBrowser, which performs all file access and keeps most of the browsing data structures up to date. A pointer to a LowWinBrowser is stored in the BrowserPtr field of TBrowserWindow. Owner points back to the TBrowserWindow instance that owns the LowWinBrowser. See _7.B for information about the fields that LowWinBrowser inherits.

The only method of LowWinBrowser that you will use directly is its constructor. In the Windows browser, you must instantiate a LowWinBrowser explicitly, unlike the Object Professional and Turbo Vision browsers where the LowWinBrowser is instantiated automatically. LowWinBrowser's constructor is documented later in this section.

```
PBrowserWindow = ^TBrowserWindow;
TBrowserWindow = object (TWindow)
    ...
    BrowserPtr : PLowWinBrowser;
    FontDescr : FontInfo;
    HorizOfs : Integer;
    Width,
    FullPage,
    FirstRow,
    MaxHorizOfs: Word;
    TextMargin : TRect;
    ...
end;
```

You create a descendant of a TBrowserWindow object to display a browser using the ObjectWindows Library window system. TBrowserWindow is used just like any other TWindow object.

BrowserPtr points to a LowWinBrowser, which handles the platform-independent work of the browser. FontDescr contains the description of the font being used by the browser. See the FontInfo type for more information. HorizOfs is the current horizontal scrolling offset in pixels, and MaxHorizOfs is the maximum allowable horizontal scroll value in pixels. Width is the current width of the browser window in pixels. FullPage contains the number of rows that can currently be displayed by the browser, not counting any header or footer rows. FirstRow equals 1 if a header is being displayed; otherwise it equals 0.

The methods of the TBrowserWindow are described in the reference section that follows. Note that TBrowserWindow inherits all methods of the OWL TWindow object. However there are a set of methods which perform all of the basic scrolling commands. Rather than describe them individually in the pages that follow, it would be easier to define them here. They are all procedures with no parameters and all are virtual.

FirstPage	scroll to and highlight the first record
LastPage	scroll to and highlight the last record
LeftHome	horizontally scroll to the leftmost column
LineDown	move the highlight down one record
LineLeft	horizontally scroll one column to the left
LineRight	horizontally scroll one column to the right
LineUp	move the highlight up one record

PageDown	move the highlight down one page
PageLeft	horizontally scroll one window width to the left
PageRight	horizontally scroll one window width to the right
PageUp	move the highlight up one page
RightHome	horizontally scroll to the rightmost column

Init Syntax

```
constructor Init (ParOnHeap : Boolean; ADrvOrFileBlockPtr : pointer;  
                  ANumberOfEltsPerRow : Word; ANumberOfRows : Word;  
                  AKeyNr : Word;  
                  ALKey, AHKey : GenKeyStr; var ADatS, AIsVarRec :  
                  Boolean);
```

Purpose

Create an instance of a LowWinBrowser.

Description

ParOnHeap should be True if a dynamic instance is being created (with New) or False if a static instance is being initialized. This parameter allows the TBrowserWindow's destructor to destroy the LowWinBrowser automatically later.

ADrvOrFileBlockPtr is the address of an IsamFileBlock, already opened.

ANumberOfEltsPerRow is currently ignored. At some point it will specify the number of screen lines displayed for each browsed record. Currently the number of screen lines per record is always 1.

ANumberOfRows specifies the maximum number of record elements buffered by the browser. The number of rows in the browser window must always be less than or equal to ANumberOfRows.

AKeyNr is the fileblock index number used for selecting and ordering the records displayed by the browser. AKeyNr can be 0, in which case an index is not used at all, the records are just taken in reference number sequence from the data file.

ALKey and AHKey determine the lowest and highest keys displayed in the browser window. All records that begin with AHKey will be displayed. For example, if AHKey = 'bird', the browser will display records having the key 'bird song'. Pass ALKey as a blank string and AHKey as a string of \$FF characters to display all records in the fileblock.

ADatS must point to a buffer large enough to hold the largest record in the fileblock. The browser reads each record into this buffer as needed. The GetThisRec and GetCurrentRec methods also read a record into this buffer.

AIsVarRec must be True for browsing a variable length record fileblock, False otherwise.

The TDemoWin constructor in BTWDEMO.PAS shows how to initialize a LowWinBrowser and a TBrowserWindow and connect them in the proper way.

See Also

ConnectLowBrowser

AdjustHorizOfs / BuildBrowseScreenRow / BuildRow

Syntax

```
function AdjustHorizOfs; (Delta : Integer) : Integer; virtual; _____
```

Purpose

Set the horizontal offset.

Description

AdjustHorizOfs changes the horizontal offset (measured in pixels) by adding the value passed in Delta. It returns the new horizontal offset, which is always in the range 0..MaxHorizOfs. The data field HorizOfs can only be changed by calling this function, as it performs some essential housekeeping.

See Also

MoveToHorizOfs

Syntax

```
function BuildBrowseScreenRow (var RR : RowRec) : Integer;
```

Purpose

Build a RowRec for the record currently in the record buffer.

Description

This function is provided primarily for use in filter routines. The filter method, PerformFilter, should execute the following sequence. First, it should call GetThisRec to load the record being tested into the record buffer. Then it should decide whether to filter or display the record. If it displays the record, it should call BuildBrowseScreenRow, which ultimately calls BuildRow to build the display string. Calling BuildBrowseScreenRow in PerformFilter avoids having to construct the RowRec a second time later. BuildBrowseScreenRow returns an error class, which should also be returned by PerformFilter.

See Also

BuildRow

GetThisRec

Syntax

```
function BuildRow (var RR : RowRec) : Integer;
```

Purpose

Build the display string for the current row. Must be overridden!

Description

You must override this method in an object derived from TBrowserWindow. When BuildRow is called, the record buffer (supplied by you when you construct the object) contains the current record (or the portion of it specified by VarRecMaxReadLen for variable length records). The function must initialize the RR.Row field with the string to be displayed for this record. The Status field of RR is already initialized when BuildRow is called. If Status equals zero, the record buffer is properly initialized. The Status field can contain other values as shown in _7.B. When Status is non-zero, the record buffer is *not* initialized. BuildRow should construct an appropriate string to indicate the lock or error condition. The IKS and Ref fields of RR are always initialized when BuildRow is called. BuildRow should return zero if it is successful. Otherwise it should return a B-Tree Filer error class.

CalcMaxWidth / CanCallLowBrowser

Syntax

```
function CalcMaxWidth; : Integer; virtual;
```

Purpose

Calculate the maximum width of a browser window in pixels.

Description

The maximum width is computed as MaxCols (defined in LOWBROWS, 128 by default) times FontDescr.ChWidth. Override this method if you need a different formula for the maximum width.

See Also

SetCharValues

Syntax

```
function CanCallLowBrowser; : Boolean; virtual;
```

Purpose

Return True if BrowserPtr points to a valid LowWinBrowser.

Description

If you override virtual methods and you must refer to the BrowserPtr field, you must ensure that CanCallLowBrowser returns True before doing so. If it returns False, your function should perform a safe default action and return.

ConnectLowBrowser / DeleteTheFont / Done

Syntax

```
function ConnectLowBrowser; (ABrowserPtr : PLowWinBrowser;  
                             AHeader, AFooter : BRLRowEltString) :  
    Boolean;
```

Purpose

Connect a LowWinBrowser object and a TBrowserWindow object.

Description

ABrowserPtr is the address of a LowWinBrowser previously instantiated. ConnectLowBrowser returns False if ABrowserPtr is nil, else True. If the instantiation of ABrowserPtr is done when ConnectLowBrowser is called, this provides a convenient way of detecting errors.

AHeader and AFooter contain the header and footer lines displayed in the browser window. Specify an empty string to leave out either line. AHeader and AFooter are limited to one line each. These can also be changed later with a call to SetHeaderFooter.

The TDemoWin constructor in BTWDEMO.PAS shows how to initialize a LowWinBrowser and a TBrowserWindow and connect them.

See Also

SetHeaderFooter

Syntax

```
procedure DeleteTheFont;; virtual;
```

Purpose

Delete the font for the browser window.

Description

This function deletes the font produced by SetTheFont. There is currently no reason for you to override this function.

See Also

SetTheFont

Syntax

```
destructor Done; virtual;
```

Purpose

Dispose of the browser window.

Description

This destructor deallocates all data structures used by the browser, then calls TWindow's Done destructor. The associated fileblock is *not* closed.

See Also

Init

EnableFilter / FilterIsOn / FirstUserInit

Syntax

```
procedure EnableFilter; (On : Boolean);
```

Purpose

Enable or disable calls to the PerformFilter method.

Description

If On is False, an internal field of the browser is set to False and the FilterIsOn function then returns False. The PerformFilter method should detect this condition and accept all records passed to it.

By default the filter is disabled. Pass True to EnableFilter to start filtering.

See Also

PerformFilter

Syntax

```
function FilterIsOn; (var InProgress : Boolean) : Boolean;
```

Purpose

Return True if the filter is enabled.

Description

The result of this function reflects the last call to EnableFilter. It should be used by the PerformFilter method to determine whether filtering is active. It's also handy if you have a menu item to toggle filtering. InProgress is set to True if FilterIsOn is called within certain methods while the browser page is being rebuilt.

See Also

EnableFilter

ShowFilterWorking

Syntax

```
procedure FirstUserInit;; virtual;
```

Purpose

Called once after ConnectLowBrowser is called and a window handle is assigned.

Description

Override this virtual method to perform any browser setup tasks that depend on having a LowWinBrowser attached to the TBrowserWindow and also on having a window handle assigned to the window (HWindow <> 0). This method is called only once, within the TBrowserWindow's SetupWindow method.

See Also

ConnectLowBrowser

GetBrowserTextRect / GetCurNrOfLines

Syntax

```
procedure GetBrowserTextRect; (var TextRect : TRect); virtual;
```

Purpose

Return the rectangle within which the header, footer, and data records are displayed.

See Also

GetRowAreaRect

Syntax

```
function GetCurNrOfLines; : Word;
```

Purpose

Return the current count of the browser lines (not including the header and footer lines).

Description

This method accounts for the pixel height of the window and the font size used for the text.

GetCurrentDatRef / GetCurrentKeyNr

Syntax

```
function GetCurrentDatRef : LongInt;
```

Purpose

Return the record number of the highlighted record.

Description

This method simply returns the data reference number of the highlighted record.

Syntax

```
function GetCurrentKeyNr : Word;
```

Purpose

Return the number of the browsing index.

Description

This method simply returns the number of the index being used for browsing. (Also available in the KeyNr member of the abstract browser.)

GetCurrentKeyStr / GetCurrentRec / GetFooter

Syntax

```
function GetCurrentKeyStr : String;
```

Purpose

Return the key string for the browsing index of the highlighted record.

Description

This method simply returns the key string associated with the browsing index of the highlighted record. The string is returned as the function result.

Syntax

```
function GetCurrentRec(var Match : Boolean) : Integer;
```

Purpose

Read the highlighted record into the record buffer.

Description

The data record for the current browser row can be read by calling this method. The record is read into the buffer specified to the constructor. The function return value contains the encountered error class. Note that it is not necessary to call this method within the BuildRow method.

Match returns True if the display string of the record just read matches the string already in the display buffer of the browser. It returns False if the record has changed.

See Also

GetThisRec

Syntax

```
function GetFooter; : BRLRowEltString;
```

Purpose

Return the current footer string.

Description

This method is an alternative to using SetHeaderFooter, for use when you are continually altering the footer line. The default implementation of this method retrieves the last footer set by SetHeaderFooter or the original footer.

See Also

GetHeader

SetHeaderFooter

Syntax

Purpose

Description

See Also

SetHeaderFooter

Purpose

Description

See Also

Syntax

Purpose

Description

See Also

GetNormalColor

GetLineNrFromY / GetLowHighKey / GetNormalColor

Syntax

```
function GetLineNrFromY; (Y : Integer) : Word;
```

Purpose

Calculate a browser row number given a pixel coordinate.

Description

This method is called by the WMLButtonDown and WMMouseMove message handlers to map the mouse position to a browser row number.

See Also

GetTextOutPosY

Syntax

```
procedure GetLowHighKey; (var ALowKey, AHighKey : GenKeyStr);
```

Purpose

Return the smallest and largest key.

Description

Sets the parameters ALowKey and AHighKey to the smallest and largest keys currently displayed by the browser.

See Also

TBrowserWindow

Syntax

```
procedure GetNormalColor; (var Color, BkColor : TColorRef);
```

Purpose

Define the colors for the Browser window.

Description

This method is called when the browser needs to determine the colors for normal (non-highlighted) lines of the window. By default, it sets the values of the foreground color (Color) and background color (BkColor) to the Windows colors for window text and background (system color indices COLOR_WINDOWTEXT and COLOR_WINDOW). If you would like to use different colors, you must override this function. You can compute Color and BkColor by calling the WINPROCS function named RGB.

See Also

GetHeaderFooterColor

GetNormalColor

GetRowAreaRect / GetSuppressTimer / GetTextOutPosY

Syntax

```
procedure GetRowAreaRect; (var Rect : TRect); virtual;
```

Purpose

Return the rectangle within which the data records are displayed.

See Also

GetBrowserTextRect

Syntax

```
function GetSuppressTimer; : Boolean; virtual;
```

Purpose

Return True if timer messages sent to the browser window are being ignored.

Description

By default, timer suppression is turned off, i.e., timer messages sent to the browser window cause a screen update whenever possible.

See Also

SetSuppressTimer

Syntax

```
function GetTextOutPosY; (LineNr : Word) : Integer; virtual;
```

Purpose

Return the pixel coordinate of a browser row number.

Description

The function result can be passed as parameter to the Windows API function TextOut.

See Also

GetLineNrFromY

GetThisRec / HandleChar / Init

Syntax

```
function GetThisRec(var RR : RowRec) : Integer;
```

Purpose

Read the specified record into the record buffer.

Description

RR contains fields named IKS and Ref which describe the key string and reference number of a record. GetThisRec is usually called within a PerformFilter method to get complete record data for making a filter decision. The function result is the error class obtained when reading the record.

See Also

PerformFilter

Syntax

```
function HandleChar;(var Msg : TMessage) : Boolean; virtual;
```

Purpose

Handle ASCII characters entered within the browser.

Description

This method is always called when the `wm_Char` message is sent to the browser window. The `Msg` parameter contains the original contents of the message, including the character itself in the `wParam` field.

If the function returns `True`, the browser assumes that `HandleChar` has completely handled the message and performs no further action on the character. If `HandleChar` returns `False`, the browser treats the character in its standard way (see "User Interface Behavior" in the overview of this section for more information). Note that `HandleChar` can modify the character it receives (for example, by uppercasing it) and return `False` to have the browser act on the modified character.

The default implementation of this method simply returns `False`. Override it for different behavior.

Syntax

```
constructor Init(AParent : PWindowsObject; ATitle : PChar);
```

Purpose

Initialize a `TBrowserWindow`.

Description

This constructor first calls `TWindow`'s `Init` constructor to initialize the OWL window object. `AParent` and `ATitle` are used here, just as for any other `TWindow` object.

The `TBrowserWindow` constructor then initializes fields of the browser window itself, but it leaves the `BrowserPtr` field set to `nil`, since no `LowWinBrowser` object is attached to it yet.

Next instantiate a `LowWinBrowser` object, then call `TBrowserWindow.ConnectLowBrowser` to connect the two. Most methods of `TBrowserWindow` do nothing until this sequence is completed.

The `TDemoWin` constructor in `BTWDEMO.PAS` shows how to initialize a `LowWinBrowser` and a `TBrowserWindow` and connect them.

See Also

ConnectLowBrowser

Done

MoveToHorizPos / MoveToRelPos / PerformFilter

Syntax

```
procedure MoveToHorizPos; (Pos : Word); virtual;
```

Purpose

Scroll the browser horizontally.

Description

The browser is scrolled horizontally until the pixel position Pos is at the left of the window. The AdjustHorizOfs method is called to ensure that Pos is within range for the browser.

See Also

AdjustHorizOfs

Syntax

```
procedure MoveToRelPos; (Pos : Word); virtual;
```

Purpose

Move the highlight bar to a relative position in the fileblock.

Description

This procedure positions the highlight bar approximately at the relative position Pos in the current browsing index. Pos can take on the values 0 (start of file) to 63 (end of file).

Syntax

```
function PerformFilter(var RR : RowRec; var UseIt : Boolean) :  
Integer; virtual;
```

Purpose

Determine whether to display a given record.

Description

The default implementation of this method accepts all records; override it for different behavior. This method is called for all records, regardless of whether EnableFilter was called with a parameter of True. It should check the value returned by FilterIsOn before doing any real filtering.

On entry to this function, the IKS and Ref fields of the RR parameter are already initialized. If possible, the filter routine should determine whether to accept the given record by using just the values of these fields. If the filter routine needs the complete data record to decide whether to accept the record, it should call the method GetThisRec (passing it RR) to load the specified record into the record buffer.

To accept the record for display, PerformFilter should set UseIt to True; to filter (ignore) the record, it should set UseIt to False. PerformFilter should return a function result of zero to indicate success; otherwise it should return an error class.

If the record is accepted for display, PerformFilter can immediately build the row to be displayed by calling BuildBrowScreenRow. This prevents the browser from having to read the data record a second time.

See Also

BuildBrowScreenRow

GetThisRec

PosClientCorruption / PostCompletePage

Syntax

```
procedure PosClientCorruption;;
```

Purpose

Indicate that the browser window needs to be fully updated.

Description

Many browser routines are optimized to minimize the amount of screen painting they perform. For example LineUp and LineDown use the Windows API routine ScrollWindow to do a fast pixel-level scroll of the window, and then redraw only a single row of the browser.

You should call PosClientCorruption to inform the browser when the entire window needs to be rebuilt and repainted. For example, you should do so when a new record is added to the database or when you change filtering criteria.

Syntax

```
function PostCompletePage : Integer; virtual;
```

Purpose

Execute an operation after constructing each browser page.

Description

By default this method does nothing. Override it to perform a custom action.

This method is called after constructing the browser page. See PreCompletePage for more information.

PostCompletePage can signal an error by returning a non-zero function result. This causes the browser to call the virtual method ShowErrorOccured with the given result.

See Also

PreCompletePage

ShowErrorOccured

PreCompletePage / SetAndUpdateBrowserScreen

Syntax

```
function PreCompletePage : Integer; virtual;
```

Purpose

Execute an operation before constructing each browser page.

Description

By default this method does nothing. Override it to perform a custom action.

A browser page consists of an array of information with one element for each row displayed within the browser window. Each element is of type RowRec (described in _7.B). Whenever a browser command is executed, the browser must rebuild the page of elements to display. The page is constructed in two steps. In the first step, the IKS and Ref fields of each element are filled in by scanning the browse index and calling the PerformFilter method. In the second step, the BuildRow method is called for each of the elements to construct a display string for each one. Note, however, that PerformFilter can construct the display string itself, which obviates the need for the second step.

PreCompletePage is called after the first step is complete. PostCompletePage is called after the second step is complete.

PreCompletePage can signal an error by returning a non-zero function result. This causes the browser to call the virtual method ShowErrorOccured with the given result. In this case, the second step is not executed.

See Also

PostCompletePage

ShowErrorOccured

Syntax

```
procedure SetAndUpdateBrowserScreen(NewKeyStr : GenKeyStr; NewRef :  
LongInt);
```

Purpose

Move the highlight bar to the specified record.

Description

NewKeyStr and NewRef specify the key string and reference number of a record to highlight. A new browser page is built and the screen is updated immediately if the browser window is visible.

You can use this routine after a search is performed on the fileblock, or after a new record is added, to position the highlight on a new record.

See Also

UpdateBrowserScreen

SetCharValues / SetHeaderFooter / SetKeyNr

Syntax

```
procedure SetCharValues;; virtual;
```

Purpose

Set the character width and height based on browser font.

Description

The browser calls this function to initialize the following data fields in the FontDescr record: ChHeight, ChWidth and ChRefWidth. It uses the Windows API GetTextMetrics call to do so. The FontDescr.ChHeightExtra field is set to 0.

Override this method if you need to apply another calculation algorithm to determine the value of these fields.

See Also

SetTheFont

Syntax

```
procedure SetHeaderFooter(AHeader, AFooter : BRLRowEltString);
```

Purpose

Change the header and footer lines.

Description

Use this method to change the header and footer specified when the browser window was constructed. Specify an empty string to disable the header or footer.

Calling this method causes the browser page to be rebuilt. If the browser window is visible, the screen is updated immediately.

See Also

TBrowserWindow

Syntax

```
procedure SetKeyNr(Value : Word);
```

Purpose

Set the index number used by the browser.

Description

Value should range between 0 and the largest index number of the fileblock being browsed.

SetKeyNr simply stores the new index number in a field of the browser object. You must specify a new current record and update the screen by calling SetAndUpdateBrowserScreen.

Index 0 is defined to be the arrival sequence of the records in the data file (i.e., the reference number sequence).

See Also

SetAndUpdateBrowserScreen

SetLowHighKey / SetMargins / SetSuppressTimer

Syntax

```
procedure SetLowHighKey(ALowKey, AHighKey : GenKeyStr);
```

Purpose

Set new key limits for the browser.

Description

ALowKey and AHighKey specify the new low and high key limits. See the TBrowserWindow constructor for more information.

SetLowHighKey simply stores the new key limits in fields of the browser. You must update the browser screen by calling UpdateBrowserScreen or SetAndUpdateBrowserScreen (if the current record is outside of the new key range).

See Also

SetAndUpdateBrowserScreen UpdateBrowserScreen

Syntax

```
procedure SetMargins;; virtual;
```

Purpose

Initialize the text margins.

Description

The browser calls this method to initialize the TextMargin field of the object. By default it sets the margins to 0 pixels on all sides. Override this method to allow for non-zero margin values.

Syntax

```
function SetSuppressTimer;(DoSuppr : Boolean) : Boolean;
```

Purpose

Suppress or enable timer message handling in the browser.

Description

Timer message handling is enabled by default. When it is enabled, the browser updates its screen whenever a timer message is received, the window is not iconized, and the browser is fully initialized.

Pass True to SetSuppressTimer to disable timer message handling. The function returns the previous state of timer handling in its function result.

See Also

GetSuppressTimer

SetTheFont / SetupWindow / ShowErrorOccured

Syntax

```
procedure SetTheFont;; virtual;
```

Purpose

Set the font for the browser window.

Description

If you would like to use a special font for the Browser window, you must override this method and assign a new font handle to the FontDescr.Font field of the object (for example, by calling CreateFontIndirect). If the font uses a fixed character width, you must set lwFont.FixedPitch to True, otherwise to False.

By default SetTheFont initializes the FontDescr.Font field to the font handle returned by the Windows API call GetStockObject(SYSTEM_FIXED_FONT). It then sets the family to FF_MODERN, the pitch to FIXED_PITCH, and the weight to FW_NORMAL. It also sets FontDescr.FixedPitch to True.

See Also

DeleteTheFont

SetCharValues

Syntax

```
procedure SetupWindow;; virtual;
```

Purpose

OWL calls this method as soon as a TWindow object is assigned a window handle.

Description

TBrowserWindow's implementation of SetupWindow first calls the inherited TWindow.SetupWindow. Then, if the BrowserPtr field of TBrowserWindow is not nil, it calls lwFirstInit, which performs additional initialization including a call to SetTheFont and FirstUserInit.

See Also

FirstUserInit

SetTheFont

Syntax

```
procedure ShowErrorOccured(AClass : Integer); virtual;
```

Purpose

Execute an operation when a browser error occurs.

Description

By default this method just produces a beep. Override it to perform a custom action. This method is called whenever an error is detected within the browser. The parameter AClass is the B-Tree Filer error class.

ShowFilterWorking / TotalCharHeight

Syntax

```
procedure ShowFilterWorking(CallState : Integer; Rejected :  
Boolean); virtual;
```

Purpose

Called by the browser as a hook to indicate that the filter is working.

Description

This method is called immediately after each call to PerformFilter to filter a record from the display. If the record was rejected, the Rejected parameter passed to ShowFilterWorking is True, otherwise False.

ShowFilterWorking is called once before actual filtering begins, with the CallState parameter set to -1. This provides an opportunity for the application to instantiate a window or prepare the screen in some other way. For each record that is tested by PerformFilter, CallState is set to 0. After the page is built, a final call is made with CallState set to 1, which allows the application to destroy the window or otherwise clean up.

The default implementation of ShowFilterWorking does nothing. Override it to provide custom status while filtering is active. When a filter is quite restrictive, selecting just a few records from a large database, a ShowFilterWorking method is important so that the user understands the long delay before a browser window is updated. The method also can allow the user to abort the filtering by pressing <Esc> or clicking on a menu item.

See Also

EnableFilter

Syntax

```
function TotalCharHeight : Word; virtual;
```

Purpose

Return the height of one browser row in pixels.

Description

This function returns the sum of FontDescr.ChHeight and FontDescr.ChHeightExtra.

See Also

SetCharValues

UpdateBrowserScreen / UseSeparator / WriteStringOut

Syntax

```
procedure UpdateBrowserScreen;
```

Purpose

Rebuild and redraw the browser screen.

Description

Call this method when the fileblock has been changed in a way that affects the browser screen. For example, if you delete a record, call UpdateBrowseScreen to account for it.

See Also

SetAndUpdateBrowserScreen

Syntax

```
function UseSeparator;(var Color : TColorRef) : Boolean;
```

Purpose

Determine whether a dividing line should be displayed between the header or footer and the body of the browser window.

Description

The browser window calls this function to determine whether separator lines are displayed. The default implementation returns False, which indicates that no separators should be used. To obtain a separator, override this method and return True. Also, return the color value to be used for the separator. You can compute Color by calling the WINPROCS function named RGB.

See Also

GetHeaderFooterColor

Syntax

```
function WriteStringOut;(var S : String; LineNr : Word; DC : HDC;  
                        XOfs : Integer) : Word; virtual;
```

Purpose

Output a browser string (header, footer, or data row).

Description

This function outputs the string S to the text row LineNr at a horizontal offset of XOfs pixels. LineNr equals 0 for the header, 1 for the first data row, etc. DC is the device context. To obtain the Y position of the write, you must call GetTextOutPosY; this function's result is the result of that call.

The browser calls this function while painting the window. You could override it if, for example, you were using a proportional font in the browser and you wanted to have WriteStringOut call the Windows API routine TabbedTextOut instead of TextOut to display the string.

When the browser calls this method, the font and the colors of the line have already been selected into the display context.

8. Network Utilities

The features of B-Tree Filer described so far let you write databases that run quickly and reliably in almost any network environment. There is often more to writing an application that really takes advantage of a network, however. First, the application should verify that it indeed does have access to a desired network. Shared support files need to be locked and unlocked. Printed reports must be managed. And direct communication between workstations opens a new realm of program functionality, especially in the emerging era of "groupware."

Keeping these needs in mind, several additional high powered units are provided with B-Tree Filer. Recognizing Novell's dominant position in PC networking today, several units provide Novell-specific capabilities:

- NWBASE - basic access to NetWare
- NWCONN - connection information
- NWBIND - access to the NetWare bindery
- NWMSG - very basic messaging capabilities
- NWIPXSPX - full access to NetWare's IPX and SPX inter-workstation communications
- NWFILE - NetWare file and directory functions
- NWPRINT - printer capturing and access to print queues
- NWSEMA - semaphore access
- NWTTS - transaction tracking

In addition, for situations that do not require full NetWare capabilities, the NETBIOS unit offers access to inter-workstation message-passing routines that are supported by almost all modern networks, including Novell, 3Com, PC LAN, and PC-NET. And the SHARE unit encapsulates the file locking routines supplied with the MS-DOS 3.x SHARE utility and also some routines specific to MS-NET and PC LAN.

There are several demonstration programs to demonstrate what the network utilities can do. NETINFO identifies what kind of network, if any, is active and reports everything it can determine about the network. NSEND, NISEND and NBSEND all do the same thing using different techniques: they copy files from one workstation to another (without using a shared server file). NSEND uses SPX services, NISEND uses IPX services and NBSEND uses NetBIOS session services, all of which you'll learn more about later in this chapter. SPX2WAY is a simple chat-type program that uses SPX communications to transmit messages from one workstation to another. BINDLIST lists all of the objects and properties in the bindery for a server. TTSFILER gives an example of how to use Novell's Transaction Tracking Services. These demonstration programs are described in the last section of this chapter.

For a general introduction to network concepts, refer to any of the following books:

Understanding Local Area Networks

Stan Schatt, Howard W. Sams & Co, 1988.

Local Area Networks, The Second Generation

T.W. Madrone, John Wiley and Sons, 1988.

Communications and Networking for the IBM PC and Compatibles

Larry Jordan, Brady Books, Simon & Schuster, 1986.

Novell supplies only C language routines to access the power of their Advanced NetWare operating system. This leaves most Turbo Pascal programmers out in the cold! The NetWare units supplied with B-Tree Filer implement many of the NetWare application program interfaces (APIs)--services to access the functions of a NetWare local area network (LAN). The tools in the NetWare units access such facilities as the printer capture, print queues, messaging services, direct station-to-station communications, NetWare's Transaction Tracking Services (TTS), NetWare's semaphores and others.

There are actually many more routines in the NetWare API than are implemented here. These were chosen to best complement the routines in B-Tree Filer. If you need others, the B-Tree Filer source code, together with Novell's documentation, should allow you to implement what you need.

The routines in this unit are only for NetWare, and they require the following environment. On the workstation you must have version 3.22 or higher of the NETX shell, or version 1.00 or higher of the VLM Requester (note that NETX.VLM does not have to be loaded in this case). On the server(s) you must have NetWare 2.15 or above (and that includes NetWare 3.x and NetWare 4.x of course). The units as a whole were tested with the following: NETX 3.22 and 3.32PTF, VLM 1.10 and 1.20, NetWare 3.11 and NetWare 4.01. The rest of this section refers to both the NETX shell and the VLM Requester as the NetWare shell.

The routines in these NetWare units use the NetWare Core Protocol (NCP) calls wherever possible. This is a direct departure from the previous B-Tree Filer NETWARE unit, which used specific NETX calls, and means that the VLM Requester can be supported without loading Novell's NETX.VLM emulation module. The later versions of the NCP calls were used wherever possible (i.e., those that support 1000-user NetWare, and those that support larger print queues). Another impetus to this change was that finally, after many years, Novell documented the NCP calls in their Client API for Assembly documentation.

This chapter assumes that you are familiar with the general concepts of local area networking, and that you have some knowledge of Novell's Advanced NetWare product line. Specifically, you should know what servers, workstations, the bindery, and print queues are. This manual makes no attempt to explain how NetWare works, nor does it explain network cabling, topologies, or theory. For further background, see the following references:

Novell Client API for C

Available directly from Novell on CD-ROM and includes the Client API for Assembly.

Programmer's Guide to NetWare

Charles G. Rose, McGraw-Hill, ISBN 0-07-607029-8

If you delve into the source code for the NetWare units, you will notice that some identifiers are interfaced but not documented. Although these identifiers all serve useful functions, documenting them would turn this chapter into a tome the size of Novell's complete developer documentation. The types and routines that are documented here are those that are reasonably accessible to a programmer who isn't a full time network programmer.

Another documentation area that could quickly grow unreasonably large is the NetWare error codes (even Novell's official documents have error codes missing). Some error codes are documented, but the list is not exhaustive. In these units error codes are word-sized, the most significant byte is the source of the error (0 means DOS, \$7F means an internal NWXXXX unit error, \$81 means a NETX

error, \$88 means a NetWare shell error, \$89 means a server error). The least significant byte is the specific error code, but note that many of these codes are reused by Novell for different functions.

NWBASE provides low-level access to the server from a workstation. It is essentially the foundation upon which the rest of the NetWare units are built. In your application, you will probably not call any of the NWBASE routines, except the ones that give information about the NetWare shell. NWBASE also defines generic types that are used in other NetWare units.

NWBASE provides functions to:

- determine the type of NetWare shell present and its version
- call a server via an NCP call
- issue an interrupt

NWBASE also has several interfaced internal routines, types, and variables that are not documented here. For more information, browse the source code for NWBASE.

Declarations

Constants

```
nwErrDPMI          = $7F01; {a DPMI access problem (no DOS memory,
etc.)}
nwErrWrongVer      = $7F02; {server is the wrong version to support
the call}
nwErrShell         = $7F03; {shell doesn't exist or is the wrong
version}
nwErrMemory        = $7F04; {out of memory}
nwErrIntr          = $7F05; {error on a real mode interrupt}
nwErrBadData       = $7F06; {bad data was passed to a routine}
nwErrTooManyConns = $7F07; {there are too many connections for the
routine}
nwErrNoMoreConns  = $7F08; {there are no more connections to
process}
```

Internal error codes. Individual routines will state if they generate any of these error codes.

Types

```
nwDayOfWeek = (nwSun, nwMon, nwTue, nwWed, nwThu, nwFri, nwSat);
TnwDate = record
  Year   : Word;
  Month  : Byte;
  Day    : Byte;
  Hour   : Byte;
  Minute : Byte;
  Second : Byte;
  WeekDay : nwDayOfWeek;
end;
```

The types defining dates for the NetWare units. The year field contains values in the range 1980..2079; the ranges of the other fields are defined with their usual meanings (for example Month ranges from 1 to 12, Hour from 0 to 23, and so on). Note that some routines do not return the WeekDay field properly.

```
nwInt = Integer;           {16-bit signed integer}
nwLong= LongInt;           {32-bit signed integer}
```

Definition of a 16 bit signed integer and a 32 bit signed integer for the NetWare units.

```
PhysicalNodeAddress; = array[1..6] of Byte;
IPXAddress; = record
```

```

    Network : nwLong; {high-low}
    Node    : PhysicalNodeAddress;
    Socket  : Word; {high-low}
end;

```

An internetwork address on a NetWare network. Used extensively by the IPX and SPX communication routines.

```
TnwErrorCode; = Word;
```

A NetWare error code type. The high byte is the source of the error, the low byte the individual error. If the value is zero, there was no error. The various values for the error source byte are \$89 (server), \$88 (VLM Requester), \$81 (NETX shell) or \$7F (one of the NetWare units).

```

TnwObjectStr; = String[47];
TnwPropStr;   = String[15];

```

Type for a bindery object name and for a bindery property name.

```

TnwRegisters; = record
    ...
end;

```

A registers type that is used internally by the NetWare units, especially by the vlmCall and nwIntr routines.

```
TnwServer; = Word;
```

A server handle. The routines in the NetWare units use a server handle to determine which of several servers on the network a call is destined for.

```

TnwShellType; = (nsNone,      {..none detected}
                 nsNETX,     {..NETX}
                 nsVLM);     {..VLM}

```

The types of NetWare shells that can be detected: no shell, a NETX shell, or the VLM Requester. nwShellType returns a variable of this type.

```
TnwUpperStr; = procedure (var S : String);
```

A procedural type to convert a string to uppercase. The NWBASE unit defines a global procedural variable of this type to uppercase a string, see nwUpperStr.

Variables

```
nwUpperStr; : TnwUpperStr;
```

A procedural variable that is used extensively by the NetWare units to convert a string to uppercase. The initialization routine of the NWBASE unit sets the variable to a simple uppercase routine that converts lowercase 'a'..'z' to uppercase 'A'..'Z'. If you need a different uppercase routine that maps other international characters to uppercase, then you can write a far global routine of type TnwUpperStr and set nwUpperStr equal to it.

nwIntr / nwServerCall / nwShellType

Syntax

```
function nwIntr; (Intr : Byte; var Regs : TnwRegisters) :  
  TnwErrorCode;
```

Purpose

Issue a real mode interrupt.

Description

In a real mode program, this routine issues an interrupt in the same manner as the Borland Pascal Intr routine. In a protected or Windows mode program, this routine issues a simulated real mode interrupt via a DPMI call.

nwIntr exists so that there is a single calling syntax between all the possible Pascal targets to make the rest of the NetWare units easier to write and maintain. It also enables the use of a single registers type structure.

The function result is usually zero. It is non-zero only if a DPMI error occurs.

Syntax

```
function nwServerCall; (Server : TnwServer; Func : Word; ReqLen :  
  Word;  
                        var Request; RpyLen : Word; var Reply) :  
  TnwErrorCode;
```

Purpose

Call a server via NCP with preset request and reply packets.

Description

This routine is documented so that you can easily call a server via NetWare Core Protocol (NCP) to use a NetWare routine that isn't interfaced. Server is the handle of the server that is the recipient of the call, Func is the NCP function number (in the Novell's Client API documentation, it is the value of AL), ReqLen is the number of bytes in Request, and RpyLen the number of bytes in Reply. When coding for protected mode or Windows, both the Request and Reply buffers can be in protected mode, this function internally copies them to a real mode buffer. See the source code for other details and hints on the use of nwServerCall.

Syntax

```
function nwShellType; : TnwShellType;
```

Purpose

Return the type of NetWare shell present on the workstation.

Description

The primary use of this routine is to discover whether a NetWare shell is present on the workstation. If nwShellType returns nsNone, then no shell was detected at startup.

Example

```
case nwShellType of  
  nsNone : Writeln('No NetWare shell found')  
  nsNETX : Writeln('A NETX shell was found')  
  nsVLM  : Writeln('A VLM Requester was found')  
end
```

See Also

nwShellVersion

nwShellVersion / vlmCall / vlmVersion

Syntax

```
function nwShellVersion; : Word;
```

Purpose

Return the version of NetWare shell present on the workstation.

Description

The high byte of the result is the major version number and the low byte is the minor version number. If there is no shell, the function returns zero.

Example

```
ShellVersion := nwShellVersion;
Writeln('The shell major version is ', Hi(ShellVersion),
        ', and the minor version is ', Lo(ShellVersion));
```

See Also

nwShellType

Syntax

```
function vlmCall; (DestID : Word; DestFunc : Word;
                  var Regs : TnwRegisters) : TnwErrorCode;
```

Purpose

Make a direct call to a VLM module.

Description

This routine is documented so that you can make a direct call to a VLM Requester module. DestID is the ID of the VLM module, DestFunc is the function number to call, and Regs is a TnwRegisters variable that defines the values of the registers at the time of the call. Segment fields in Regs must be real mode segment values, not protected mode selector values. See the source code for examples of how to use vlmCall.

Syntax

```
function vlmVersion; (DestID : Word) : Word;
```

Purpose

Return the version number of a VLM module.

Description

This routine is documented so that you can get the version of a particular VLM module. DestID is the ID of the VLM module. The major version number is returned in the high byte of the function result and the minor version number in the low byte. The NWBase unit source code has a complete list of VLM modules and their IDs. A small representative list follows:

VLM.EXE	\$01	the VLM manager
CONN.VLM	\$10	connection services
TRAN.VLM	\$20	transport services
REDIR.VLM	\$40	redirection services
PRINT.VLM	\$42	print services

If vlmVersion returns zero, then either the destination ID is invalid or the VLM module was not loaded.

NWCONN provides access to the workstation's connection details. A connection in NetWare terms is a link between a server and a workstation. The workstation is known to a server by means of a connection number. Connection numbers are allocated on a first-come, first-served basis when the workstation's NetWare shell is loaded. They start at 1, going up to the limit that is hard-coded into the server software. Therefore, if you have more than one server on your network, your workstation will be known by different connection numbers for each of the servers.

The workstation differentiates between the servers it is attached to by means of a handle. Under a NETX shell, the handles are numbered from 1 to 8, but under the VLM Requester, this does not apply. Thus it does not make sense to use the server handle for anything else except to identify the server concerned. Do not assume, for example, that the server you first logged on to has a numerically lower handle than the next one you log on to.

NWCONN provides functions to:

- determine the available servers on the network
- obtain the default or primary server
- get information about a server
- determine the connection number of the workstation
- determine the details for a given connection

Declarations

Constants

```
MaxNetworks; = 8;
```

The maximum number of networks returned by nwGetNetworkList.

Types

```
PnwConnList = ^TnwConnList;
TnwConnList; = record
  Count : Word;
  List : array [0..126] of Word;
end;
```

A structure describing a list of connection numbers. Used primarily by the NWMSG basic message unit.

```
TnwConnInfo; = record
  ObjectID : nwLong;           {..the logged in object's ID}
  ObjectType : Word;           {..the logged in object's type}
  ObjectName : TnwObjectStr;   {..the name of the object}
  LoginDate : TnwDate;         {..the time/date the object logged
on}
end;
```

A structure returned by nwGetConnInfo to describe the bindery object that is logged on to the connection. Usually this means a user, but other types of bindery objects can log onto connections. ObjectID is the bindery object ID, ObjectType is the bindery type of that object, ObjectName is the login name. LoginDate is the date and time the object logged into the connection (the WeekDay field is also returned by nwGetConnInfo).

```
TnwEnumServerFunc; = function (Name : TnwServerName; Server : Word;
var ExtraData) : Boolean;
```

Type of the function that nwEnumServers calls when it enumerates the available servers. Name is the server name, Server is its handle, and ExtraData is an untyped var parameter that was passed to the original call to nwEnumServers. The function should return True if the enumeration is to continue, False if it should stop.

```
TnwNetworkList; = record
  Count : Word;
  List : array [1..MaxNetworks] of nwLong;
end;
```

A structure describing a list of network numbers. The routine nwGetNetworkList returns a variable of this type. Count is the number of network numbers in the List field. List is an array of network numbers.

```
TnwServerInfo; = record
  ServerName : TnwServerName;
  NetWareVer : Byte;
  NetWareSub : Byte;
  MaxConns : Word;
  UsedConns : Word;
  MaxVols : Word;
  Revision : Byte;
  SFTLevel : Byte;
  TTSLLevel : Byte;
  PeakConn : Word;
  AccountVer : Byte;
  VAPVer : Byte;
  QueueVer : Byte;
  PrintServVer : Byte;
  VirtualVer : Byte;
  SecurityVer : Byte;
  BridgeVer : Byte;
  Reserved : array [1..60] of Byte;
end;
```

A structure returned by the function nwGetServerInfo. Most of the fields describe version numbers of the NetWare software running on the server. The most important fields are NetWareVer (the major version number), NetWareSub (the minor version number), MaxConns (the maximum number of workstation connections), and UsedConns (the number of connections in use). Under NetWare 4.x MaxConns dynamically increases as more workstations attach, up to the absolute maximum that the server software allows.

```
TnwServerName; = TnwObjectStr;
```

Type for a server name.

nwDefaultServer / nwEnumServers / nwGetConnInfo

Syntax

```
function nwDefaultServer; : TnwServer;
```

Purpose

Return the default server handle.

Description

Under the NETX shell, the default server handle is either the preferred server, the server defined by the current drive, or the primary server (the server that was originally logged into). Under the VLM Requester, the default server is the primary server.

Generally it is better to select a server handle by some other means. For example, by calling `nwEnumServers` you can find out which servers are available and let the user select one. Another method is to hard-code the server name into the application and use `nwServerFromName` to get the handle. Yet another method is to calculate the server handle from a DOS drive letter by calling `nwParseFileName` (see `NWFILE` in `_8.A.5`).

Syntax

```
procedure nwEnumServers; (EnumFunc : TnwEnumServerFunc; var  
  ExtraData);
```

Purpose

Cycle through the available servers.

Description

This procedure enumerates the available servers. For each server found, it calls a function (EnumFunc) of the `TnwEnumServerFunc` type. `ExtraData` is an untyped var parameter that is not used by `nwEnumServers`, it is just passed on as a parameter to `EnumFunc`. You can use `ExtraData` to pass information to each call of `EnumFunc`.

Under the NETX shell, the number of servers is 8 or less. Under the VLM Requester, the number of servers is configurable (see the NetWare workstation configuration manuals for details).

See Also

`TnwEnumServerFunc`

Syntax

```
function nwGetConnInfo; (Server : TnwServer; ConnNo : Word;  
  var CI : TnwConnInfo) : TnwErrorCode;
```

Purpose

Return login information for a connection number on the specified server.

Description

The information returned includes the bindery object ID, the object type and name, and the date and time it logged into the connection. If the connection number specifies an unknown connection, `nwGetConnInfo` returns with a non-zero error code and the `CI` structure is left uninitialized.

See Also

`nwGetConnNo`

`nwGetConnNoForUser`

nwGetConnNo / nwGetConnNoForUser

Syntax

```
function nwGetConnNo;(Server : TnwServer) : Word;
```

Purpose

Return the connection number for this workstation.

Description

Each workstation is given a connection number when it attaches to a server. The connection number for a single workstation will be different for different servers.

Various methods exist to determine the connection number for another workstation. If the user name is known, you can call nwGetConnNoForUser to iterate through the connection numbers. Another method is to iterate through the available connections (the UsedConns field returned by nwGetServerInfo). Connection numbers start at 1.

See Also

nwGetConnNoForUser

nwGetServerInfo

Syntax

```
function nwGetConnNoForUser;(Server : TnwServer; UserName :  
TnwObjectStr;  
var ConnNo : Word) : TnwErrorCode;
```

Purpose

Return the connection number for a user.

Description

This function is designed to operate in an iterative manner. It returns the next connection number greater than the one specified in ConnNo for the user defined by UserName. You can iterate through all the connection numbers for a particular user by calling nwGetConnNoForUser repeatedly, each time passing the ConnNo returned by the previous call. The function returns nwErrNoMoreConns (defined in NWBASE in _8.A.1) when there are no more connection numbers.

Example

```
var  
  ConnNo : Word;  
  DefServer : TnwServer;  
...  
writeln('JOANNA is present..')  
DefServer := nwDefaultServer;  
Status := 0;  
while Status = 0 do  
begin  
  Status := nwGetConnNoForUser(DefServer, 'JOANNA', ConnNo);  
  if Status = 0 then  
    writeln('..on ', ConnNo);  
end;
```

This example lists the connection numbers for the user JOANNA on the default server. Notice that the first time through the loop, ConnNo is zero. From then on, ConnNo is the value previously returned by the nwGetConnNoForUser routine.

See Also

nwGetConnNo

nwGetInternetAddress / nwGetNetworkList

Syntax

```
function nwGetInternetAddress; (Server : TnwServer; ConnNo : Word;  
                               var IA : IPXAddress) : TnwErrorCode;
```

Purpose

Return the internetwork address for the specified connection number.

Description

Use this routine to translate a known connection number (can be obtained from nwGetConnNoForUser) into an internetwork address for IPX and SPX communications. nwGetInternetAddress returns an IPXAddress structure with the network and node fields filled in. The Socket field is the socket number that the NetWare shell uses to communicate with the server and should not be used by any other application.

See Also

nwGetConnNoForUser

Syntax

```
procedure nwGetNetworkList; (Server : TnwServer; var NetList :  
                             TnwNetworkList);
```

Purpose

Return a list of the network numbers accessible on a given server.

Description

This function is useful when you need to broadcast an IPX message to all workstations that are connected to the default server. In a multi-server environment, these stations might not all share the same IPX network number. nwGetNetworkList returns a list that contains all unique network numbers of connected stations. It calls nwGetInternetAddress for all connections in use and builds the network list. The TnwNetworkList Count field is the number of unique network numbers, and the List field is an array of these numbers. If an error occurs, Count contains zero.

Example

See NISEND.PAS and NSSEND.PAS for examples of using this function.

nwGetServerInfo / nwGetServerTime / nwIsLoggedIn

Syntax

```
function nwGetServerInfo;(Server : TnwServer;  
                           var SI : TnwServerInfo) : TnwErrorCode;
```

Purpose

Return information about a server.

Description

The information returned includes the server's name, its version number, the maximum number of connections, and the number of connections currently in use. nwGetServerInfo returns zero if no error occurred.

See Also

nwGetServerTime

Syntax

```
function nwGetServerTime;(Server : TnwServer; var DT : TnwDate) :  
TnwErrorCode;
```

Purpose

Return the date and time on the given server.

Description

The TnwDate structure is defined in _8.A.1. The fields have the following ranges:

Year	1980..2079
Month	1..12
Day	1..31
Hour	0..23
Minute	0..59
Second	0..59
Weekday	nwSun..nwSat

Since workstations running under NetWare can have different internal clock settings, it is good practice to synchronize network operations using the date and time returned from a given reference server. The NetWare LOGIN program does this synchronization.

See Also

nwGetServerInfo

nwSetServerTime

Syntax

```
function nwIsLoggedIn;(Server : TnwServer) : Boolean;
```

Purpose

Return True if the calling workstation is logged into a server.

Description

If the workstation is logged into the server (or, using NetWare 4.x terminology, is authenticated) this routine returns True. If the workstation is attached to the server, but not logged in, nwIsLoggedIn returns False (as it does if an error occurs during the call).

Many of the routines in the NWXxx units work successfully only for a properly logged in workstation, but some work even if the workstation is just attached to the server.

nwServerFromName / nwServerVersion

Syntax

```
function nwServerFromName, (Name : TnwServerName) : TnwServer;
```

Purpose

Return a handle for a server name.

Description

This function provides a way to get the server handle that's required for most of the routines in the NetWare units. If the server name is not found in the available server list, nwServerFromName returns zero. The handle list is maintained by the NetWare shell.

Another method to get the server handle is to use the nwEnumServers procedure and select the server you want to use.

Yet another method is to use nwParseFileName (in the NWFILE unit in _8.A.5) if you know the mapped drive letter. Pass 'd:\' to nwParseFileName (with the d replaced by the mapped drive letter) and nwParseFileName returns the server handle.

See Also

nwEnumServers

nwParseFileName (NWFILE)

Syntax

```
function nwServerVersion; (Server : TnwServer) : Word;
```

Purpose

Return the version of the specified server.

Description

The major version number is returned in the most significant byte and the minor version number is returned in the least significant byte. If the server handle is not found (or another error occurs), nwServerVersion returns zero.

See Also

nwServerFromName

nwSetServerTime

Syntax

```
function nwSetServerTime;(Server : TnwServer; var DT : TnwDate) : _____  
TnwErrorCode;
```

Purpose

Set the date and time on the specified server.

Description

The TnwDate structure is defined in _8.A.1. The fields have the following ranges:

Year	1980..2079
Month	1..12
Day	1..31
Hour	0..23
Minute	0..59
Second	0..59
Weekday	nwSun..nwSat

The time passed in the DT record must be the local time. If the server uses UTC time (e.g. NetWare 4.x) it makes the necessary adjustments itself.

Using this routine and a time signal, a workstation can set the clocks for all the attached servers by using the nwEnumServers procedure and a properly coded enumerator function.

See Also

nwGetServerTime

NWBIND provides access to the server bindery. The bindery is the secure database where the server stores various information about users, groups, queues, other servers, and so on. Under NetWare 4.x, the bindery doesn't exist in the form that users of NetWare 2.x and 3.x know, but instead is emulated by the server software from Directory Services information. NWBIND provides functions to:

- add and modify details about bindery objects
- list bindery objects
- add, modify, or delete properties for bindery objects

This section does not provide a full education in how to access and use the bindery. However, it might be beneficial to define a few terms that are used extensively in this unit.

The bindery consists of a set of objects (these are not OOP-style objects). Each object has a unique 4 byte identifier (in B-Tree Filer terms, this is a primary key). Also, as an alternative, each object can be referenced by its name and its type. The name and type taken together must be unique. The objects can be anything, but the most common ones are users, groups, other servers, and queues.

To further define each object, you can specify properties. These can be viewed as "fields" of the object. Each property has a name and a value. Within each object, each property must have a unique name. For a user object, examples of properties are 'LOGIN_CONTROL' and 'IDENTIFICATION'.

The value of a property is like a variable length record with a section length of 128 bytes. The property value can either be totally user-defined or it can be a "set." A set property consists of a list of object IDs. For example, in a NetWare 3.x bindery, there is generally a group object called EVERYONE. It has a property called GROUP_MEMBERS which is a set property that contains all the user object IDs.

Properties that are not sets can have any structure. The implementer defines the property's layout. The layouts for the well-known Novell properties are not shown here. See the Novell NetWare Client API for more details.

Although the NWBIND unit enables you to create and delete objects such as users, be aware that creating a new user object does not automatically create all the required properties like the Novell programs SYSCON or NWADMIN do. It is entirely your responsibility to create all the default properties and their values and also the links to other objects (like the group EVERYONE). Similarly when you delete an object, NetWare does not automatically delete the object ID from set properties in other objects, for example.

See the demonstration program BINDLIST for an example of how to use the NWBIND unit. It lists all the objects and their properties in the bindery.

Declarations

Constants

```
nwbAnyOne      = $00; {access allowed to all clients}
nwbLogged      = $01; {access allowed to logged in clients}
nwbObject      = $02; {access allowed to the object itself}
nwbSupervisor  = $03; {access allowed to the supervisor}
nwbNetWare     = $04; {access allowed only to the NetWare operating
system}
```

Security flags. The security byte that is returned by routines such as `nwbScanObject` consists of two nibbles (4 bits each), the upper nibble refers to write access and the lower nibble to read access. Each nibble takes one of the 5 values above. Hence a security byte of \$32 means that the object (i.e., generally the user) can read the information, but only the supervisor can alter it.

```
nwbErrServerOutOfMem = $8996; {server out of memory}
nwbErrMemberExists   = $89E9; {object already exists as member in
set}
nwbErrNotMember      = $89EA; {object does not exist as member in
set}
nwbErrNotSetProperty = $89EB; {property is not a set}
nwbErrNoSuchSegment  = $89EC; {segment number does not exist}
nwbErrPropExists     = $89ED; {property already exists}
nwbErrObjExists      = $89EE; {object already exists}
nwbErrInvName        = $89EF; {name contains invalid characters}
nwbErrWildcardBanned = $89F0; {no wildcards allowed for this
call}
nwbErrInvSecurity    = $89F1; {invalid bindery security}
nwbErrNoObjRenamePriv = $89F3; {user has no object rename
privileges}
nwbErrNoObjDeletePriv = $89F4; {user has no object delete
privileges}
nwbErrNoObjCreatePriv = $89F5; {user has no object create
privileges}
nwbErrNoPropDeletePriv = $89F6; {user has no property delete
privileges}
nwbErrNoPropCreatePriv = $89F7; {user has no property create
privileges}
nwbErrNoPropWritePriv = $89F8; {user has no property write
privileges}
nwbErrNoPropReadPriv  = $89F9; {user has no property read
privileges}
nwbErrNoSuchProperty  = $89FB; {given property does not exist}
nwbErrNoSuchObject    = $89FC; {given object does not exist}
nwbErrBinderyLocked   = $89FE; {the bindery is locked}
nwbErrBinderyFailure  = $89FF; {the bindery has failed}
```

Error codes that can be returned by the server when using bindery services. This list is the one provided by Novell. It is not guaranteed to be exhaustive.

```
nwboUnknown      = $0000;
nwboUser         = $0001;
nwboGroup        = $0002;
nwboPrintQueue   = $0003;
nwboFileServer   = $0004;
nwboJobServer    = $0005;
nwboGateway      = $0006;
nwboPrintServer  = $0007;
nwboArchiveQueue = $0008;
nwboArchiveServer = $0009;
nwboJobQueue     = $000A;
nwboAdministration = $000B;
```

```

nwboNASSNAGateway    = $0021;
nwboRemoteBridge     = $0026;
nwboRemBridgeServer  = $0027;
nwboTimeSyncServer   = $002D;
nwboArchiveServerSAP = $002E;
nwboAdvertisingPrint = $0047;
nwboBtrieveVAP       = $0050;
nwboPrintQueueUser   = $0051;
nwboWild              = $FFFF;

```

Types of well-known bindery objects. nwboWild is used for wildcard scanning; see the example for nwScanObject later in this section.

Types

```
TnwPasswordStr; = String[127];
```

A type that defines a password for an object.

```

TnwPropValue; = record
  case Boolean of
    True  : (pvItem : Array[1..128] of Char);
    False : (pvSet   : Array[1..32] of nwLong);
  end;

```

A structure that defines a property segment value for an object in the bindery. A property for an object consists of one or more of these segments. If the property is a set, then it is assumed that it consists of an array of bindery object IDs in blocks of 32. If the property is an item, then it is assumed to consist of application-defined fields in 128 byte segments. For the layout of well-known properties, see the Novell Client API documentation.

nwbAddObjectToSet / nwbChangeObjectSecurity

Syntax

```
function nwbAddObjectToSet;(Server : TnwServer; ObjType : Word;
                           ObjName : TnwObjectStr; PropName:
                           TnwPropStr;
                           MemberObjType : Word;
                           MemberObjName : TnwObjectStr) :
                           TnwErrorCode;
```

Purpose

Add a bindery object to a set property of a bindery object.

Description

The object uniquely defined by MemberObjName and MemberObjType is added to the PropName property owned by the object uniquely defined by ObjName and ObjType. Note that all objects must reside in the same bindery; you cannot cross reference an object from the bindery on one server to the bindery on another server.

The user must have sufficient write access rights to the object whose property is being added to, as well as to the property itself.

See Also

nwbDeleteObjectFromSet

Syntax

```
function nwbChangeObjectSecurity;(Server : TnwServer; ObjType : Word;
                                   ObjName : TnwObjectStr;
                                   NewSecurity : Byte) : TnwErrorCode;
```

Purpose

Change the security flag for the specified bindery object.

Description

The NewSecurity byte consists of two nibbles, the upper one for write access, the lower one for read access. The different nibble values are defined by the constants nwbAnyOne..nwbNetWare (described earlier in this section). The security byte of the bindery object defined by ObjName and ObjType is altered to NewSecurity by this call, providing the user has sufficient write access authority.

See Also

nwbChangePropertySecurity

nwbChangePassword / nwbChangePropertySecurity

Syntax

```
function nwbChangePassword; (Server : TnwServer; ObjType : Word;  
                             ObjName : TnwObjectStr; OldPassword,  
                             NewPassword : TnwPasswordStr) :  
    TnwErrorCode;
```

Purpose

Change the password of a bindery object.

Description

This routine does not work for servers that use encrypted passwords. If the OldPassword matches, then the password for the object uniquely defined by ObjName and ObjType is changed to NewPassword.

See Also

nwbVerifyPassword

Syntax

```
function nwbChangePropertySecurity; (Server : TnwServer; ObjType :  
Word;  
                                     ObjName : TnwObjectStr; PropName :  
TnwPropStr;  
                                     NewPropSecurity : Byte) :  
    TnwErrorCode;
```

Purpose

Change the security flag for a bindery object's property.

Description

The NewPropSecurity byte consists of two nibbles, the upper one for write access, the lower one for read access. The nibble values are defined by the constants nwbAnyOne.. nwbNetWare (described earlier in this section). The security byte of property PropName for the bindery object defined by ObjName and ObjType is altered to NewPropSecurity by this call, providing the user has sufficient write access authority.

See Also

nwbChangeObjectSecurity

nwbCloseBindery / nwbCreateObject

Syntax

```
function nwbCloseBindery;(Server : TnwServer) : TnwErrorCode; _____
```

Purpose

Close the bindery for backup purposes.

Description

This call is ignored under NetWare 4.x (the bindery is emulated - there are no bindery files). For NetWare 2.x and 3.x, use this call only if you need to backup the bindery files (for their names see the NetWare documentation). The server is effectively crippled while the bindery is closed, so reopen the bindery as soon as possible.

See Also

nwbOpenBindery

Syntax

```
function nwbCreateObject;(Server : TnwServer; ObjType : Word;  
                           ObjName : TnwObjectStr; ObjIsDynamic :  
Boolean;  
                           ObjSecurity : Byte) : TnwErrorCode;
```

Purpose

Create a new bindery object.

Description

If you use this routine to create a 'well-known' object, such as a user, then it is entirely up to you to create and initialize the properties for that object so that the normal NetWare utilities can access it. Therefore, it is recommended to create only your own object types, where you can have full control over property contents.

The function creates a bindery object called ObjName, of type ObjType, with security byte ObjSecurity (see nwbChangeObjectSecurity). If ObjIsDynamic is True, the object is temporary and is deleted the next time the server is started. If ObjIsDynamic is False, the object is permanent.

To get the object ID of the newly created object, call nwbGetObjectID.

See Also

nwbChangeObjectSecurity
nwbGetObjectID

nwbDeleteObject

nwbCreateProperty / nwbDeleteObject

Syntax

```
function nwbCreateProperty; (Server : TnwServer; ObjType : Word;  
                             ObjName : TnwObjectStr; PropName :  
                             TnwPropStr;  
                             PropIsDynamic, PropIsSet : Boolean;  
                             PropSecurity : Byte) : TnwErrorCode;
```

Purpose

Create a new property for a bindery object.

Description

This routine creates a property called PropName for the bindery object given by ObjName and ObjType. If PropIsDynamic is True, the property is temporary and is deleted the next time the server is started. If PropIsDynamic is False, the property is permanent. If PropIsSet is True, the property is assumed to be a set (i.e., a list of bindery object IDs), otherwise the property structure depends on the application accessing it. The PropSecurity byte consists of two nibbles, the upper one for write access, the lower one for read access. The nibble values are defined by the constants nwbAnyOne..nwbNetWare. The user must have sufficient write access rights to the object. To add data to the property, use nwbWritePropertyValue or nwbAddObjectToSet.

See Also

nwbAddObjectToSet
nwbWritePropertyValue

nwbDeleteProperty

Syntax

```
function nwbDeleteObject; (Server : TnwServer; ObjType : Word;  
                           ObjName : TnwObjectStr) : TnwErrorCode;
```

Purpose

Delete a bindery object.

Description

This routine deletes the bindery object defined by ObjName and ObjType from the bindery. All of the object's properties are also deleted. The user must have sufficient write access rights to the object.

See Also

nwbCreateObject

nwbDeleteObjectFromSet / nwbDeleteProperty

Syntax

```
function nwbDeleteObjectFromSet;(Server : TnwServer; ObjType : Word,  
                                ObjName : TnwObjectStr; PropName:  
TnwPropStr;  
                                MemberObjType : Word;  
                                MemberObjName : TnwObjectStr) :  
TnwErrorCode;
```

Purpose

Delete a bindery object from a set property of a bindery object.

Description

The object defined by MemberObjName and MemberObjType is deleted from the PropName property of the object defined by ObjName and ObjType. The user must have sufficient write access rights to the object and its property.

See Also

nwbAddObjectToSet

Syntax

```
function nwbDeleteProperty;(Server : TnwServer; ObjType : Word;  
                             ObjName : TnwObjectStr;  
                             PropName: TnwPropStr) : TnwErrorCode;
```

Purpose

Delete a property.

Description

This routine deletes the property PropName (and all its data) from the object defined by ObjName and ObjType. The user must have sufficient write access rights to the object and its property.

See Also

nwbCreateProperty

nwbGetBinderyAccessLevel / nwbGetObjectID

Syntax

```
function nwbGetBinderyAccessLevel; (Server : TnwServer; var  
  AccessLevel : Byte;  
                                     var ObjID : nwLong) : TnwErrorCode;
```

Purpose

Return the workstation's access level to the bindery.

Description

The AccessLevel returned is a standard security byte: the high nibble specifies the user's write privileges, the low nibble specifies the read privileges. The ObjID returned is the logged on user's bindery object ID.

Syntax

```
function nwbGetObjectID; (Server : TnwServer; ObjType : Word;  
                          ObjName : TnwObjectStr;  
                          var ObjID : nwLong) : TnwErrorCode;
```

Purpose

Return the bindery object ID for the specified object.

Description

This function returns the bindery object ID for the object defined by ObjName and ObjType.

See Also

nwbGetObjectName

nwbGetObjectNames / nwbIsObjectInSet

Syntax

```
function nwbGetObjectNames; (Server : TnwServer; ObjID : nwLong; var  
ObjType : Word;  
var ObjName : TnwObjectStr) : TnwErrorCode;
```

Purpose

Return the name and type of the specified bindery object.

Description

This routine returns the name and bindery object type for a given bindery object ID.

See Also

nwbGetObjectID

Syntax

```
function nwbIsObjectInSet; (Server : TnwServer; ObjType : Word;  
ObjName : TnwObjectStr; PropName :  
TnwPropStr;  
MemberObjType : Word;  
MemberObjName : TnwObjectStr) :  
TnwErrorCode;
```

Purpose

Return True if a bindery object is present in a set property.

Description

This function returns zero if the bindery object defined by MemberObjName and MemberObjType exists in the set property PropName for the object defined by ObjName and ObjType. The user must have sufficient read access rights to the object and its property. If the object is not in the set, the return code is nwbErrNotMember.

See Also

nwbAddObjectToSet

nwbDeleteObjectFromSet

nwbOpenBindery / nwbReadPropertyValue

Syntax

```
function nwbOpenBindery, (Server : TnwServer) : TnwErrorCode;
```

Purpose

Open the bindery.

Description

This call is the complement of nwbCloseBindery. This call is ignored under NetWare 4.x (the bindery is emulated).

See Also

nwbCloseBindery

Syntax

```
function nwbReadPropertyValue; (Server : TnwServer; ObjType : Word;
                                ObjName : TnwObjectStr; PropName :
                                TnwPropStr;
                                SegmentNumber: Byte; var PropValue :
                                TnwPropValue;
                                var PropIsDynamic: Boolean;
                                var PropIsSet : Boolean;
                                var MoreSegments : Boolean) :
                                TnwErrorCode;
```

Purpose

Read the value of a property for a bindery object.

Description

Property values consist of one or more 128 byte segments, each segment numbered consecutively from 1. This function reads the SegmentNumber part of property PropName for the bindery object defined by ObjName and ObjType. Other fields returned are PropIsDynamic and PropIsSet (see nwbCreateProperty for their definition) and also a flag to define whether there are any more segments after this one. nwbReadPropertyValue returns zero if it is successful, otherwise a non-zero error code. The user must have sufficient read access rights to the object and its property.

See Also

nwbCreateProperty

nwbWritePropertyValue

nwbRenameObject / nwbScanObject

Syntax

```
function nwbRenameObject;(Server : TnwServer; ObjType : Word; _____  
                           OldObjName, NewObjName : TnwObjectStr) :  
TnwErrorCode;
```

Purpose

Rename a bindery object.

Description

This routine changes the name of a bindery object from OldObjName to NewObjName. The user must have sufficient write access rights to the object.

See Also

nwbCreateObject

Syntax

```
function nwbScanObject;(Server : TnwServer; var ObjType : Word;  
                           var ObjName : TnwObjectStr; var ObjID :  
nwLong;  
                           var ObjIsDynamic : Boolean; var ObjSecurity :  
Byte;  
                           var HasProperties : Boolean) : TnwErrorCode;
```

Purpose

Sequentially scan through the objects in a bindery.

Description

This function enables you to move through a bindery, retrieving information about objects found there. It is equivalent to the FindFirst/FindNext routines in the DOS unit.

The scan is started by specifying ObjType, either as a specific bindery object type or as the wildcard object type nwboWild, and ObjName, either as a specific name or as a string with a '*' wildcard at the end. Set ObjID to -1. Call nwbScanObject and it returns the first object in the bindery that matches your search criteria. Leave the ObjID alone, and set up the ObjType and ObjName fields as before and call nwbScanObject again. It returns with the next object in sequence.

nwbScanObject returns the type, name, and object ID for the bindery object. For the definitions of ObjIsDynamic and ObjSecurity, see nwbCreateObject. The boolean HasProperties is True if the object has any properties.

Example

```
ObjID := -1;  
Status := 0;  
while Status = 0 do  
begin  
  ObjType := nwboWild;  
  ObjName := '*';  
  Status := nwbScanObject(Server, ObjType, ObjName, ObjID,  
                           ObjDyn, ObjSec, HasProps);  
  if Status = 0 then  
  { do something with this object }  
end;
```

This example cycles through all the objects of all types in the bindery. There are two wildcards in action here: the object type is set to nwboWild which means 'match any object type', and the object name is set to '*' which means 'match any name'. To cycle through all the users instead, set ObjType to nwboUser. To cycle through all users beginning with 'J', set ObjType to nwboUser and the ObjName variable to 'J*'.

nwbRenameObject / nwbScanObject

For other examples see the BINDLIST demonstration program and the implementation of the nwEnumQueues routine in the NWPRINT unit (in 8.A.8).

See Also

nwbCreateObject

nwbScanProperty / nwbVerifyPassword

Syntax

```
function nwbScanProperty; (Server : TnwServer; ObjType : Word; _____  
                           ObjName : TnwObjectStr; var Sequence :  
nwLong;  
                           var PropName : TnwPropStr; var PropIsDynamic  
: Boolean;  
                           var PropIsSet : Boolean; var PropSecurity :  
Byte;  
                           var HasValue : Boolean;  
                           var MoreProps : Boolean) : TnwErrorCode;
```

Purpose

Sequentially scan through the properties for a bindery object.

Description

Use this function to move through the property list for a given bindery object, retrieving information about the individual properties found there. It is equivalent to the FindFirst/FindNext routines in the DOS unit.

The scan is started by specifying the Sequence field as -1. Call nwbScanProperty and it returns the first property for the object. Leave the Sequence field alone, call nwbScanProperty again, and it returns with the next property in sequence. When the MoreProps parameter is False, there are no more properties to scan.

nwbScanProperty returns the name of the property (PropName), whether it is dynamic or not (PropIsDynamic), and whether it is a set or not (PropIsSet). The PropSecurity byte consists of two nibbles, the upper one for write access, the lower one for read access. The nibble values are defined by the constants nwbAnyOne..nwbNetWare (described earlier in this section). HasValue is True if the property has a value.

Warning: under NetWare 4.x, the PropIsDynamic and PropIsSet boolean values do not seem to be returned properly by the bindery emulation - the nwbReadPropertyValue routine however does return their correct values.

See Also

nwbCreateProperty

Syntax

```
function nwbVerifyPassword; (Server : TnwServer; ObjType : Word;  
                           ObjName : TnwObjectStr;  
                           Password: TnwPasswordStr) : TnwErrorCode;
```

Purpose

Verify a bindery object password.

Description

This function does not work with servers that use encrypted passwords. Generally this routine can be used to verify user passwords.

See Also

nwbChangePassword

nwbWritePropertyValue

Syntax

```
function nwbWritePropertyValue, (Server : TnwServer; ObjType : Word;  
                                ObjName : TnwObjectStr; PropName:  
    TnwPropStr;  
                                SegmentNumber : Byte;  
                                var PropValue : TnwPropValue;  
                                EraseRemainingSegments : Boolean) :  
    TnwErrorCode;
```

Purpose

Write a segment to a property for a bindery object.

Description

This routine should be used to update properties that are not sets. For a property that is a set, use `nwbAddObjectToSet`.

The property to be written to is `PropName`, and is owned by the bindery object defined by `ObjName` and `ObjType`. The number of the segment to write is `SegmentNumber` and its actual value is passed in `PropValue`. Novell's recommendation is to write the property segments in order, starting at 1. When you write the last segment of your property value, set the `EraseRemainingSegments` parameter to `True` and the server's bindery software closes off the property value at that point.

See Also

`nwbAddObjectToSet`

`nwbCreateProperty`

NWMSG provides access to NetWare's broadcast message facilities. The routines provided by NWMSG are fairly primitive and depend on the servers present on the network. Basically, it allows you to send and receive a message between two workstations.

NWMSG provides routines to:

- set and get the workstation's broadcast mode
- get a saved broadcast message
- send a broadcast message to other connections
- send a message to the server console

Before you can receive messages, you must instruct the NetWare shell to save them for you (the default is for the shell to display all messages immediately). This is done by setting the workstation's broadcast mode (see `nwSetBroadcastMode` later in this section). If you select to save all messages, each server you are connected to will save one message for you.

Sending messages is simple to do. You specify a list of connection numbers to receive the message and the message itself, and then call the send message function.

Some of the problems that can be encountered when using broadcast messages are:

- The messages are not guaranteed to arrive.
- If you do not read a saved message, and another arrives for your workstation, the later message is discarded with no warning to either you or the sender.
- For a NetWare 2.x or 3.1x server, the maximum message length is 58 characters, and the maximum number of workstations you can sent it to is 254.
- For a NetWare 4.x server, the maximum message length is 254 characters, and the maximum number of workstations you can sent it to is 62.

For more robust NetWare inter-workstation communications, please refer to "NWIPXSPX: Advanced Message Services" in `_8.A.9`.

Declarations

Types

```
TnwBroadcastMode; = (bmDisplayBoth,    {display both server and user
messages}
                    bmDisplayServer, {display only server messages}
                    bmStoreServer,    {store only server messages}
                    bmStoreBoth);     {store both server and user
messages}
```

The workstation broadcast modes used to tell the NetWare how to handle received messages. Use `bmDisplayBoth` to display messages received from a server and from other users. Use `bmDisplayServer` to display messages received from a server and discard messages from other users. Use `bmStoreServer` save messages received from a server for later retrieval and discard messages from other users. Use `bmStoreBoth` to save messages received from a server and save messages from other users. Note that a server only saves one message at a time for a workstation. If a new message arrives, the older one is overwritten.

nwGetBroadcastMessage / nwGetBroadcastMode

Syntax

```
function nwGetBroadcastMessage; (Server : TnwServer;  
                                var Message : String) : TnwErrorCode;
```

Purpose

Return a broadcast message.

Description

If the workstation's broadcast mode is set to one of the 'store' options, this routine retrieves a stored message. If there is no message, then the returned string is empty.

See Also

nwSendBroadcastMessage nwSetBroadcastMode

Syntax

```
function nwGetBroadcastMode; (Server : TnwServer;  
                              var Mode : TnwBroadcastMode) :  
TnwErrorCode;
```

Purpose

Return the workstation's current broadcast mode.

Description

Each workstation has its own broadcast mode. Currently there is only one broadcast mode per workstation, however this might change with future versions of the VLM Requester.

See Also

nwSetBroadcastMode

nwSendBroadcastMessage / nwSendMessageToConsole

Syntax

```
function nwSendBroadcastMessage;(Server : TnwServer; Message : String;  
                                var ToList : TnwConnList) :  
TnwErrorCode;
```

Purpose

Send a message to a list of connections.

Description

The message length is dependent on the version of NetWare. Version 3.20 and higher allow up to 254 characters, but earlier versions allow only up to 58 characters. The maximum number of connection numbers in the list also varies with server version. For NetWare 3.20 and higher, the maximum number of connections that a message can be sent to is 62. For earlier versions of NetWare, the maximum is 254 connections.

The message is not guaranteed to arrive (it depends on many things, including whether the receiving workstation has a relevant broadcast mode). If receipt of a message is an issue, use IPX or SPX communications for message transmission (see NWIPXSPX in _8.A.9).

See Also

nwGetBroadcastMessage

Syntax

```
function nwSendMessageToConsole;(Server : TnwServer;  
                                Message : String) : TnwErrorCode;
```

Purpose

Send a message to the file server console.

Description

This routine sends a message of at most 55 characters. The message is seen immediately if the server is acting in console mode. If not, the message is stored until the next time you enter console mode.

See Also

nwSendBroadcastMessage

nwSetBroadcastMode

Syntax

```
function nwSetBroadcastMode, (Server : TnwServer,  
                             Mode : TnwBroadcastMode) : TnwErrorCode;
```

Purpose

Set the workstation's broadcast mode.

Description

If you alter the broadcast mode for an application, it is advisable to save the state of the broadcast mode prior to changing it and to restore it to the original value before terminating the program.

See Also

nwGetBroadcastMode

NWFILE provides some access to NetWare's file and directory services. Generally this unit should be viewed as an adjunct to some of the other NetWare units. It was not designed to provide a plethora of NetWare file and directory routines because it is far easier to use the corresponding DOS routines. NWFILE provides routines to:

- parse a DOS, NetWare, or Universal Naming Convention (UNC) filename into a NetWare server, volume, and path
- set and get file attributes
- lock and unlock file regions directly

Declarations

Constants

```
nweaSearchMode      = $07;
nweaTransactional   = $10;
nweaIndexed         = $20;
nweaReadAudit       = $40;
nweaWriteAudit      = $80;
```

NetWare's extended file attributes. The main one to be aware of is the transactional bit. A file that has this bit set in its extended attribute byte will be tracked by NetWare's Transaction Tracking Services (TTS). See the NWTTS unit in _8.A.7 for more information.

```
nwfaExecuteOnly     = $08;
nwfaShareable       = $80;
```

NetWare's extra file attributes. These attributes are returned with the normal DOS file attributes (for these bit values, see the Borland Pascal or DOS documentation). nwfaExecuteOnly defines the file as being an executable program (an EXE or COM file). Once this attribute is set it cannot be unset. Note that it is a redefinition of the Volume Label bit for DOS files on local drives. nwfaShareable defines the file as being shareable for those programs who do not open the file in a shareable mode. You should not use this bit; its use was implemented by Novell in the days before DOS 3.x and file sharing appeared. B-Tree Filer, for example, shares the files in its fileblocks without using this bit.

```
nwfErrUnknownServer = $7F21; {Server name not found}
nwfErrUnknownVolume = $7F22; {Volume name not found}
nwfErrNotOnServer   = $7F23; {Path is not on a server}
nwfErrNoFileName    = $7F23; {Filename missing}
nwfErrUNCTooShort   = $7F31; {UNC: filename < 7 characters}
nwfErrUNCBadStart   = $7F32; {UNC: filename didn't start with
'\\'}
nwfErrUNCBadServer  = $7F33; {UNC: server name < 2 chars}
nwfErrUNCBadVolume  = $7F34; {UNC: volume name < 2 chars}
nwfErrUNCBadRoot    = $7F35; {UNC: \\ after volume name}
nwfErrNWBadServer   = $7F41; {NW: unknown server name}
nwfErrNWBadVolume   = $7F42; {NW: unknown volume name}
nwfErrDOSBadDrive   = $7F51; {DOS: bad drive letter}
```

Error codes for the NWFILE unit.

Types

```
TnwFileHandle; = array [0..2] of Word;
```

A NetWare file handle, used internally.

```
TnwVolumeName; = String[17];
```

A NetWare volume name, including the terminating colon (:).

nwGetFileAttr

Syntax

```
function nwGetFileAttr; (FileName : String; var FAttr : Byte;
                        var ExtFAttr : Byte) : TnwErrorCode;
```

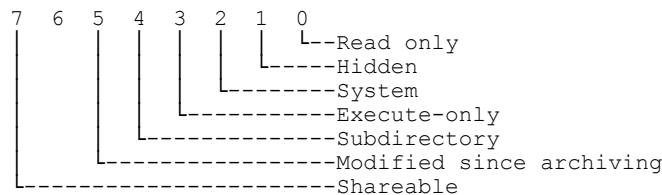
Purpose

Return the normal and extended file attributes for a file.

Description

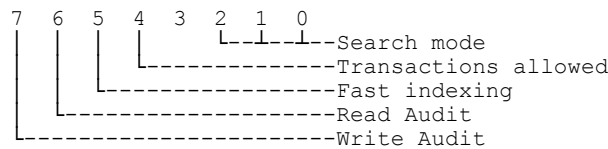
The file name can be specified in either DOS, NetWare, or UNC format. This function extracts the server and volume from the name by using `nwParseFileName`. If the file is on a NetWare server, both the DOS attributes and the NetWare attributes for the file are returned.

FAttr contains the DOS-style attributes. This bit-mapped byte is interpreted as follows:



The bits have their normal DOS interpretation, with the exception of the Execute-Only and Shareable flags, which are Novell specific. See the descriptions of `nwfaExecuteOnly` and `nwfaShareable` earlier in this section.

The extended file attributes parameter, ExtFAttr, is a bit-mapped byte, interpreted as follows:



You can use the defines `nweaSearchMode`..`nweaWriteAudit` described earlier in this section to access these bits. The search mode bits are relevant only when Path specifies an executable file, such as an EXE or COM file. The value of these bits determines whether and how the executable file searches for its data files. Normally, no search is done and the data file must be found in the current directory or in another directory explicitly named by the application. When searching is enabled, NetWare automatically looks in the default directory first, then on all search drives (NetWare's equivalent of the DOS PATH). The search occurs transparently when the application makes a call to open the file.

Together, the three search mode bits can represent values from 0 to 7, interpreted as follows:

- 0 No search instructions. The executable file uses the search method specified in NetWare's NET.CFG file. This mode is the default for all executable files.
- 1 If a complete pathname is specified by the executable file, only that path is searched. If only a filename is specified, NetWare searches the current directory followed by all search drives.
- 2 Only the current directory is searched.
- 3 If a complete pathname is specified by the executable file, only that path is searched. If only a filename is specified and the file is opened Read-Only, NetWare searches the current directory followed by all search drives.
- 4 Reserved.
- 5 NetWare searches the default directory followed by all search drives, whether or not a path is specified.
- 6 Reserved.

nwGetFileAttr

⁷ If the file is opened Read-Only, NetWare searches the default directory followed by all search drives, whether or not a path is specified.

When the transactional bit is set, the file allows support of transaction tracking (TTS). When the fast indexing bit is set, NetWare keeps an index of all blocks in the file to improve the speed of random access. This bit should be set for frequently accessed files larger than 2 megabytes. (Recent versions of NetWare automatically index large files regardless of this bit.) Read and write audit bits indicate that accesses to the file will be recorded in an audit file. This feature is not yet implemented in NetWare.

See Also

nwParseFileName

nwSetFileAttr

nwLockRecord

Syntax

```
function nwLockRecord(Handle : Word; Start, Len : nwLong;  
                      Timeout : Word) : TnwErrorCode;
```

Purpose

Lock a region of a file.

Description

This routine makes a NetWare-specific call to lock part of a file. Handle is the DOS file handle of the already opened file, Start is the starting byte (zero based) of the region, and Len is the number of bytes to lock. Timeout is the length of time in clock ticks (18.2 ticks per second) to try to lock the file region. nwLockRecord returns when either the region is locked, or when Timeout ticks expire. A result of zero means that the lock was successful, a non-zero result means the lock attempt was unsuccessful.

Note: although NetWare allows you to lock the same region (or part of a region) more than once, it is *not* a good idea to do so. Overlapping regions can only be unlocked once.

See Also

nwUnlockRecord

nwParseFileName

Syntax

```
function nwParseFileName, (FileName : String, var Server : TnwServer,  
    var ServerName : TnwServerName;  
    var VolumeName : TnwVolumeName;  
    var Path : String) : TnwErrorCode;
```

Purpose

Parse a file name into a NetWare server, volume, and full path.

Description

This routine takes a file name in DOS, NetWare, or UNC format and attempts to extract the server handle and name, volume name, and remaining path from it. The input filename does not have to be in uppercase, as the routine internally converts it to uppercase.

A UNC format filename is in the form '\\SERVER\VOLUME\PATH' (i.e., it begins with a double backslash). This routine extracts the server name and the volume name from the string and checks that they exist. The remaining path is returned without any checking on its validity.

A NetWare file name is either in the form 'SERVER\VOLUME:PATH' or in the form 'VOLUME:PATH' where the server is assumed to be the default server. The routine extracts the server name, if present, and the volume name, and checks that they exist. The remaining path is returned without any checking on its validity.

Any other file name format is assumed to be a DOS style file name, with or without a drive character and directory component. If the filename has a drive component (i.e., it starts with 'd:' where d is a alphabetic character) this is extracted; if not, the current drive is determined. The drive letter is then checked for a NetWare drive mapping. If the drive is a mapped NetWare volume, the server and volume names are determined and the remaining path is calculated from the fake root and current directory. If the drive is not a NetWare mapping, the drive letter is extracted and returned in the volume parameter and the remaining path is calculated from the current directory on that drive.

The function result is zero if the input file name was parsed correctly, otherwise it is an error code. If successful, there are two cases to consider: either the file name referred to a path on a NetWare server, or it referred to a local drive. In the first case, on return Server is the server handle, ServerName is its name, VolumeName is the NetWare volume terminated by a colon, and Path is the remaining path from the input file name and current directory/fake root. Path does not start with a backslash. You can create a full NetWare path with the following code:

```
NetWarePath := ServerName + '\' + VolumeName + Path;
```

In the second case (i.e., the input file name is on a local drive), the returned value of Server is zero, ServerName is an empty string, VolumeName is the drive letter plus a colon, and Path is the remaining path from the root of that drive. Path does not start with a backslash. You can create a full DOS path with the following code:

```
DOSPath := VolumeName + '\' + Path;
```

To find out which server and volume a DOS drive is mapped to, call nwParseFileName with the string 'd:\' where 'd' is the drive letter. The demonstration program NETINFO uses this technique.

Examples

The following examples assume that the default server is MAINSERVER, drive P: is mapped to 'MAINSERVER\SYS:', and drive R: is "root" mapped to 'MAINSERVER\SYS:PUBLIC'.

Calling nwParseFileName with these filename strings results in the following values of ServerName, VolumeName, and Path.

```
FileName:    C:\  
ServerName:  <null string>  
VolumeName:  C:  
Path:        <null string>
```

Drive C: is assumed to be a local drive.

nwParseFileName

```
FileName:      P:\
ServerName:    MAINSERVER
VolumeName:    SYS:
Path:          <null string>

FileName:      R:\
ServerName:    MAINSERVER
VolumeName:    SYS:
Path:          PUBLIC

FileName:      P:\PUBLIC\FILENAME.EXT
ServerName:    MAINSERVER
VolumeName:    SYS:
Path:          PUBLIC\FILENAME.EXT

FileName:      R:\FILENAME.EXT
ServerName:    MAINSERVER
VolumeName:    SYS:
Path:          PUBLIC\FILENAME.EXT

FileName:      R:FILENAME.EXT
ServerName:    MAINSERVER
VolumeName:    SYS:
Path:          PUBLIC<current directory>\FILENAME.EXT
```

<current directory> refers to the current directory on drive R:.

nwSetFileAttr / nwUNCtoNetWare / nwUnlockRecord

Syntax

```
function nwSetFileAttr (FileName : String; FAttr : Byte;  
                        ExtFAttr : Byte) : TnwErrorCode;
```

Purpose

Set the normal and extended file attributes for a file.

Description

See nwGetFileAttr for details on the possible attributes.

See Also

nwGetFileAttr

Syntax

```
function nwUNCtoNetWare (UNC : String; var NW : String) :  
TnwErrorCode;
```

Purpose

Convert a UNC filename to a NetWare filename.

Description

This routine is used internally to convert a UNC style filename (\\SERVER\VOLUME\PATH) to a NetWare style pathname (SERVER\VOLUME:PATH). No checking is done on the validity of the server name or volume name. The only checking done is to verify the basic format (i.e., the string starts with a double backslash, and so on).

See Also

nwParseFileName

Syntax

```
function nwUnlockRecord (Handle : Word; Start, Len : nwLong) :  
TnwErrorCode;
```

Purpose

Unlock a region of a file.

Description

This routine makes a NetWare-specific call to unlock part of a file that was locked with nwLockRecord. Handle is the DOS file handle of the already opened file. The values of Start and Len must agree with those passed to the original call to nwLockRecord. nwUnlockRecord returns zero if the lock was released, otherwise it returns an error code.

See Also

nwLockRecord

NWSEMA provides complete access to NetWare's semaphore services. Semaphores provide a flexible but simple service that is useful in many multi-user applications. A semaphore is a signed byte value that is stored in server memory and can be accessed and changed by all workstations.

The traditional use for semaphores is as a synchronization or locking mechanism. For example, if the value of the semaphore is greater than or equal to zero, a particular resource can safely be accessed by a workstation. To reserve the resource, the station decrements the semaphore and assures that the semaphore is still non-negative. When it's done using the resource, it increments the semaphore value again. By setting the semaphore's initial value, the application can control the number of stations that have simultaneous access to the resource.

Semaphores are not limited to this single purpose, however. It's also possible to use them as shared variables, subject to the constraint that the semaphore can conveniently hold values ranging only from 0 to 127. The FBDEMO bonus program supplied with B-Tree Filer uses a semaphore in this manner to signal that a station has modified an index. The other stations detect the changed semaphore value and refresh their browser screen. The use of a semaphore significantly reduces server and network loading compared to reading and writing a shared file for the same purpose.

A semaphore is described by three main properties:

- an ASCII name up to 127 characters long. Case is not important. Each station that accesses the semaphore must know this name. A station opens a semaphore by specifying the name. It obtains a four byte handle that efficiently refers to the semaphore thereafter.
- an open count. This tracks the number of stations that currently have access to the semaphore. The open count can also be used as a resource restriction mechanism.
- a value. The value of a semaphore can range from -127 to +127. Functions are provided to set an initial value when the first station opens the semaphore, and to increment or decrement the value. Values less than zero are treated in a special way: when a station decrements a semaphore below zero, NetWare automatically places the station in a queue waiting for the semaphore value to increase to at least zero. If another station increments the semaphore in the meantime, the decrement call returns successfully. If the decrement call times out before the value increases to zero, the call fails. A station can check the value of a semaphore at any time; if the value is below zero, it represents the number of stations waiting to access a resource. Because of the special treatment of negative values, semaphores used as shared variables are generally limited to the range from 0 to 127.

Numerous tests in Windows have shown that making a direct call (via DPMI services) to the NetWare semaphore API causes some kind of locking or access problems. The only known way to get around these problems is to make calls to the Novell NWCALLS.DLL that is provided with Windows 3.1. Therefore, if you use the NWSEMA unit for a Windows target, the code will resolve to making calls to the relevant exported routines from the NWCALLS DLL.

Declarations

Constants

```
nwsErrInvValue  = $7F21; {Negative initial value on open
operation}
nwsErrInvName   = $7F22; {Invalid semaphore name on open
operation, eg null}
nwsErrInvHandle = $89FF; {Invalid semaphore handle passed to
routine}
nwsErrTimeOut   = $897F; {Timeout on nwDecSema}
nwsErrOverflow  = $8901; {Overflow on nwIncSema}
```

Error codes that can be returned by using the semaphore routines.

Types

```
TnwSemaName; = String[127];
```

A type defining a semaphore name.

nwCloseSema / nwDecSema / nwExamineSema

Syntax

```
function nwCloseSema; (Server : TnwServer; Handle : nwLong) : _____  
TnwErrorCode;
```

Purpose

Close a semaphore.

Description

Server is the server's handle for the semaphore. Handle must have been obtained from a previous call to nwOpenSema. nwCloseSema decrements the open count for the semaphore. When the open count reaches zero, the semaphore is automatically deleted by the server. nwCloseSema does not alter the semaphore's value. If you terminate an application without closing your semaphores, the NetWare shell does so itself.

The function returns 0 if it is successful, otherwise it returns nwsErrInvHandle.

See Also

nwOpenSema

Syntax

```
function nwDecSema; (Server : TnwServer; Handle : nwLong;  
                    TimeOut : Word) : TnwErrorCode;
```

Purpose

Decrement the value of a semaphore.

Description

Server is the handle of the server for the semaphore. Handle must have been obtained from a previous call to nwOpenSema. nwDecSema decrements the value of the semaphore. If the result is greater than or equal to zero, the function returns immediately with a result of 0. If the value is negative, the workstation is put in a queue until another workstation increments the semaphore or TimeOut ticks go by (18.2 ticks per second). If another workstation increments the semaphore, the function returns 0. If a timeout occurs, the semaphore's value is incremented again and the function returns nwsErrTimeout.

This routine is named WaitOnSemaphore in the Novell documentation.

See Also

nwIncSema

nwOpenSema

Syntax

```
function nwExamineSema; (Server : TnwServer; Handle : nwLong; var  
Value : nwInt;  
                        var OpenCount : Word) : TnwErrorCode;
```

Purpose

Return the open count and value of a semaphore.

Description

Server is the handle of the server for the semaphore. Handle must have been obtained from a previous call to nwOpenSema.

See Also

nwOpenSema

nwIncSema / nwOpenSema

Syntax

```
function nwIncSema; (Server : TnwServer; Handle : nwLong) :  
  TnwErrorCode;
```

Purpose

Increment the value of a semaphore.

Description

Server is the handle of the server for the semaphore. Handle must have been obtained from a previous call to nwOpenSema. nwIncSema checks the value of the semaphore. If it is 127, the function returns with nwsErrOverflow. If it is less than 127, the value of the semaphore is incremented by one and the function returns with zero.

If there are callers of nwDecSema waiting in a queue when nwIncSema is called, the first process in the queue is released.

This routine is named SignalSemaphore in the Novell documentation.

See Also

nwOpenSema

nwDecSema

Syntax

```
function nwOpenSema; (Server : TnwServer; Name : TnwSemaName;  
  InitialValue : nwInt;  
  var OpenCount : Word; var Handle : nwLong) :  
  TnwErrorCode;
```

Purpose

Open or create a semaphore.

Description

You must call this function before using a semaphore. Name is a string that labels the semaphore. All stations using this semaphore must refer to the same name (although case is not significant).

InitialValue sets the initial value of the semaphore. This value is ignored except when the first station opens the semaphore. The initial value must be in the range from 0 to 127 inclusive.

The number of stations (including the current one) that have opened the semaphore is returned in the variable OpenCount. A four byte handle to the semaphore is returned in the variable Handle. This handle must be passed to all of the other routines for accessing semaphores.

The function returns zero if the semaphore is successfully opened, nwsErrInvName if the name is empty, or nwsErrInvValue if the initial value is invalid.

See Also

nwCloseSema

Novell's Advanced NetWare SFT (system fault tolerant NetWare) provides a sophisticated transaction tracking system to improve data integrity. With transaction tracking services (TTS), the network operating system can guarantee that a complete sequence of operations is written to disk, or that none of it is. The NWTTS module provides functions to:

- determine whether TTS services are available
- begin, end, or abort a transaction
- check status of a transaction
- disable or enable transaction tracking system-wide

NetWare supports two categories of transactions: explicit and implicit. Explicit transactions are initiated when the application makes an explicit call to the server to begin a transaction. Implicit transactions are started automatically when the server recognizes that certain levels of record locking on a transactional file have been activated by the application. The NWTTS module supports only the explicit method.

Before using TTS on a file, remember to mark it transactional by using the NetWare FLAG utility or by calling `nwSetFileAttr` from the `NWFILE` unit (see [_8.A.5](#)).

If you use TTS on a B-Tree Filer fileblock, you should mark the index and data files transactional. Do *not* mark the dialog file transactional. Because of the way the dialog file is used in Filer, marking it transactional causes TTS to misbehave.

The demo program `TTSFILER` shows one method of using transaction tracking with B-Tree Filer.

nwTTSAbort / nwTTSAvailable

Syntax

```
function nwTTSAbort, (Server : TnwServer) : TnwErrorCode;
```

Purpose

Abort a transaction (rollback).

Description

If nwTTSAbort returns with a function result of zero, the transaction has been rolled back, meaning that any disk updates associated with the transaction are undone or removed from server memory buffers. For this function to work, TTS must be available and enabled.

If the transaction is associated with one or more B-Tree Filer fileblocks, be sure to call BTInformTTSAbortSuccessful for each fileblock. This causes B-Tree Filer module to purge its index buffers of any information that might be incorrect after the transaction is backed out.

Aborting a transaction usually releases all physical and logical record locks related to the transaction. Nevertheless, if any locks were placed by calling functions in B-Tree Filer (BTLockFileBlock, for example), they should be unlocked before aborting the transaction. This assures that the Filer internal data structures are correctly updated.

An error code is returned as the function result:

```
0 Transaction successfully aborted.
$89FD TTS is disabled (no rollback was performed).
$89FE Transaction aborted but records remain locked.
$89FF No explicit transaction was active.
```

See Also

BTInformTTSAbortSuccessful (FILER) nwTTSBegin
nwTTSEnd

Syntax

```
function nwTTSAvailable; (Server : TnwServer) : Boolean;
```

Purpose

Determine whether transaction tracking services are available.

Description

Call this function prior to any other TTS routines to determine whether transaction tracking services are available.

See Also

nwTTSDisable nwTTSEnable

nwTTSTBegin / nwTTSTDisable

Syntax

```
function nwTTSTBegin;(Server : TnwServer) : TnwErrorCode;
```

Purpose

Begin a transaction.

Description

A "transaction" is a sequence of logically related file operations that must happen as a group-- either all must be successful or none should be saved to disk. After the transaction begins, no related file operations are committed to disk until nwTTSTEnd is called.

For a file to be used with transaction tracking, it must be flagged as transactional (see nwGetFileAttr). Also, TTS services must be available and enabled (see nwTTSTAvailable and nwTTSTEnable).

A status code is returned in function result:

```
0 Transaction begun.
$8996 Out of dynamic workspace.
$89FE Implicit transaction already active (implicit now made
explicit).
$89FF Explicit transaction already active (existing transaction
continues).
```

Only \$8996 is considered an error condition. If possible, the application should try again later.

Implicit transactions, mentioned in error \$89FE, are not described here. See Novell's Client API reference manuals for more information.

See Also

nwGetFileAttr (NWFILE)	nwTTSTAbort
nwTTSTAvailable	nwTTSTEnable
nwTTSTEnd	

Syntax

```
function nwTTSTDisable;(Server : TnwServer) : Boolean;
```

Purpose

Disable transaction tracking on a server.

Description

The caller must have supervisory privileges in order to disable TTS. The function returns True if it succeeds; False otherwise.

Any transactions written after TTS is disabled cannot be undone. A transaction that has not written anything since transactions were disabled can still be reversed while transactions are disabled. Normally, transactions should be disabled only for the purposes of testing.

See Also

nwTTSTEnable

nwTTSEnable / nwTTSEnd

Syntax

```
function nwTTSEnable;(Server : TnwServer) : Boolean;
```

Purpose

Enable transaction tracking on a server.

Description

The caller must have supervisory privileges in order to disable TTS. If TTS is available, it is active by default. The function returns True if it succeeds; False otherwise.

Any previous transaction backout information is erased, except for transactions that were active when transactions were disabled and did not write anything thereafter. These transactions are not erased and can still be backed out.

Always verify that TTS services are available (by calling nwTTSAvailable) before issuing nwTTSEnable. NetWare sometimes returns a successful result for nwTTSEnable even if TTS is not available.

See Also

nwTTSAvailable

nwTTSDisable

Syntax

```
function nwTTSEnd;(Server : TnwServer; var ID : nwLong) :  
TnwErrorCode;
```

Purpose

End a transaction (commit).

Description

The transaction has not necessarily been committed to disk when this function returns. Use the ID value returned by nwTTSEnd in a call to nwTTSlCommitted to determine when a transaction has actually been written to disk. The Novell documentation states that a transaction takes between 3 and 5 seconds to be committed, however experiments have shown commitment times of 6-7 seconds.

If the server crashes before a transaction is fully committed to disk, any changes will be undone when the file server is rebooted.

If transaction tracking services are disabled but available, ID can still be passed to nwTTSlCommitted to determine when the transaction has been committed to disk. Any transactions written after TTS is disabled cannot be backed out.

If the transaction is associated with one or more B-Tree Filer fileblocks, be sure to call BTUnlockFileBlock for each fileblock before calling nwTTSEnd. This causes B-Tree Filer to flush its index buffers of any information that might be associated with the transaction.

Ending a transaction releases all physical and logical record locks related to the transaction.

The function returns a status code:

```
0 Transaction ended successfully.  
$89FD TTS disabled.  
$89FE Transaction ended but records remain locked.  
$89FF No explicit transaction was active.
```

See Also

BTUnlockFileBlock (FILER)
nwTTSlCommitted

nwTTSEnable

nwTTSIsCommitted

Syntax

```
function nwTTSIsCommitted; (Server : TnwServer; ID : nwLong) : Boolean;
```

Purpose

Determine whether a transaction was successfully written to disk.

Description

After calling nwTTSEnd, you can call nwTTSIsCommitted to verify that a transaction has been committed to disk. ID is the transaction reference number returned by nwTTSEnd.

nwTTSIsCommitted returns True when the transaction has been committed; False if it hasn't yet been committed. NetWare only checks the ID against its internal queue of active transactions. If the ID isn't found in this queue, nwTTSIsCommitted returns True. Hence passing an invalid ID returns True.

The Novell documentation states that a transaction will take between 3 and 5 seconds to be committed, however experiments have shown commitment times of 6-7 seconds. Therefore, you should not wait for an individual transaction to be committed, but collect the IDs from several transactions, and then check them all to be committed in one operation. Of course, you do not need to do this check. NetWare ensures that transactions are committed, and the latest data is always presented to workstations without the need to wait for nwTTSIsCommitted to return True.

This function can also be used to determine if a transaction has been committed to disk even if nwTTSEnd reports that transaction tracking is disabled.

See Also

nwTTSDisable

nwTTSEnd

NWPRINT provides routines to control the local workstation's printer capture and also to access print queues on a server. Printer capture refers to the ability for the local NetWare shell to 'capture' all the data that is sent to a local printer (LPT1, LPT2, etc) and send it instead to a file on the server or to a print job on a server's print queue. This capability is usually provided by NetWare's CAPTURE program, and NWPRINT provides an interface to the same functions, allowing you to integrate capture into your application. It provides the following capture capabilities:

- determine the number of local printers (under the VLM Requester, you can access up to 9 local printers, from LPT1 to LPT9)
- start a capture session to a print job on a print queue
- start a capture session to a file on a server volume
- get or set the capture flags (the configuration settings) for each printer
- flush a capture
- close a capture
- get and set the name used on banner pages

In addition, NWPRINT provides several routines to access a NetWare print queue. These functions are useful when the output to be printed is already stored in a file.

- enumerate the print queues on a server
- create a new print job on a queue (returning a writeable Pascal file variable that is used to add data to the print job)
- close an open print job
- abort an open print job
- alter the position of a print job in a print queue
- get and set the attributes of a print job
- delete a print job from a queue

Printer Capture

The main problem with setting up capture for a printer is when to call the capture routines, how and when to output to the captured printer, and what interactions are required between your application and the capture process.

There are three main cases to consider. You will output data to the printer in one of three ways:

1. via BIOS interrupt \$17 calls
2. via unbuffered DOS file calls
3. via buffered DOS file calls

The first case is the simplest. Start the capture first (call `nwStartCaptureToFile` or `nwStartCaptureToQueue`). Output characters to the relevant printer via INT \$17 function 0. You can then abort the capture by calling `nwCancelCapture` to discard all the printer output captured so far, or end the capture by calling `nwEndCapture` to close the print job and release it for printing (if you were capturing to a file instead, the file is closed). If you are capturing to a print queue, you can

call `nwFlushCapture` to close the current print job and release it for printing, but the printer remains captured to the same print queue.

The second case uses unbuffered DOS calls. This means that the application itself is not buffering any data to the printer. With Borland Pascal, this is usually done by using a variable of type `File` and using `BlockWrite` to output data to the file. In this case you should start capturing first by calling `nwStartCaptureToFile` or `nwStartCaptureToQueue`. Open the printer file by using `Assign` and `Rewrite`; the name used by `Assign` will be one of 'LPT1' to 'LPT9' depending on the printer. Output to this open file by using `BlockWrite` calls. If you now want to abort the capture and discard all output, call `nwCancelCapture` first and then call `Close` to close the printer. If you want to end the capture, call `Close` to close the file and then call `nwEndCapture` (or `nwCancelCapture`). The call to `Close` will be intercepted by the NetWare shell and cause the print job to be closed and released for printing (or the capture file to be closed). To close the current print job but leave the printer captured, then either call `nwFlushCapture`, or close the file and reopen it with `Rewrite`. Do not do this for a printer that is captured to a file; the only options in this case are canceling or ending the capture.

The final case uses buffered DOS calls. With Borland Pascal, this is the normal case because it is done through the use of a variable of type `Text` and calls to `Write` and `WriteLn`. Pascal's text file routines internally buffer characters to the printer. Here you must ensure that any buffered data is flushed before calling any capture routines. You should first start the capture, then open the printer as a file by using `Assign` and `Rewrite`. Output to the file with `Write` and `WriteLn`. Now comes the choice of what to do with the capture. You *must* call the Pascal `Flush` routine first to ensure that the text file buffers are flushed to DOS. Once this is done, then the choices are the same as the previous case. If you now want to abort the capture and discard all output, call `nwCancelCapture` first and then call `Close` to close the printer. If you want to end the capture, call `Close` to close the file and then call `nwEndCapture` (or `nwCancelCapture`). The call to `Close` will be intercepted by the NetWare shell and cause the print job to be closed and released for printing (or the capture file to be closed). To close the current print job but leave the printer captured, then either call `nwFlushCapture`, or close the file and reopen it with `Rewrite`. Do not do this for a printer that is captured to a file; the only options in this case are canceling or ending the capture.

Declarations

Constants

```
nwjcAutoStart      = $08;      {autostart even if server connection  
broken}  
nwjcRestart        = $10;      {remains in queue after job cancelled}  
nwjcEntryOpen      = $20;      {job file is being created}  
nwjcUserHold       = $40;      {user has job on hold}  
nwjcOperatorHold   = $80;      {operator has job on hold}
```

Bit masks used for the `JobControlFlags` field in the `TnwPrintJob` structures.

```
nwpErrBadPrinter   = $7F51;  
nwqErrNoSuchJob    = $89D5;
```

Error codes for the NWPRINT unit. `nwpErrBadPrinter` means that a bad `nwLPTx` value was passed to a routine: there are not enough printers that can be captured. `nwqErrNoSuchJob` means that the print job (as defined by a `TnwPrintJob` variable) is invalid or no longer exists (it has been printed or deleted from the queue).

```
nwpfSuppressFF     = $08;      {do not issue form feed at end}  
nwpfTabExpand       = $40;      {enable tab expansion}  
nwpfPrintBanner     = $80;      {print banner page}
```

Bit masks used for the PrintFlags fields in the TnwCaptureFlags and TnwPrintJob structures.

```
nwPrintASAP; : TnwDate = (...);
```

Set the TargetExecTime field in your TnwPrintJob variable to this value to make a print job print as soon as possible.

Types

```
TnwBannerName; = String[12];
TnwBannerJob; = String[12];
TnwFormName; = String[12];
```

A user name string and a job name string for use on a banner page. TnwFormName is a string for a form name in the TnwCaptureFlags and TnwPrintJob structures.

```
TnwCaptureFlags; = record
  {read/write values - set by nwSetCaptureFlags}
  PrintFlags      : Byte;          {print flags}
  TabSize         : Byte;          {tab size (1..18)}
  NumCopies       : Byte;          {number of copies (1..255)}
  FormType        : Byte;          {form type (0..255)}
  MaxLines        : Word;          {maximum lines per page}
  MaxCols         : Word;          {maximum columns per line}
  FlushTimeout    : Word;          {ticks before automatic flush}
  BannerJobName   : TnwBannerJob; {job name for banner page}
  FlushOnClose    : Boolean;       {True when autoflush enabled}
  FormName        : TnwFormName;   {name of the current form}
  {read only values - returned by nwGetCaptureFlags}
  Printer         : TnwPrinter;    {the local printer}
  IsCaptured     : Boolean;        {True if printer is captured}
  IsCapturingData : Boolean;       {True if print data is being
captured}
  IsDoingTimeOut  : Boolean;        {True if capture is timing out}
  IsCapturedToFile : Boolean;       {True if captured to a file}
  Server          : TnwServer;     {server processing the capture}
  QueueID         : nwLong;        {print queue bindery ID if}
end;                               { IsCapturedToFile is False}
```

Data structure used and returned by the workstation capture flag routines nwGetCaptureFlags and nwSetCaptureFlags. The first part of the structure contains all the fields that you can set with nwSetCaptureFlags. The last part of the structure contains the extra fields that are returned from the NetWare shell with nwGetCaptureFlags.

The read/write fields are:

PrintFlags	Bit-mapped flags (nwpfSuppressFF, nwpfTabExpand, nwpfPrintBanner).
TabSize	Number of spaces for each tab when expansion is enabled (1..18).
NumCopies	Number of copies to print (1..255).
FormType	Form type (0..255). If the form type currently recorded by
	NetWare differs, a message is displayed on the server console.
MaxLines	The default form type is zero.
automatic form	The maximum number of lines per page before an
	feed is inserted.
MaxCols	The maximum number of columns per line before an
automatic line	break is inserted.
FlushTimeout	The time in clock ticks that the capture process
waits without	

job. Zero	receiving any characters before it flushes the print
	means no timeout.
BannerJobName	A string that describes the job name for the banner
page. This	
nwSetBannerName.	is a different name than the one controlled by
	If it contains a null string, the capture file name
is printed	in this location on the job banner.
FlushOnClose	True if the print job is flushed when the DOS LPT
device is	
printing).	closed (i.e., the print job is released for
FormName	A string containing the name of the current form.
This is	displayed for informational purposes by PCONSOLE.

The read only fields are:

```
Printer      The local DOS LPT device (nwLPT1..nwLPT9) that
these flags are
for.
IsCaptured   True if the printer is currently captured.
IsCapturingData True if print output data is being captured.
IsDoingTimeout True if the capture is timing out.
IsCapturedToFile True if the printer is being captured to a file.
Server       Handle of the server processing the capture.
QueueID      Bindery ID of the print queue, if captured to one.
```

When a printer is captured, but before any data is written, IsCaptured is True. If the printer is captured to a print queue, IsCapturedToFile is False, and the Server and QueueID fields uniquely identify the queue. If the printer is captured to a file, IsCapturedToFile is True and Server is the handle of the server where the file is (note that the NetWare Client API does not allow you to find out the name of the file being written to). Once the first character is written to the printer after the capture is started, IsCapturingData is set to True and, if a timeout was set, IsDoingTimeout is also set to True.

```
TnwEnumPrintJobFunc; = function (JobNumber : nwLong; var ExtraData)
: Boolean;
```

Type of the function that nwqEnumPrintJobs calls when it enumerates the available print jobs on a queue. JobNumber is the number of the print job, and ExtraData is an untyped var parameter that was passed to the original call to nwqEnumPrintJobs. The function should return True if the enumeration is to continue, False if it should stop.

```
TnwEnumQueueFunc; = function (Name : TnwObjectStr; ID : nwLong;
var ExtraData) : Boolean;
```

Type of the function that nwEnumQueues calls when it enumerates the available print queues on a server. Name is the queue name, ID its bindery object ID, and ExtraData is an untyped var parameter that was passed to the original call to nwEnumQueues. The function should return True if the enumeration is to continue, False if it should stop.

```
TnwPrinter; = (nwLPT1, nwLPT2, nwLPT3, nwLPT4, nwLPT5, nwLPT6,
nwLPT7, nwLPT8,
nwLPT9);
```

A type that enumerates the possible printers for the print capture routines. nwLPT4 through nwLPT9 are only available with the VLM Requester. See the Novell NetWare workstation configuration documentation for information on how to enable them.

```
TnwPrintJob; = record
  VerifyFlag      : Word;          {** verify flag}
  Server          : Word;          {** server handle}
  QueueID         : nwLong;        {** print queue bindery ID}
  ServerVersion   : Word;          {** effective server version}
  ClientStation   : nwLong;        {** client who started job: conn.
number}
  ClientTaskNum   : nwLong;        {** ...task number}
  ClientID        : nwLong;        {** ...bindery object ID}
  TargetServerID  : nwLong;        {  print server ID, -1 = any}
  TargetExecTime  : TnwDate;       {  time to print job, $FF =
ASAP}
  JobEntryTime    : TnwDate;       {** time job entered the queue}
  JobNumber       : nwLong;        {** job number}
  JobType         : Word;          {** type of job (usually 0)}
  JobPosition     : Word;          {** position in queue, 1 = at
top}
  JobControlFlags : Word;          {  job control flags}
  JobFileName     : String[13];    {** filename of queue job}
  JobFileHandle   : TnwFileHandle; {** NetWare handle of
```

```

JobFileName}
  ServerStation : nwLong;      (** print server: conn.number}
  ServerTaskNum : nwLong;      (** ...task number}
  ServerID      : nwLong;      (** ...bindery object ID}
  JobDesc       : String[49];  { description of job}
  PrintFlags    : Byte;        { print flags}
  TabSize       : Byte;        { default tab size}
  FormName      : TnwFormName; { name of form to print on}
  NumCopies     : Byte;        { number of copies}
  MaxLines      : Word;        { max lines per page}
  MaxCols       : Word;        { max columns per page}
  BannerUserName : TnwBannerName; { user name for banner page}
  BannerJobName  : TnwBannerJob; { job name for banner page}
  JobFileSize    : nwLong;      (** size of file JobFileName}
end;

```

The data structure that defines a print job on a queue. A variable of this type is initialized by `nwCreatePrintJobFile`, and is maintained by `nwRefreshPrintJob` amongst others. The fields marked **(**)** are read only and are set either by Queue Management Services (QMS) or by the NWPRINT unit. All other fields can be changed by `nwChangePrintJob`. You should not change the read only fields because the print job routines use them to track the print job (in particular the `Server`, `QueueID`, `ServerVersion` and `JobNumber` fields). The `TnwDate` fields `TargetExecTime` and `JobEntryTime` do not return and do not use the `WeekDay` field.

The following describes the `TnwPrintJob` fields:

<code>VerifyFlag</code>	Used for validity checking.
<code>Server</code>	Server handle for the print queue.
<code>QueueID</code>	Bindery ID of the print queue.
<code>ServerVersion</code>	Effective server version: early versions of NETX cannot use
	the 'new' NetWare print job data structure, so this
	forces
	the NWPRINT routines to use 'old' calls.
<code>ClientStation</code>	Connection number of the workstation of the user
<code>who started</code>	the job.
<code>ClientTaskNum</code>	User's task number.
<code>ClientID</code>	Bindery ID for the user.
<code>TargetServerID</code>	Bindery ID for the print server ID that will
<code>service the job</code>	(-1 means any print server can service the job).
<code>TargetExecTime</code>	Time to print the job (use <code>nwPrintASAP</code> to print as
<code>soon as</code>	possible).
<code>JobEntryTime</code>	Time that the print job was created on the queue.
<code>JobNumber</code>	Job number.
<code>JobType</code>	Type of job (usually 0).
<code>JobPosition</code>	Job's position in the queue (1 means at the top).
<code>JobControlFlags</code>	Job control flags (see the <code>nwjfXXXXX</code> constants).
<code>JobFileName</code>	Name of file where QMS is storing data.
<code>JobFileHandle</code>	NetWare handle of <code>JobFileName</code> .
<code>ServerStation</code>	Connection number of the print server.
<code>ServerTaskNum</code>	Task number of the print server.
<code>ServerID</code>	Bindery ID of the print server.
<code>JobDesc</code>	String describing the job (for display in
<code>PCONSOLE).</code>	
<code>PrintFlags</code>	Bit-mapped print flags (see the <code>nwpfXXXXX</code>
<code>constants).</code>	
<code>TabSize</code>	Number of spaces for each tab when expansion is

enabled (1..18).

FormName	Name of the form to print on.
NumCopies	Number of copies.
MaxLines	Maximum number of lines per page.
MaxCols	Maximum number of columns per page.
BannerUserName	User name for the banner page.
BannerJobName	Job name for the banner page.
JobFileSize	The size of the print job's file.

nwCancelCapture / nwEndCapture / nwEnumQueues

Syntax

```
function nwCancelCapture;(Printer : TnwPrinter) : TnwErrorCode;
```

Purpose

Abort a capture session.

Description

If the printer was captured to a print queue, then the print job is closed and deleted from the print queue. If the capture was to a file, the file is closed and deleted from the server. If no print data has been captured since the start capture call, a call to this routine just stops the capture.

After a call to nwCancelCapture, the printer is no longer captured. Call nwStartCaptureToQueue or nwStartCaptureToFile to start the capture for the printer again.

The function returns an error code in its function result, or zero if it is successful.

See Also

nwEndCapture	nwStartCaptureToFile
nwStartCaptureToQueue	

Syntax

```
function nwEndCapture;(Printer : TnwPrinter) : TnwErrorCode;
```

Purpose

Close a capture session.

Description

If the printer was captured to a print queue, then the print job is closed and released for printing. If the capture was to a file, the file is closed. If no print data has been captured since the start capture call, a call to this routine just stops the capture.

After a call to nwEndCapture, the printer is no longer captured. Call nwStartCaptureToQueue or nwStartCaptureToFile to start the capture for the printer again.

The function returns an error code in its function result, or zero if it is successful.

See Also

nwCancelCapture	nwStartCaptureToFile
nwStartCaptureToQueue	

Syntax

```
procedure nwEnumQueues;(Server : TnwServer; EnumFunc :  
  TnwEnumQueueFunc;  
  var ExtraData);
```

Purpose

Enumerate the print queues on a server.

Description

This procedure enumerates the available print queues on the specified server. For each print queue found it calls a function (EnumFunc) of the TnwEnumQueueFunc type. ExtraData is an untyped var parameter that is not used by nwEnumQueues. It is just passed on as a parameter to your EnumFunc. You can use this parameter to pass any extra data you need.

nwFlushCapture / nwGetBannerName

Syntax

```
function nwFlushCapture, (Printer : TnwPrinter) : TnwErrorCode;
```

Purpose

Flush a capture.

Description

The effect of this call depends on whether the printer was captured to file or print queue.

If the printer was captured to a print queue, the print job is closed and released for printing. The printer is still captured after a call to this routine and will be captured to the same print queue. Sending another character to the printer starts another print job on that queue.

If the capture was to a file, the file is closed. The printer is no longer captured after this call.

The function returns an error code in its function result, or zero if it is successful.

A problem occurs when capturing to a file under NETX and using DOS calls to access the printer. For example, assume you capture printer LPT3 to a file, open the LPT3 device via a DOS open file call (with Assign and Rewrite), and then output data to the LPT3 file (with WriteLn). If you close LPT3 (with Close) the NetWare shell does an automatic flush capture call on the printer. Under NETX this raw flush call does not end the capture but leaves the LPT3 printer captured, however it is in some indeterminate state. The best way to avoid this problem is to not use nwFlushCapture when capturing to a file, but to call nwEndCapture immediately after the call to close the DOS file.

See Also

nwCancelCapture

nwEndCapture

Syntax

```
function nwGetBannerName; (var Name : TnwBannerName) : TnwErrorCode;
```

Purpose

Get the user name for print job banner pages.

Description

When the NetWare capture facilities are used, a banner page that identifies the origin of each print job optionally precedes the printout itself. nwGetBannerName returns the user name that will appear on this banner page. The user name can be changed by calling nwSetBannerName. The default for the banner user name is generally the user's login name.

There is only one banner user name per workstation. You cannot have one banner name per server or per printer.

The function returns a status code in its function result, or zero if it is successful.

See Also

nwSetBannerName

nwGetCaptureFlags / nwGetNumPrinters / nwIsCaptured

Syntax

```
function nwGetCaptureFlags;(Printer : TnwPrinter;  
                           var CapFlags : TnwCaptureFlags) :  
TnwErrorCode;
```

Purpose

Get the capture flags for the printer.

Description

A variable of type TnwCaptureFlags describes how a particular job will be printed. It applies to print jobs started using capture facilities (nwStartCaptureToFile or nwStartCaptureToQueue).

Generally you should start the capture for a printer, call nwGetCaptureFlags to get the current settings, modify the field(s) that you care about, and then call nwSetCaptureFlags to alter the settings for this capture session. At this point you can start printing to the printer.

The capture flag values are not preserved from capture session to capture session. You should assume that the current values are lost when nwEndCapture or nwCancelCapture is called. Under some versions of the VLM Requester you should not call nwGetCaptureFlags if the printer is not captured because, although the function is successful, the capture flags structure may contain invalid data. Therefore, you should *only* call nwGetCaptureFlags when the printer is captured (i.e., nwIsCaptured returns True for the printer).

A lot of information is returned in CapFlags, and much of it is fairly esoteric. The typical fields you will need to alter are PrintFlags, NumCopies, JobName, FlushTimeout, and possibly the form definition fields. See the description of TnwCaptureFlags earlier in this section.

The function returns a status code in its function result (zero means successful).

See Also

nwGetBannerName

nwSetCaptureFlags

Syntax

```
function nwGetNumPrinters; : Byte;
```

Purpose

Return the number of printers available for capture.

Description

Under NETX, this routine always returns 3 (for the LPT1, LPT2 and LPT3 printers).

Under the VLM Requester, the total number of printers is configurable by the user in the NET.CFG file and the value varies from 0 to 9. If you configure 0 printers, the PRINT.VLM module does not load itself, and all of the capture routines return nwErrBadPrinter. This routine also returns 0 for the number of printers if the PRINT.VLM module is not loaded (loading it is optional).

Syntax

```
function nwIsCaptured;(Printer : TnwPrinter) : Boolean;
```

Purpose

Return True if the printer is being captured.

Description

Under some versions of the VLM Requester, you must call this routine before calling nwGetCaptureFlags or nwSetCaptureFlags. Otherwise nwGetCaptureFlags might return invalid values and nwSetCaptureFlags might seem to succeed, but in fact does not do anything.

nwGetCaptureFlags / nwGetNumPrinters / nwIsCaptured

See Also

~~nwEndCapture~~

~~nwStartCaptureToFile~~

nwStartCaptureToQueue

nwqAbortPrintJobFile / nwqChangePrintJob

Syntax

```
function nwqAbortPrintJobFile;(var PrintJobData : TnwPrintJob;
                               var F : file) : TnwErrorCode;
```

Purpose

Abort and delete an open print job on a queue.

Description

This routine aborts an open print job that was created with `nwqCreatePrintJobFile`. `PrintJobData` is the variable that was returned by the create routine and `F` is the file variable that goes with it. This function deletes the print job from the queue (all data in the print job is discarded) and closes the file variable `F`.

After this call, the `PrintJobData` variable cannot be used by any other print job routine (the print job it references no longer exists), except another call to `nwqCreatePrintJobFile`.

Do not use this routine if you closed the print job by calling `nwqClosePrintJobFile` and need to delete the print job. Use `nwqRemovePrintJob` instead.

See Also

`nwqClosePrintJobFile`

`nwqCreatePrintJobFile`

Syntax

```
function nwqChangePrintJob;(var PrintJobData : TnwPrintJob) :
TnwErrorCode;
```

Purpose

Change the settings for a print job.

Description

This routine changes various settings for a print job. `PrintJobData` must have been initialized by a call to `nwqCreatePrintJobFile`. The definition of the `TnwPrintJob` type (earlier in this section) shows which settings of the print job you can change. Generally you will be changing the description strings, the job control settings, and the print flag settings.

You can call this routine at any time in the lifetime of the print job, you do not have to wait until the print job is closed. For example, the usual method is to create the new print job with `nwqCreatePrintJobFile`, alter the settings of the print job by calling this routine (i.e., put a user hold on it, alter the `JobDesc` field to a useful text string), write data to the print job, and then close the print job by calling `nwqClosePrintJobFile`.

This routine cannot be used to alter the position of a print job in the queue; this is done by calling `nwqChangePrintJobPos`, and requires the user to have queue operator rights for the queue.

See Also

`nwqChangePrintJobPos`
`nwqCreatePrintJobFile`

`nwqClosePrintJobFile`

nwqChangePrintJobPos / nwqClosePrintJobFile

Syntax

```
function nwqChangePrintJobPos, (var PrintJobData : TnwPrintJob;  
                                NewPosition : Word) : TnwErrorCode;
```

Purpose

Change the position of a print job in a queue.

Description

PrintJobData must have been initialized by a call to nwqCreatePrintJobFile or nwqGetPrintJob. NewPosition is the numeric position that you want the print job to be moved to (use a value of 1 for the top of the queue). If you pass a value for NewPosition that is beyond the last job in the queue, the job is moved to the end of the queue.

After this function completes, PrintJobData contains the latest information about the print job.

To use this routine you must have queue operator rights, otherwise an error code is returned.

See Also

nwqChangePrintJob	nwqCreatePrintJobFile
nwqGetPrintJob	

Syntax

```
function nwqClosePrintJobFile, (var PrintJobData : TnwPrintJob;  
                                var F : file) : TnwErrorCode;
```

Purpose

Close an open print job on a queue and mark it ready for printing.

Description

This routine closes an open print job that was created with nwqCreatePrintJobFile. PrintJobData is the variable returned by the create routine and F is the file variable that goes with it. nwqClosePrintJobFile closes the print job and NetWare's Queue Management System marks the print job ready for processing by a print server. The file variable F is closed.

The print job is printed when it reaches the top of the queue (the JobPosition field has value 1), providing that there is no hold on the print job (either an operator hold or a user hold) or that you have not specified a TargetExecTime value for some time in the future. You can modify the job control flags, including the hold options, by calling nwqChangePrintJob. To modify the job's position, call nwqChangePrintJobPos.

After this function completes, PrintJobData contains the current information about the print job (its position in the queue and its final size).

See Also

nwqChangePrintJob	nwqChangePrintJobPos
nwqClosePrintJobFile	nwqCreatePrintJobFile

nwqCreatePrintJobFile

Syntax

```
function nwqCreatePrintJobFile;(Server : TnwServer; QueueName : TnwObjectStr;  
                                var PrintJobData : TnwPrintJob;  
                                var F : file) : TnwErrorCode;
```

Purpose

Create a new print job on a queue and return a file variable for it.

Description

This routine creates a print job on a queue, and allows you to write data to it. It returns a variable of type File (which has been opened by the routine) that is used to write to the print job, and also a variable of type TnwPrintJob which is used to track the print job.

Server and QueueName uniquely define the print queue. The queue must be a print queue and it must exist on the server given by the server handle. You can get the names of all the print queues on a server by calling nwEnumQueues.

After this function completes, PrintJobData contains the information about the print job. You use this variable for all the other print job routines (to alter the settings for the job, to alter its position in the queue, to delete it from the queue). See the definition of TnwPrintJob earlier in this section for details on the fields within the variable. The print job is created with the following default values: -1 for the TargetServerID (meaning any print server can print the print job), TargetExecTime is set to nwPrintASAP (meaning print as soon as possible), JobDesc and BannerJobName are set to '(unknown)', and BannerUserName is set to the value returned by nwGetBannerName.

You must pass a closed file variable as a parameter to the routine, and it is returned opened, with a record length of 1 byte. You can output data to this file by using BlockWrite and the data goes directly to the print job on the queue.

Once you have written data to the print job, you have a choice of actions to take. You can close the print job by calling nwqClosePrintJobFile and allow the print server to print the job. Or you can abort the print job by calling nwqAbortPrintJobFile to discard all the data and delete the print job. Both routines close the file variable.

You must never close the file variable F yourself because there is a link between the file and the print job. Call nwqClosePrintJobFile or nwqAbortPrintJobFile instead.

See Also

nwEnumQueues
nwqClosePrintJobFile

nwqAbortPrintJobFile

nwqEnumPrintJobs / nwqGetPrintJob

Syntax

```
procedure nwqEnumPrintJobs;(Server : TnwServer; QueueName :  
  TnwObjectStr;  
  
  EnumFunc : TnwEnumPrintJobFunc; var  
  ExtraData);
```

Purpose

Enumerate all the print jobs on a queue.

Description

This procedure enumerates the current print jobs on the specified print queue and server. For each print job, it calls a function (EnumFunc) of the TnwEnumPrintJobFunc type. ExtraData is an untyped var parameter that is not used by nwqEnumPrintJobs. It is just passed on as a parameter to your EnumFunc. You can use it to pass any extra data you need.

You can use nwqEnumPrintJobs to build a list of print job numbers. Then to get the information for a single print job, call the nwqGetPrintJob routine (this can be done inside your EnumFunc routine).

Under NetWare 2.x, the internal call made by nwqEnumPrintJobs can obtain all the print job numbers as one snapshot in time, and there can a maximum of 250 print jobs active in any one queue. However, under NetWare 3.x and 4.x, the internal call can only get print job numbers in batches of 125, and there can be many more jobs in a queue than 250. Hence if the number of jobs in a queue is greater than 125 there must be two or more internal calls to get the full list of job numbers. Between these calls to NetWare's Queue Management Services, a print job might be fully processed and removed, moving all the other jobs up one position in the queue. This means that the second call to NetWare will 'miss' a print job number because it moved into the first 125 print jobs. Unfortunately there is nothing that can be done about this problem other than making you aware of it.

See Also

nwqGetPrintJob

Syntax

```
function nwqGetPrintJob;(Server : TnwServer; QueueName :  
  TnwObjectStr;  
  
  JobNumber : nwLong;  
  var PrintJobData : TnwPrintJob) :  
  TnwErrorCode;
```

Purpose

Return the print job data for a print job on a queue.

Description

This function is designed to be used in tandem with nwqEnumPrintJobs to get the print job information for a print job on a queue. For each print job on the queue, nwqEnumPrintJobs calls your EnumFunc with the job number as one of the parameters. You can then use this job number in a call to nwqGetPrintJob to get the information about the print job (the user name, the job description, and so on).

You can use nwqEnumPrintJobs and nwqGetPrintJob together to write a simple PCONSOLE type utility.

nwqGetPrintJob returns an error code, or zero if it is successful. The most common error code is nwqErrNoSuchJob which means either you passed an invalid job number or the print job was completed and no longer exists.

See Also

nwqEnumPrintJobs

nwqRefreshPrintJob / nwqRemovePrintJob

Syntax

```
function nwqRefreshPrintJob; (var PrintJobData : TnwPrintJob) : _____  
TnwErrorCode;
```

Purpose

Get the latest information about a print job.

Description

PrintJobData must have been initialized by a call to nwqCreatePrintJobFile or nwqGetPrintJob. After this function completes, PrintJobData contains the latest information about the print job.

Generally the only field that changes asynchronously is the JobPosition field, when print jobs prior to this in the queue get printed and removed from the queue.

See Also

nwqChangePrintJob	nwqCreatePrintJobFile
nwqGetPrintJob	

Syntax

```
function nwqRemovePrintJob; (var PrintJobData : TnwPrintJob) :  
TnwErrorCode;
```

Purpose

Delete a print job from a queue.

Description

PrintJobData must have been initialized by a call to nwqCreatePrintJobFile or nwqGetPrintJob. This function deletes the print job from the queue and all print data associated with it is discarded.

The print job must be closed by calling nwqClosePrintJobFile before you can call this routine. If you want to delete an open print job, call nwqAbortPrintJobFile instead.

After this call, PrintJobData cannot be used by any other print job routine (the print job it references no longer exists), except nwqCreatePrintJobFile.

See Also

nwqAbortPrintJobFile	nwqClosePrintJobFile
nwqCreatePrintJobFile	nwqGetPrintJob

nwSetBannerName / nwSetCaptureFlags

Syntax

```
function nwSetBannerName, (Name : TnwBannerName) : TnwErrorCode;
```

Purpose

Set the user name for print job banner pages.

Description

When the NetWare capture facilities are used, a banner page that identifies the origin of each print job optionally precedes the printout itself. Name is a string that appears on this banner page and can be at most 12 characters long.

There is only one banner user name per workstation. You cannot have one banner name per server or per printer.

The function returns a status code in its function result (zero means successful).

See Also

nwGetBannerName

Syntax

```
function nwSetCaptureFlags; (Printer : TnwPrinter;  
                             CapFlags : TnwCaptureFlags) :  
TnwErrorCode;
```

Purpose

Set the capture flags for the printer.

Description

A variable of type TnwCaptureFlags describes how a particular job will be printed. It applies to print jobs started using capture facilities (nwStartCaptureToFile or nwStartCaptureToQueue).

This function enables an application to control various capture attributes such as the banner page, tab size, number of copies, and so on.

Generally you should start the capture for a printer, call nwGetCaptureFlags to get the current settings, modify the field(s) that you care about, and then call nwSetCaptureFlags to alter the settings for this capture session. At this point you can start printing to the printer.

The capture flag values are not preserved from capture session to capture session. You should assume that the current values are lost when nwEndCapture or nwCancelCapture is called. Generally, under the NETX shell the capture flags are preserved, but under the VLM Requester, they are not. You should only use nwSetCaptureFlags when the printer is captured (i.e., nwIsCaptured returns True for the printer).

The typical fields of the CapFlags record you will need to alter are PrintFlags, NumCopies, JobName, FlushTimeout, and possibly the form definition fields. See the description of TnwCaptureFlags earlier in this section.

The function returns a status code in its function result (zero means successful).

See Also

nwGetCaptureFlags

nwSetBannerName

nwStartCaptureToFile / nwStartCaptureToQueue

Syntax

```
function nwStartCaptureToFile; (Printer: TnwPrinter;
                               FileName : String) : TnwErrorCode;
```

Purpose

Start capturing printer output to a file.

Description

When this function is called, there cannot be any capture active for the printer. You can check if a capture is active by calling `nwIsCaptured`. Then use `nwCancelCapture` if necessary. The file specified by `FileName` must specify a path that is on a NetWare server, you cannot capture to a file on a local disk or to a file on a remote non-NetWare drive.

After this function is called, all output to the printer is redirected to the specified file. The file is created as a new file. If the file exists already, it is deleted first.

Because of NETX limitations, it is an error to capture more than one printer to a file. `nwStartCaptureToFile` does not check for this possibility, so you must. This is not a problem with the VLM Requester because it allows multiple captures to file.

See Also

<code>nwCancelCapture</code>	<code>nwIsCaptured</code>
<code>nwStartCaptureToQueue</code>	

Syntax

```
function nwStartCaptureToQueue; (Printer : TnwPrinter; Server :
TnwServer;
                               QueueName : TnwObjectStr) :
TnwErrorCode;
```

Purpose

Start capturing printer output to a print queue.

Description

When this function is called, there cannot be any capture active for the printer. You can check if a capture is active by calling `nwIsCaptured`. Then use `nwCancelCapture` if necessary. The print queue specified by `QueueName` must exist on the NetWare server defined by the handle in `Server`.

After this call executes successfully, the local NetWare shell is primed to capture all printer output to the queue. However, a print job has not yet been created on the queue. A print job is created when the shell intercepts the first character output to the captured printer.

See Also

<code>nwCancelCapture</code>	<code>nwIsCaptured</code>
<code>nwStartCaptureToFile</code>	

NetWare supports two advanced methods of communicating between workstations. These methods are used internally by NetWare itself and have the significant advantage that they don't use any server resources. These advanced methods are referred to as IPX and SPX services.

IPX Services

IPX stands for Internetwork Packet eXchange protocol. It is Novell's implementation of the Internetwork Datagram Packet Protocol designed by Xerox. The term "internetwork" is used to refer to the current network as well as any other networks that may be bridged to it. IPX requires Advanced NetWare version 2.0 or later. Applications running under Advanced NetWare use IPX drivers to communicate directly with other workstations, servers, or devices.

Each device on the internetwork is called a node and has a unique address. IPX allows a node to send a packet to, or receive a packet from, any other node. Packets are arbitrary byte sequences as long as 546 bytes (some networks now support larger IPX packets). IPX isolates the application from the details of the packet's physical routing. Although IPX does not guarantee delivery of a packet (the other node might still ignore it), guaranteed delivery protocols can be built using its services. SPX (described in the following) is a guaranteed delivery protocol based on IPX.

The NWIPXSPX unit implements IPX routines that hide some of the details of the primitive IPX calls provided by Novell. This encapsulation makes the routines easier for you to use, and also allows the same routines to work under both DOS and Windows.

Two important data types associated with IPX services are the IPX header, `TipxHeader`, and the event control block, `TipxECB`. `TipxHeader` contains routing information used by IPX to deliver a packet. `TipxECB` includes a pointer to an `TipxHeader` and additional pointers to one or more data buffers that store the packet itself.

To send a packet to another node, several fields in the `TipxHeader` must be specified:

- the network address, a four byte number
- the physical node address, a six byte number
- the socket, a two byte number

The network address specifies the *network* of the destination node (required since IPX can bridge networks). Storing zero in this field indicates the same network as that of the sender.

The node address specifies a particular station within the network. The function `IPXInternetAddress` returns the network and node address of the calling station, and `nwGetInternetAddress` (described in [_8.A.2](#)) returns the address of a given connection on the current network. The clever use of IPX broadcasts also allows a sending station to determine the address of any station that is listening for the broadcast.

In addition to the node and network addresses, sending and receiving workstations must agree on a "socket." You can think of sockets as two-way radio channels; both sender and receiver must be tuned to the same channel for communications to occur. A node may have up to 20 sockets open at a time as default. You can specify the socket capacity of a workstation in its `NET.CFG` file. See the NetWare documentation for details.

The caller must specify a socket number that doesn't conflict with other sockets currently active on the network. Socket numbers ranging from \$4000 to \$7FFF are called dynamic sockets; these are fair game for any user of the network. Values \$8000 and larger are well-known sockets, whose values are reserved by Novell for approved applications. Sockets less than \$4000 are reserved by Novell for internal use.

The easiest way to select unique but agreed-upon numbers is to choose them arbitrarily from the range of dynamic sockets and to build them into an application's configuration information. In the unlikely event that there is a conflict with another program running on the network, the network administrator can choose a different number and reconfigure the application.

To summarize this brief discussion of IPX services, following are the IPX services provided by the NWIPXSPX unit:

- determine whether IPX services are available
- get a workstation's network address
- open, close, or allocate a socket
- send or listen for a message
- cancel send or listen request
- relinquish control to IPX driver

SPX Services

NetWare implements a higher level of node-to-node communications called SPX (Sequenced Packet eXchange protocol). SPX is based on the Sequenced Packet Protocol (SPP) defined by Xerox.

Building upon the IPX services, SPX provides a system for packets of data. Packets that are not acknowledged by the recipient within a specified time are automatically retransmitted. If successive retransmissions fail, the connection is assumed to be broken. SPX automatically selects appropriate timeout and retransmission counts based on the physical characteristics of the network hardware. Hence, applications using SPX need not be concerned with the details of the guaranteed delivery system. Because SPX has more overhead than IPX, the maximum packet size is slightly smaller: 534 bytes.

Unlike IPX messages, which are received by all nodes that happen to be listening, an SPX connection transfers messages between just two nodes at a time. During the establishment of an SPX connection, one node acts as the caller and the other node acts as the listener. Once the connection is made, however, the connection is two-way; either side can send or receive.

Calls to Novell's IPX and SPX low-level routines do nothing more than submit a request for a particular message event. The actual completion of the event does not occur until an undetermined amount of time later. NetWare requires a separate event control block (of type TipxECB) to manage each pending event. Establishing an SPX connection between two stations actually requires two event control blocks on the sender's side--one for sending out the request, and another to receive confirmation from the receiver.

The receiver of SPX messages may require that even more event control blocks be available simultaneously. For example, suppose the receiver expects to receive 10 packets of information. The receiver could declare just one event control block. It would listen for each packet, process it upon arrival, and quickly make the single TipxECB available for listening again. However, this

would leave open a window when an incoming packet might not find an TipxECB available, resulting in a lost packet. A more reliable approach would be to make two event control blocks available, in order to guarantee that one is always ready while the other is temporarily tied up. In some cases, it might even be desirable to make 10 event control blocks available; in this way the entire transaction could be completed without resubmitting control blocks to SPX.

The SPX routines in the NWIPXSPX unit make it easy to deal with these situations by automatically managing a pool of event control blocks. When an application establishes an SPX connection, it simply specifies the number of event control blocks and the maximum size of each data packet. The demonstration program NSSEND (see _8.D.4) is a good example of the use of these pools and their associated functions.

Like the IPX functions in the NWIPXSPX unit, the SPX functions hide many of the details needed to use the raw NetWare routines. The NWIPXSPX functions install a real-time event service routine, manage a pool of event buffers, build event packets for you, and so on. This encapsulation makes the routines much easier to use, and also allows the same routines to work under both DOS and Windows.

The NWIPXSPX unit provides the following facilities for accessing SPX:

- determine whether SPX services are available
- establish or terminate an SPX connection
- send or listen for a message
- cancel send or listen request

The functions in the NWIPXSPX unit are written to function for all targets: real mode DOS, protected mode DOS, and Windows. When your application is in protected mode or Windows, it is essential that the buffers and control blocks passed to the routines be located in memory that is accessible in both real and protected mode because Novell's IPX/SPX driver is written for real mode. In Windows terms, this means that the memory used by these buffers must be allocated by calling GlobalDosAlloc. The NWIPXSPX unit hides these details from you by providing routines that you call to allocate control blocks (IPXAllocEventRec) and data buffers (IPXAllocPacket). In DOS programs, these routines simply call GetMem; in protected mode DOS and Windows, they call GlobalDosAlloc. Either way you must call IPXAllocEventRec and IPXAllocPacket to get these control blocks and buffers.

The program NISEND.PAS in the DEMOS directory implements a robust station-to-station file transfer program using IPX services. NSSEND.PAS is a similar program using SPX services. Another program named SPX2WAY.PAS is an interactive "chat" program that demonstrates two-way SPX communications. The following examples are extracted primarily from these programs. Descriptions of the programs are provided in _8.D.

IPX Examples

Before attempting to use IPX or SPX services, an application should ensure that these services are available. It is not essential for the sending or receiving station to be logged into a NetWare file server, although the auxiliary services provided by the file server may be useful to the application in other ways. To determine whether IPX is available, call IPXServicesAvail. Similarly, call SPXServicesAvail to check whether SPX is available.

The next task of any application using IPX or SPX is to determine the internet address of the receiver. There are at least two methods for doing this. First, if the connection number of the

receiver is known (perhaps after calling `nwGetConnNoFromName`), the internet address can be determined by calling `nwGetInternetAddress`.

It's also possible to find a receiver using an IPX broadcast. IPX defines a special address (`IPXAllNodes`) that allows any listening node on the same network to receive the message. Once a node receives this broadcast message, the sender can determine its actual internet address from a field of the IPX header structure. This algorithm is a bit too long to print here, but it is demonstrated in the `OpenSession` function in `NISEND.PAS`. In outline, `OpenSession` works as follows. The sender posts an IPX listen event on a preselected "handshake" socket. Then it enters a loop where it broadcasts a message on a different preselected socket and checks to see whether the handshake listen has received a message. Once the handshake listen is received, the sender can get the receiver's internet address from the message header. The receiver starts by posting a listen on the second socket. Then it enters a loop waiting to receive a broadcast message on this socket. Once it does so, it immediately posts an IPX message to the original sender using the handshake socket. On both sides of the transmission, the code allows for a user break from the loop and for an automatic timeout.

Given the receiver's address and an agreed-upon socket number, the following code shows how to send an IPX message.

```
const
    Socket = $4445
var
    DataBuf : PPacket;
    Event    : PipxEventRec;
    Status   : Byte;

DataBuf := IPXAllocPacket(IPXMaxDataSize);
Event   := IPXAllocEventRec(IPXDoNothingESR);

... initialize the message in DataBuf^ ...

Status := IPXSend(Event, Receiver, Socket, True, IPXMaxDataSize,
DataBuf);

IPXCloseSocket(Socket);
IPXFreeEventRec(Event);
IPXFreePacket(DataBuf);
```

`IPXMaxDataSize` specifies the maximum length of the message in this example. No single IPX message can exceed 546 bytes (unless you are running on a network that supports larger IPX packets, but in this case please note that the `NWIPXSPX` unit is configured for a 546 byte packet). `PPacket` (the type of the `DataBuf` variable) is an untyped pointer. In `NISEND.PAS`, for example, it is used as a pointer to a structure that contains fields for message type, message length, and message data. `PipxEventRec` (a variable of type `PipxEventRec` points to a variable of this type) is a structure defined in `NWIPXSPX` that encapsulates all of the information needed to send or receive an IPX message. `IPXAllocPacket` allocates space for the message buffer that can be used in either DOS or Windows applications. `IPXAllocEventRec` does the same thing for an event management buffer.

The Event data structure does not need to be initialized before passing it to `IPXSend`. The application does need to initialize the buffer pointed to by `DataBuf` to contain the desired message.

IPXSend sends the message to the specified Receiver using the specified Socket. Note that the Socket member in the Receiver structure is ignored; the socket information is taken from the Socket parameter instead. IPXSend opens the specified socket if it isn't already open.

In this example, the boolean True passed as the fourth parameter indicates that the function waits until the send event is complete. This is generally the approach to take for IPX sends: because the send does not require any acknowledgment from the receiver, the event is completed almost as soon as IPXSend is called. If False is passed in this position instead, IPXSend does not wait for the event to complete; the application can do other work until it determines the IPX event is complete (by calling IPXEventComplete).

SendIPX returns a status code indicating whether the message could be sent. If a non-zero code is returned, the caller can handle the error accordingly. Remember, however, that a successful send does not guarantee that any other station received the message. When using IPX services, such confirmation must be coded into the application itself via some kind of protocol which you design. NISEND.C shows a simple method of doing so; an alternative is to use SPX services instead. SPX not only confirms that messages are received but also performs automatic retry when messages are lost.

The code fragment ends by closing the socket and deallocating the buffers used for event management and data.

The receiver would use code like the following, which must refer to the same socket number.

```
const
    Socket = $4445;
var
    DataBuf : PPacket;
    Event   : PipxEventRec;
    Status  : Byte;

DataBuf := IPXAllocPacket(MaxPacketSize);
Event := IPXAllocEventRec(IPXDoNothingESR);

Status := IPXListen(Event, Socket, False, MaxPacketSize, DataBuf);
if (Status = 0) then
    while (not IPXEventComplete(Event, Status)) do begin
        ... check for user break ...
        ... check for automatic timeout ...
        ... perform other work ...
    end;

if (Status = 0) then
    ... use the data received in DataBuf ...

IPXCloseSocket(Socket);
IPXFreeEventRec(Event);
IPXFreePacket(DataBuf);
```

This code allocates buffers in the same way that the sending side does. The code is identical until IPXListen is called. Here a parameter of False passed in the third position indicates that IPXListen should *not* wait for a message to be received. This is almost always the preferred approach for a listen request, since there is no guarantee that a sender will ever send a message.

The code then calls IPXEventComplete in a loop waiting for a message to be received, at which time IPXEventComplete returns True. Within the loop, the application can look for a keypress from

an impatient user, check for a timeout period, or perform some other kind of useful work. Once the event is complete, the Status parameter is initialized to contain the final result code of the event. If this Status code is zero, the received message is contained in the buffer pointed to by DataBuf.

Note that IPXListen does not return the actual length of the message received; it only specifies the maximum message length the buffer can hold. If message length varies, it is the application's responsibility to encode the length within the message itself. If the length of a received message exceeds the specified maximum size, IPX stores the part of the message that will fit and returns a non-zero warning code. Another call to IPXListen will return the remainder of the message.

Important note: the receiver of IPX messages must always call IPXListen to post a listen event *before* the sender calls IPXSend. Otherwise, the message will be lost and the sender will receive no indication of the failure. NISEND.C is carefully designed so that the receiver always posts its listen events before allowing the sender to continue.

SPX Examples

SPX messages are more reliable, but have higher overhead, are slower, and are somewhat more complicated to use than IPX. As with NetBIOS sessions, the sender and receiver must establish a connection prior to exchanging messages. To do so, the two stations must agree on socket numbers, and the calling station must know the listening station's Internet address.

As with IPX, the most general method for establishing an SPX connection is too long to print here. Refer to the OpenSession function in NSEND for the details. In outline, the code works as follows. The receiver calls SPXListenForConn, using a no-wait service, to listen for the sender's attempt to establish a connection. Then the receiver goes into a loop. In this loop, it checks to see whether the SPXListenForConn event has been completed successfully. If so, the connection is made; otherwise, the receiver sends an IPX broadcast to help the sender find it. The loop continues for a specified number of seconds or until the user presses <Esc>.

The sender starts by posting an IPX listen event using a no-wait service, in order to capture the receiver's IPX broadcast. The sender then waits in a loop until the IPX message is received, the user presses <Esc>, or time runs out. If the IPX broadcast is received, the sender obtains the receiver's Internet address from the IPX message header. It immediately uses this address to call SPXEstablishConn, again in a no-wait mode. The sender then continues to loop until SPXEstablishConn succeeds or the loop times out.

Once both SPXListenForConn and SPXEstablishConn complete successfully, the connection is established and both sides have connection ID numbers to refer to for message passing. At this point the distinction between sender and receiver is arbitrary; either side can send and receive messages.

The following code fragment shows how to send SPX messages:

```
{following are used only when setting up the connection}
const
    SocketE = $4445;          {for station calling SPXEstablishConn}
    SocketL = $4446;          {for station calling SPXListenForConn}
    SocketH = $4447;          {for IPX handshake}
var
    Receiver : IPXAddress;    {address of partner}
    IPXEvent  : PipxEventRec; {used to find receiver}
    IPBuf     : PPacket;      {ditto}
```

```

{following are used for sending SPX messages}
  DataBuf : PPacket;      {send buffer}
  Event   : PspxEvtRec; {send event}

  Status : Byte;

{allocate data and event buffers}
IPXEvent = IPXAllocEventRec(IPXDoNothingESR);
IPXBuf = IPXAllocPacket(1);
Event = SPXAllocEventRec(2, SPXMaxDataSize);
DataBuf = IPXAllocPacket(SPXMaxDataSize);

... create connection with a routine like OpenSession ...

{clean up the IPX variables that are no longer required}
IPXCloseSocket(SocketH);
IPXFreeEventRec(IPXEvent);
IPXFreePacket(IPXBuf);

... initialize the message in DataBuf^ ...

{send message using a no-wait service}
Status = SPXSend(Event, False, SPXMaxDataSize, DataBuf);
if (Status = 0) then
  while (not SPXEventComplete(Event, Status)) do begin
    ... check for user break ...
    ... check for automatic timeout ...
    ... perform other work ...
  end;

{clean up}
SPXTerminateConn(Event);
SPXFreeEventRec(Event);

```

Due to the additional overhead for confirmed delivery, the maximum packet size of SPX is slightly smaller than IPX: 534 bytes.

The first several variables in this example are used only when creating the SPX connection. The first two sockets are used by the two sides of the connection for sending and receiving SPX messages, the partner that called `SPXEstablishConn` will use `SocketE`, and the other (which called `SPXListenForConn`) will use `SocketL`. These two sockets could be the same, but it is not recommended. The third socket is used for sending and receiving the initial IPX broadcast. Note that the same socket can never be used simultaneously for IPX and SPX communications.

`DataBuf` is a pointer to a buffer that is large enough to hold one message. The sender later initializes this buffer and passes its address to `SPXSend`.

`Event` is a pointer to a fairly complex structure that `NWIPXSPX` uses to manage SPX communications. This structure contains event control blocks and data buffers used to receive data and confirmation messages from the partner. The `Event` structure is allocated by calling `SPXAllocEventRec` and passing it the number of data buffers required and the maximum size of any message to be received. The number of buffers must always be at least two, even if the application will only send messages. The buffers are used internally by SPX when establishing the connection and when receiving packet confirmations. Because the example shows only the send side of a connection, it allocates the minimum number of buffers.

After the various buffers are allocated, the example must call a routine like NSSend's OpenSession, which finds the SPX partner and creates a connection. This process fully initializes the Event structure, which thereafter holds the connection ID number and other information needed to manage the connection.

The application then stores the message in the DataBuf buffer and calls SPXSend. Even though a connection is known to exist, it calls SPXSend in a no-wait mode (passing zero as the second parameter). There are two reasons why this is important. First, because SPX uses a confirmed delivery service, the send event is not complete until a confirmation is received from the partner. Hence, if the connection were broken, the sender could wait forever. Second, SPX uses internal flow control. If the receiver's buffers are full while it processes messages received previously, the sender's message is held until buffers are freed by the receiver.

The example ends by shutting down the SPX connection and freeing the buffers it allocated. SPXTerminateConn notifies the receiver that the connection is being broken. Alternatively, SPXAbortConn could be called; this function breaks the connection without any notification.

Here is the corresponding code for the receiver:

```
{following are used only when setting up the connection}
const
  SocketE = $4445;          {for station calling SPXEstablishConn}
  SocketL = $4446;          {for station calling SPXListenForConn}
  SocketH = $4447;          {for IPX handshake}
var
  Receiver : IPXAddress;    {address of partner}
  IPXEvent : PipxEventRec;  {used to find receiver}
  IPBuf    : PPacket;       {ditto}

{following are used for receiving SPX messages}
  DataBuf  : PPacket;       {receive buffer}
  Event    : PspcxEventRec; {receive event}
  DataIndex: Byte;          {index of last packet received}
  DataType : Byte;         {data type of last packet}

  Status : Byte;

{allocate data and event buffers}
IPXEvent = IPXAllocEventRec(IPXDoNothingESR);
IPXBuf = IPXAllocPacket(1);
Event = SPXAllocEventRec(10, SPXMaxDataSize);

... create connection with a routine like OpenSession ...

{clean up the IPX variables that are no longer required}
IPXCloseSocket(SocketH);
IPXFreeEventRec(IPXEvent);
IPXFreePacket(IPXBuf);

{wait for a message to arrive}
while (not SPXPacketReceived(Event, DataIndex, DataType,
                             Status, DataBuf)) do begin
  ... check for user break ...
  ... check for automatic timeout ...
  ... perform other work ...
end;
```

```

    ... use the data pointed to by DataBuf ...

    {resubmit the listen event}
    SPXReactivateECB(Event, DataIndex);

    {clean up}
    SPXTerminateConn(Event);
    SPXFreeEventRec(Event);

```

Up to the point where the receiver waits for a message, this code is very similar to that of the sender. Note that the receiver asks for a larger number of buffers when it calls `SPXAllocEventRec`. Of course this isn't important when only one message is being sent, as in this example, but it may be desirable when a steady stream of information is being sent to the receiver and the receiver must perform some processing on each message. More buffers might prevent the sender from flow-blockage while the receiver processes messages.

The receiver waits for messages by calling `SPXPacketReceived` in a loop. When a message arrives, `SPXPacketReceived` returns `True` and initializes `DataIndex`, `DataType`, `Status`, and `DataBuf`. `DataBuf` is a pointer to one of the buffers allocated when `SPXAllocEventRec` was called. This buffer contains the received message. `DataIndex` is the internal index number of this buffer; it is used later when calling `SPXReactivateECB`. `DataType` contains an internal SPX data type. It is zero for messages sent by your application. A non-zero value indicates that the connection was terminated or some other error occurred. `Status` is the completion code of the IPX event control block used for receiving this message. It is usually zero, but any non-zero value indicates an error.

At this point the application can use the message pointed to by `DataBuf`. For example, `NSSSEND` checks the application-specific type of the message and usually writes the message to a disk file. Once the application is finished with the message, it must call `SPXReactivateECB` with the index of the buffer that it is releasing for reuse. If it doesn't do so, the buffers could all become used and the sender won't be able to send any more messages.

The example ends just the same as the sender's side. Only one side's call to `SPXTerminateConn` is actually needed, but having the two calls causes no harm.

Declarations

Constants

```
IPXAllNodes; : PhysicalNodeAddress = ($FF,$FF,$FF,$FF,$FF,$FF);
```

When passed as a node address to an IPX send routine, this value indicates that the message should be sent to all nodes. Novell warns that not all hardware configurations support this special address. Even when it is supported, only nodes that are listening through the specified socket will actually receive the message. A destination address of `IPXAllNodes` is not allowed for SPX messages.

```
IPXMaxDataSize; = 546;
```

The maximum number of bytes of data in an IPX packet. More modern networks can support a larger IPX packet size (and hence a larger amount of data), but the `NWIPXSPX` unit does not provide a means of determining whether such a network is available. In addition, some inter-network routers do not support large IPX packets.

```
SPXMaxDataSize; = 534;
```

The maximum number of bytes of data in an SPX packet.

```
SPXMaxPoolCount; = 16;
```


The maximum number of event control blocks that can be actively listening for SPX messages in a 'listen pool'. Although this value can be increased, it generally does not make sense to do so.

```
SPXRetryCount; : Byte = 0;
```

The retry count for SPX operations. A value of zero indicates that SPX should use its predetermined count; any other value is used explicitly.

```
SPXWatchDog; : Boolean = False;
```

Specifies whether to activate the SPX Watchdog process. If SPXWatchDog is True, then an SPX connection uses a special monitor feature to ensure that this connection stays active. Essentially, the SPX driver sends small packets regularly to the partner to test whether the partner is still present or not. If the connection is broken, SPXPacketReceived returns True with a special packet data type.

Types

Many of the types described in this section need not be accessed directly by an application. Descriptions are included to illustrate the hierarchy of information leading to the high level types passed as parameters to the NWIPXSPX routines.

Although the data types refer to standard Pascal identifiers, you should know that NetWare reverses the byte ordering of almost all Word and LongInt fields. That is, the most significant byte is stored first instead of last. The high level routines in NWIPXSPX reverse the order as necessary, for example when storing a socket number you pass as a parameter.

```
PPIPacket; = Pointer;
```

A variable of this type is returned from IPXAllocPacket. It is a pointer to a generic buffer that is used to hold messages that are transmitted with IPX and SPX.

```
PipxHeader = ^TipxHeader;
TipxHeader; = record
    CheckSum      : Word;
    Len           : Word;
    TransportControl: Byte;
    PacketType    : Byte;
    Destination   : IPXAddress;
    Source        : IPXAddress;
end;
```

Used by IPX to manage a message. CheckSum is used internally for error control. Len is the length of the transmitted packet, including the header and data. Valid lengths range from 30 to 576. TransportControl is used internally. Destination and Source specify the internetwork addresses of the receiver and sender of the message, respectively. TipxHeader fields are managed for you by the NWIPXSPX unit and by the IPX driver itself, so you will use them for read-only purposes, if at all. The field you will reference most often is Source to find out who sent a particular message.

```
TipxFragment; = record
    Data : Pointer;
    Size : Word;
end;
```

Holds the address and size of a "fragment." A packet is a collection of fragments, which are often used to ease the receiver's job of separating the packet's components.

```
TipxECB; = record
    Link           : Pointer;
    ESRAddress     : Pointer;
    InUse          : Byte;
    CompletionCode : Byte;
    SocketNumber   : Word;
    IPXWorkspace   : LongInt;
    DriverWorkspace : array[1..12] of Byte;
    ImmediateAddress: PhysicalNodeAddress;
```

```

    FragmentCount    : Word;
    FD1              : TipxFragment;
    FD2              : TipxFragment;
end;

```

The control block for an IPX event; an event is the process of sending or listening for a message. Although the NWIPXSPX unit manages the contents of this record for you, a little description might clarify how it works.

ESRAddress holds the address of an "event service routine" (ESR). If the pointer isn't Nil, the IPX driver will call the routine it points to when an event is complete. This forms the basis for interrupt driven messaging protocols. If you want to use an event service routine, write a procedure of type IPXEventServiceRoutine and pass it to the IPXAllocEventRec routine. This takes care of setting up the ESRAddress field for you.

The InUse field is initialized to zero before submitting an event control block to IPX. IPX sets the field to a non-zero value to specify the event's status. When the field is zero again, the event is complete.

Once an event is complete, CompletionCode specifies its outcome. See the IPXEventComplete function for a list of the possible values.

SocketNumber is the socket being used for communication.

ImmediateAddress holds the address of a node within the current network. If the message originated from, or is destined for, another network, ImmediateAddress holds the address of the internetwork bridge node. The NWIPXSPX unit initializes this field when it builds an IPX event control block.

A data packet can be composed of several fragments. The first fragment must always begin with a TipxHeader. Remaining fragments can hold the actual packet data. The NWIPXSPX unit always uses two fragments, one for the header and the other for the data. The sum of the Size fields of all the individual fragments must not exceed 576 bytes.

```

IPXEventServiceRoutine; = procedure (FromAES : Boolean;
                                     IPXEvent : PipxEventRec);

```

Procedure prototype for a Pascal event service routine (ESR). An ESR is called by the IPX driver when an event completes. The FromAES boolean defines whether the event completion came from the Asynchronous Event Scheduler (AES) or not. In your programming this will normally be False. IPXEvent is a pointer to the management record whose event was completed.

Your ESR must be extremely short and fast because it is called with interrupts off (do not turn interrupts on because the code that calls your ESR is not reentrant), and it cannot call any DOS or BIOS services. Generally you will only use an ESR on listen type events, in which case you can copy the message received onto a queue and resubmit the listen event. In Windows applications, another good suggestion is to send the main application a message to say that the listen event completed.

The ESR routine is installed by passing it as a parameter to IPXAllocEventRec.

```

PipxEventRec = ^TipxEventRec;
TipxEventRec; = record
    ECB      : TipxECB;
    Header   : TipxHeader;
    Next     : PipxEventRec;
    UserESR  : IPXEventServiceRoutine;
end;

```

The high level type used by NWIPXSPX to manage IPX messages. A variable of this type must be allocated by IPXAllocEventRec by the calling program and passed to the various IPX message routines. IPXAllocEventRec initializes the fields. The ECB field describes the message itself with pointers to its contents and information about message routing and status. The Header field describes the source and destination of the message. The Next field enables the NWIPXSPX routines to form an internal linked list of TipxEventRec variables, and the UserESR field is the

event service routine that is called when the event completes. The routines in the NWIPXSPX unit initialize and maintain all fields for you.

```
PspxEvtRec = ^TspxEvtRec;
TspxEvtRec; = record
    ECB      : TipxEcb;
    Header    : TspxHeader;
    AmConnected: Boolean;
    ConnID    : Word;
    Next      : PspxEvtRec;
    PacketSize: Word;
    PoolCount : Byte;
    QueueCount: Byte;
    Pool      : array [1..SPXMaxPoolCount] of PspxPoolECB;
    Queue     : array [1..SPXMaxPoolCount] of Byte;
end;
```

The high level type used by NWIPXSPX to manage SPX messages. A variable of this type must be allocated by SPXAllocEventRec by the calling program and passed to the various SPX message routines.

The ECB and Header fields are used to establish and terminate a connection. The AmConnected field is initially False and is set to True by the NWIPXSPX unit when a connection is established, in which case the ConnID field contains the connection ID for this SPX session. When the connection is broken either by calling SPXTerminateConn or SPXAbortConn or by some other problem, the NWIPXSPX routines set the AmConnected flag to False again and clear the connection number field.

The Next field enables the NWIPXSPX unit to maintain a linked list of TspxEvtRecs. The PacketSize field is the buffer size that you passed to the SPXAllocEventRec routines. The PoolCount and QueueCount fields are the numbers of listen buffers in the pool and the number of listen buffers that have received a message and are awaiting processing. The Pool field is the array of listen buffers for this event management record. The Queue field is an array of indexes of listen buffers that are awaiting processing.

The routines in the NWIPXSPX unit initialize and maintain the values in the TspxEvtRec, you should not alter any values yourself.

```
PspxHeader = ^TspxHeader;
TspxHeader; = record
    IPXHeader      : TipxHeader;
    ConnectControl : Byte;
    DataStreamType : Byte;
    SourceConnID   : Word;
    DestConnID     : Word;
    SequenceNo     : Word;
    AcknowledgeNo  : Word;
    AllocationNo   : Word;
end;
```

Used by SPX to manage a message. All fields are maintained internally by the NWIPXSPX unit and the SPX driver.

```
PspxPoolECB = ^TspxPoolECB;
TspxPoolECB; = record
    PoolECB : TipxEcb;
    PoolHeader : TspxHeader;
    PoolData : Array[1..SPXMaxDataSize] of Byte;
end;
```

Used by the NWIPXSPX unit to manage a pool of listen buffers for an SPX connection. A single listen buffer is a variable of this type; the TspxEvtRec variable allocated by SPXAllocEventRec tracks an array of these buffers. The PoolData field is just large enough to contain a buffer of the size specified during a call to SPXAllocEventRec. The PoolECB field is

an IPX event control block that listens for incoming SPX messages, the PoolHeader field is an SPX header defining the source and destination of the SPX message.

```
TspStatus; = record
  State                : Byte;
  Flag                 : Byte;
  SourceConn           : Word;
  DestConn             : Word;
  SequenceNum          : Word;
  AckNum               : Word;
  AllocNum             : Word;
  RemoteAckNum         : Word;
  RemoteAllocNum       : Word;
  ConnSocket           : Word;
  ImmediateAdd         : PhysicalNodeAddress;
  Destination          : IPXAddress;
  RetransmitCount      : Word;
  EstimatedRoundTripDelay : Word;
  RetransmittedPackets : Word;
  SuppressedPackets    : Word;
end;
```

The data structure returned by SPXGetConnStatus. The most important result returned by this function is its return code, which indicates whether the connection still exists. The information in the structure itself may be useful in rare situations. NWIPXSPX swaps byte ordering as necessary to bring the structure to PC standards before returning the structure. Here is a brief summary of the fields.

State can take on four values: 1 - waiting to receive an establish connection packet; 2 - attempting to establish a connection by sending establish packets; 3 - connection established; 4 - the remote has terminated its half of the connection.

If bit 1 of Flag is set (i.e. (Flag and \$02) <> 0), the SPX watchdog is monitoring the connection. The SPX watchdog watches for connections that aren't used for a long period of time. The other bits of this field are used internally by SPX.

SourceConn and DestConn are the connection ID numbers of the local and remote nodes respectively.

SequenceNum and AckNum are packet numbers assigned internally by SPX. These numbers increase sequentially from zero and wrap back to zero when they reach 0xFFFF. SequenceNum is the number of the next packet the local station will send. AckNum is the number of the next packet the local station expects to receive.

AllocNum is the highest packet number the local station can receive given the number of available listening IPX event control blocks. This number is increased as the local station adds (or reactivates) control blocks in the listen pool. SPX uses this field to implement flow control.

RemoteAckNum and RemoteAllocNum are analogous to AckNum and AllocNum, but they refer to the packet numbers the remote station expects to receive from the local station.

ConnSocket is the socket number the local station is using to send and receive SPX packets. ImmediateAdd is the address of a node on the current network. If the entire connection is on one network, ImmediateAdd is the remote station. Otherwise, it is the address of a bridge device. Destination is the complete address of the remote station, including the socket number that it is using to send and receive SPX packets.

RetransmitCount indicates the number of times since SPX was loaded that SPX has attempted to retransmit an unacknowledged packet before determining that the remote station has become unreachable. EstimatedRoundTripDelay is the number of clock ticks that SPX waits before assuming that a packet is lost and retransmitting. RetransmittedPackets is the number of retransmitted packets that were acknowledged by the remote station. SuppressedPackets is the number of packets the local station has discarded because they were duplicates of previously received packets or because they were outside of the allowable packet number range.

IPXAllocateEventRec / IPXAllocPacket

Syntax

```
function IPXAllocateEventRec; (ESR : IPXEventServiceRoutine) : TipxEventRec;
```

Purpose

Allocate and return a pointer to a TipxEventRec.

Description

You must allocate an IPX event record (TipxEventRec) prior to sending or receiving an IPX message. The TipxEventRec variable must remain accessible to the IPX driver until the operation is complete.

In protected mode environments, the TipxEventRec is allocated using GlobalDOSAlloc to ensure that it is accessible from both the real and protected modes of the CPU. This is necessary because the IPX driver is written for DOS real mode operation. IPXAllocEventRec returns a protected mode pointer (in selector:offset format) that you can dereference in your application without any conversion. The NWIPXSPX unit converts it internally to a real mode pointer (segment:offset format) prior to passing control to the IPX driver.

IPXAllocEventRec returns Nil if there is insufficient free memory to satisfy the request.

See Also

IPXAllocPacket

IPXFreeEventRec

Syntax

```
function IPXAllocPacket; (Size : Word) : PPacket;
```

Purpose

Allocate and return a pointer to a data packet buffer.

Description

All buffers that are used for message transmission via IPX or SPX must be allocated with this routine.

In protected mode, IPXAllocPacket ensures that the buffer is in real mode addressable memory. This is necessary because the IPX driver is written for DOS real mode operation.

IPXAllocPacket returns a protected mode pointer (in selector:offset format) that you can dereference in your application without any conversion.

IPXAllocPacket returns Nil if there is insufficient free memory to satisfy the request.

See Also

IPXAllocEventRec

IPXFreePacket

IPXCancelEvent / IPXCloseSocket

Syntax

```
function IPXCancelEvent, (IPXEvent : PipxEventRec) : Byte;
```

Purpose

Cancel any pending IPX event.

Description

After an event is successfully cancelled, the application can reuse the event variable IPXEvent. If you call IPXEventComplete after cancelling an event, it returns True and a CompletionCode of \$FC.

A status code is returned in the function result:

```
$00 Success.  
$F9 Event was active but couldn't cancel it.  
$FF Event was not active.
```

Error \$F9 occurs if the IPX driver is in the process of transferring data to or from the fragment buffers of the packet when the cancellation request occurs. When cancelling a send event, there is no guarantee that the intended receiver won't receive the packet anyway.

See Also

IPXEventComplete

Syntax

```
procedure IPXCloseSocket; (Socket : Word);
```

Purpose

Close the specified socket.

Description

Closing a socket automatically cancels any pending messages that are associated with the socket. Any further incoming packets are ignored. Attempting to close a socket that is not open has no effect.

Each workstation can have no more than 20 sockets open at a time (the limit is configurable via NET.CFG, however). For some applications, this may mean that it's a good idea to close each socket as soon as its event is complete.

Transient (non-TSR) applications must close all their sockets before returning to DOS. Even if the sockets were opened with the Forever flag set to False, there exists a small window of time after the application has halted and before the IPX manager can close the socket when an incoming packet might cause a system crash. It is best to define an exit procedure that closes all open sockets no matter how the program terminates.

If you are using event service routines, note that IPXCloseSocket *cannot* be called within the service routine.

See Also

IPXOpenSocket

IPXOpenUniqueSocket

IPXEventComplete / IPXFreeEventRec / IPXFreePacket

Syntax

```
function IPXEventComplete;(IPXEvent : PipxEventRec;  
                           var CompletionCode : Byte) : Boolean;
```

Purpose

Return the status of a pending IPX event.

Description

IPXEvent is the event to check. When IPXEventComplete returns True, the event is complete. In this case, CompletionCode returns the status of the event. If IPXEventComplete returns False, the value of CompletionCode is undefined.

Call IPXEventComplete after submitting an IPX event with IPXSend or IPXListen. If either function was called with the WaitForCompletion flag set to True, then IPXEventComplete is guaranteed to return True on the first call. Otherwise, the program should poll IPXEventComplete until it returns True. In the meantime, the application can continue to do other work.

Each call to IPXEventComplete generates a call to IPXRelinquish, which gives the IPX driver an opportunity to service pending events. If the application spends a significant amount of time between calls to IPXEventComplete, it should call IPXRelinquish regularly to assure that incoming packets are detected.

CompletionCode can return several values, including:

```
$00 Packet was sent or received successfully.  
$FC Send or listen request was cancelled.  
$FD Malformed packet (too small, too large, incorrect fragment size  
    or count).  
$FE Packet is undeliverable (destination cannot be detected, no  
    internetwork bridge is available, or there has been a hardware  
    failure).  
$FF Unable to send packet (there has been a hardware or network  
    failure).
```

See Also

IPXListen
IPXSend
IPXRelinquish

Syntax

```
procedure IPXFreeEventRec;(IPXEvent : PipxEventRec);
```

Purpose

Free an event record allocated by IPXAllocEventRec.

Description

All event records allocated using IPXAllocEventRec should be freed by calling IPXFreeEventRec. Otherwise heap space that is no longer in use remains allocated.

See Also

IPXAllocEventRec

Syntax

```
procedure IPXFreePacket;(P : PPacket);
```

Purpose

Free a packet buffer allocated by IPXAllocPacket.

IPXEventComplete / IPXFreeEventRec / IPXFreePacket

Description

~~All packet buffers allocated using IPXAllocPacket should be freed by calling IPXFreePacket.~~
Otherwise heap space that is no longer in use remains allocated.

See Also

IPXAllocPacket

IPXInternetAddress / IPXListen

Syntax

```
procedure IPXInternetAddress; (var Address : IPXAddress);
```

Purpose

Return the internetwork address of the calling workstation.

Description

The network number and physical node address of the caller are returned by this function. The network number is returned in hi-lo order, which is the format expected by other functions to which the address is passed. The third member of the IPXAddress type, the Socket, is not initialized by this call. The functions to which an IPXAddress variable is passed do not expect Socket to be initialized.

See Also

nwGetInternetAddress (NWCONN)

Syntax

```
function IPXListen; (IPXEvent : PipxEventRec; Socket : Word;  
                    WaitForCompletion : Boolean; MaxPacketSize : Word;  
                    DataPacket : PPacket) : Byte;
```

Purpose

Initialize IPX data structures to receive a packet, and submit the event to IPX.

Description

The structure pointed to by IPXEvent is initialized by IPXListen; it maintains all the information required for managing the message. The caller must specify a Socket number for the transmission. Sender and receiver must agree on a socket number. See IPXOpenSocket for details. IPXListen will open the socket if it isn't already open.

When WaitForCompletion is False, IPXListen returns immediately after submitting the request to listen. In this case, the calling application can continue with other tasks, but it must poll the IPXEventComplete function to determine when the event is complete. If WaitForCompletion is True, IPXListen itself polls until the event is complete. However this is not a good idea in a Windows environment (no other tasks will be able to get control during this period) and is not recommended anyway (if no other station is sending you an IPX packet, you'll essentially hang your workstation). The socket is not automatically closed upon completion of the listen event. The completion code returned by IPXEventComplete indicates whether a message was actually received.

MaxPacketSize specifies the maximum number of bytes to receive, not including the header information managed internally by IPXListen. The largest allowable packet size is 546 bytes (IPXMaxDataSize). DataPacket is a pointer to a variable of any type, as long as the variable is large enough to hold at least the number of bytes specified by MaxPacketSize. This DataPacket buffer must have been allocated by IPXAllocPacket.

Both IPXEvent and DataPacket must be pointers to variables that remain in scope throughout the duration of the event. In other words, do not declare these pointers as local variables and exit the routine before the event is complete.

A status code is returned in function result:

```
$00 Success.  
$01 MaxPacketSize too large.  
$FE Socket table full.
```

See Also

IPXCloseSocket
IPXOpenSocket

IPXEventComplete
IPXSend

IPXOpenSocket / IPXOpenUniqueSocket

Syntax

```
function IPXOpenSocket; (Socket : Word; Forever : Boolean) : Byte;
```

Purpose

Open a socket for use by IPX or SPX.

Description

Sender and receiver must agree on a socket number in order to communicate. Sockets numbered from \$4000 to \$7FFF are available for use by any application. A workstation can have up to 20 sockets open at one time (this is configurable via NET.CFG).

When Forever is True, the socket remains open until explicitly closed, even after the current program terminates. Otherwise, the IPX driver closes it automatically when the program ends. Setting Forever True would be useful for a TSR that requires IPX services.

The IPXSend and IPXListen routines open the requested socket when they are called, if necessary. Thus IPXOpenSocket might be most useful for determining that a socket *can* be opened before an application proceeds too far.

Even though the IPX driver automatically closes sockets when Forever is set False, it is still recommended that the application explicitly close its own sockets. See IPXCloseSocket for more information.

A status code is returned in function result:

```
$00 Success.  
$FE Socket table is full.  
$FF Socket already open.
```

See Also

IPXCloseSocket

IPXOpenUniqueSocket

Syntax

```
function IPXOpenUniqueSocket; (var Socket : Word; Forever : Boolean) :  
Byte;
```

Purpose

Open and return a unique socket number.

Description

This function works like IPXOpenSocket, but it returns a socket number instead of having one passed to it. The returned socket is any valid available socket found by the IPX driver. The difficulty in using this routine is that the socket number must usually be communicated to the receiving partner using other means.

A status code is returned in function result:

```
$00 Success.  
$FE Socket table is full.
```

See Also

IPXCloseSocket

IPXOpenSocket

IPXRelinquish

Syntax

```
procedure IPXRelinquish;;
```

Purpose

Temporarily relinquish application control to the IPX driver.

Description

This procedure serves one of two purposes, depending on whether it is invoked from a non-dedicated file server or a workstation. On a server, this function temporarily suspends the calling process so that the server program gets CPU resources immediately. Similarly, on a workstation, the NetWare shell gets temporary control during which it can check for incoming and outgoing messages.

This procedure should be called frequently while an application waits for an IPX or SPX event to complete. IPXEventComplete automatically calls IPXRelinquish. However, if an application performs significant processing between calls to IPXEventComplete, it is wise to call IPXRelinquish regularly so that the driver can process incoming messages.

See Also

IPXEventComplete

IPXSend

Syntax

```
function IPXSend; (IPXEvent : PIPXEventRec; Receiver : IPXAddress;  
Socket : Word;  
WaitForCompletion : Boolean; DataPacketSize : Word;  
DataPacket : PPacket) : Byte;
```

Purpose

Initialize IPX data structures to send a packet, and submit the event to IPX.

Description

The event record pointed to by IPXEvent is initialized by IPXSend and maintains all the information required for managing the message. The IPXEvent variable must be obtained by calling IPXAllocEventRec.

The caller must initialize Receiver to specify where the packet should go. See nwGetInternetAddress in _8.A.2 for one method to determine Receiver's address. See the demonstration program NISEND.PAS for another method. The caller must also specify a Socket number for the transmission. Sender and receiver must agree on a socket number. See IPXOpenSocket for details. IPXSend opens the socket if it isn't already open.

When WaitForCompletion is False, IPXSend returns immediately after submitting the request to send. In this case, the calling application can continue with other tasks, but it must poll the IPXEventComplete function to determine when the event is complete. If WaitForCompletion is True, IPXSend itself polls until the event is complete. Because IPX messages are not confirmed by the receiver, it's generally fast and safe to pass True to allow the send to finish before IPXSend returns. If WaitForCompletion is True, the status of the send is returned in the function result; otherwise call IPXEventComplete to determine the status.

The socket is not automatically closed upon completion of the send event. See IPXCloseSocket.

DataPacketSize specifies the number of bytes to send, not including the header information managed internally by IPXSend. The largest allowable packet size is 546 bytes. DataPacket is a pointer to a variable of any type, as long as the variable holds at least the number of bytes specified by DataPacketSize. The DataPacket buffer must be obtained by calling IPXAllocPacket.

Both IPXEvent and DataPacket must be pointers to variables that remain in scope throughout the duration of the event. In other words, do not declare these pointers as local variables and exit the routine before the event is complete.

A status code is returned in function result:

```
$00 Success.  
$01 DataPacketSize too large.  
$FE Socket table full.
```

See Also

IPXCloseSocket
IPXListen

IPXEventComplete
IPXOpenSocket

IPXServicesAvail / SPXAbortConn

Syntax

```
function IPXServicesAvail; : Boolean;
```

Purpose

Determine whether IPX services are available. *Call this first!*

Description

You *must* call this routine before using any of the IPS or SPX functions described in this section. When this routine returns True, it also initializes a pointer that is used to make calls to the IPX driver.

If IPXServicesAvail returns False, no IPX driver is available and you must not call any other IPX or SPX routines.

IPX services may be available even if the NetWare shell is not loaded or the user is not logged in to a file server.

Example

```
if IPXServicesAvail then begin
  if IPXOpenSocket($7001, True) = 0 then begin
    {Perform IPX communications}
  end else
    {Bad socket} ;
end else
  {No IPX driver loaded} ;
```

Tests for IPX services before using them.

See Also

SPXServicesAvail

Syntax

```
procedure SPXAbortConn; (SPXEvent : PspcxEventRec);
```

Purpose

Abruptly terminate an existing SPX connection.

Description

SPXAbortConn does not notify the connection partner of the decision to break the connection. The partner will eventually discover that the connection is no longer valid when it sends a packet or attempts to terminate the connection. If the connection was established with the SPX Watchdog feature (see the SPXWatchdog typed constant earlier in this section), the partner will receive a failed connection message the next time the monitor tests the connection.

This routine is designed to unilaterally break a connection when a serious error condition is encountered. Under most circumstances, SPXTerminateConn should be called instead.

See Also

SPXEstablishConn

SPXTerminateConn

SPXAllocEventRec

Syntax

```
function SPXAllocEventRec (NumECBs : Byte; MaxPacketSize : Word) :  
    PspxEvtRec;
```

Purpose

Allocate an SPX event record and associated buffers.

Description

This function allocates both the event control buffers and the data buffers for an SPX connection, unlike NetBIOS and IPX services where you must call separate functions to allocate the two types of buffers. This behavior is appropriate for SPX communications, because SPX requires multiple buffers which are filled asynchronously and queued up for the application to use in a sequential fashion. The buffers are sometimes used by SPX for receiving packet confirmation messages. Although the application never sees these messages, buffers must still be available for their reception. The buffers are also used during a handshaking process that occurs when a connection is first established. The PspxEvtRec returned by SPXAllocEventRec points to a structure that manages this complex SPX behavior automatically.

NumECBs specifies the number of message buffers, each of size MaxPacketSize bytes, allocated for the SPX event record. NumECBs must always be at least equal to two. NumECBs can be no more than SPXMaxPoolCount, which defaults to 16. Because SPX has automatic flow control (it won't overflow the buffers regardless of how many you specify within this range), the choice of NumECBs is primarily a matter of performance. If the receiver of messages has more processing to do than the sender, it may make sense to give the receiver a larger number of buffers. If the sender and receiver perform roughly equal amounts of processing, there is no need to use more than two buffers for either side.

MaxPacketSize must be in the range from 1 to SPXMaxDataSize (534).

If NumECBs or MaxPacketSize is outside of the valid range, or if there is insufficient memory to allocate the structure, SPXAllocEventRec returns NULL.

SPXAllocEventRec calls IPXAllocPacket to allocate its memory. As a result, the pointers are valid for both DOS and Windows applications. The value returned in a protected mode DOS or Windows application is a protected mode pointer that can be used without any conversion.

See Also

IPXAllocPacket

SPXFreeEventRec

SPXCancelListenForConn / SPXECBsListening

Syntax

```
procedure SPXCancelListenForConn; (SPXEvent : PspcxEventRec);
```

Purpose

Cancel an SPXListenForConn event.

Description

An SPX connection number is only allocated for a workstation calling SPXListenForConn when that event completes normally. Contrast this with SPXEstablishConn where the connection number is allocated immediately. Hence to cancel an SPXListenForConn event, you cannot call SPXTerminateConn or SPXAbortConn because they require a connection number in the event record. Instead, you can call SPXCancelListenForConn.

See Also

SPXAbortConn	SPXListenForConn
SPXTerminateConn	

Syntax

```
function SPXECBsListening; (SPXEvent : PspcxEventRec) : Byte;
```

Purpose

Determine how many listen buffers are actively listening.

Description

This function returns a number in the range from zero to NumECBs (as passed to SPXAllocEventRec). If no buffers are listening, all of the connection's data buffers are already filled with incoming data and the sending side of the connection is blocked from sending further messages. If all buffers are listening, no received messages are pending.

The primary use of this function is for tuning the number of buffers to improve performance.

See Also

SPXAllocEventRec	SPXGetConnStatus
SPXReactivateECB	

SPXEstablishConn

Syntax

```
function SPXEstablishConn, (SPXEvent : PipxEventRec; Receiver :  
    IPXAddress;  
                           LocalSocket : Word; RemoteSocket : Word;  
                           WaitForCompletion : Boolean) : Byte;
```

Purpose

Establish an SPX connection with a listening partner.

Description

An initial message is sent to the partner's internetwork address in an attempt to create an SPX connection. If the partner responds correctly, then a communication channel is established. The partner must call SPXListenForConn in order to establish its side of the link.

SPXEstablishConn uses the SPXEvent parameter as a work area, similar to the way that file operations use a file variable. You must allocate the buffers pointed to by SPXEvent by calling SPXAllocEventRec before calling SPXEstablishConn, but you don't need to initialize the buffers. SPXEvent must then remain continuously accessible as long as the SPX connection is active.

The caller must initialize Receiver to specify the partner's address. See nwGetInternetAddress in 8.A.2 for one method to determine this address. See the demonstration program NSSEND.PAS for another method. The caller must also specify two socket numbers for the connection: the first socket is used by the local program (LocalSocket) to communicate with its SPX driver and the other is used by the remote program to communicate with *its* SPX driver. These two socket numbers must be different. The sender and receiver must agree on the two socket numbers. See IPXOpenSocket for information regarding socket numbers. SPXEstablishConn opens the local socket if it isn't already open.

When WaitForCompletion is False, SPXEstablishConn returns immediately after submitting the request to establish a connection. In this case, the calling application can continue with other tasks, but it must poll the SPXEventComplete function to determine when the event is complete. If WaitForCompletion is True, SPXEstablishConn itself polls until the event is complete.

Generally, WaitForCompletion should be False when calling SPXEstablishConn, since there's no guarantee that another station will be listening for the connection.

The SPX connection ID is allocated immediately by this function and is stored in the SPXEvent variable. Once this event completes, there is no distinction between the station that established the connection and the station that listened for it. Either side can send or receive messages. See the demonstration program SPX2WAY.PAS for an example.

A status code is returned in the function result:

```
$00 Success.  
$EF Local connection table is full.  
$FD Malformed packet (NETWARE internal error).  
$FE The socket table is full.
```

Call SPXTerminateConn to cancel an SPXEstablishConn call that's in progress.

See Also

nwGetInternetAddress (NWCONN)	SPXEventComplete
SPXListenForConn	SPXSend
SPXTerminateConn	

SPXEventComplete / SPXFreeEventRec

Syntax

```
function SPXEventComplete; (SPXEvent : PspcxEventRec;  
                           var CompletionCode : Byte) : Boolean;
```

Purpose

Poll the status of an SPX event to determine when the event is complete.

Description

This routine is necessary because SPX messaging services are asynchronous. When an application calls SPXEstablishConn, SPXListenForConn, or SPXSend and the WaitForCompletion parameter is False, SPX submits a message header to a queue and returns immediately.

Only later, when the message has made its way to the receiver and a confirmation is received, is the event considered complete. At that time, the caller can check the outcome, e.g., whether the connection was established or the message was successfully sent. SPXEventComplete returns True when the specified event is complete; False otherwise. It is never safe to call one of these functions using the same SPXEvent variable until the previous event is complete.

When SPXEventComplete returns True, CompletionCode returns a status code:

```
$00 Event completed successfully.  
$EC Connection terminated by remote partner.  
$ED Target failed to answer.  
$EE Invalid connection ID in SPXEvent variable.  
$EF Local connection table is full.  
$FC Event cancelled.  
$FD Malformed packet. (too small, too large, incorrect fragment  
size or count)  
$FE The socket table is full.  
$FF Underlying socket not open. (NETWARE internal error)
```

Each call to SPXEventComplete generates a call to IPXRelinquish, which gives the IPX driver an opportunity to service pending IPX and SPX events. If the application spends a significant amount of time between calls to SPXEventComplete, it should call IPXRelinquish regularly to assure that incoming packets are detected.

If SPXEventComplete determines that the connection was broken (i.e., the completion code is \$EC), the event record is cleared after canceling all listen events.

See Also

IPXEventComplete

IPXRelinquish

Syntax

```
procedure SPXFreeEventRec; (SPXEvent : PspcxEventRec);
```

Purpose

Free an SPX event record and associated buffers.

Description

All SPX event records allocated using SPXAllocEventRec should be freed by calling SPXFreeEventRec. Otherwise heap space that is no longer in use remains allocated. For protected mode DOS and Windows applications, a DPMI callback and selector resource would also remain allocated.

See Also

SPXAllocEventRec

SPXGetConnStatus / SPXListenForConn

Syntax

```
function SPXGetConnStatus; (SPXEvent : PspcxEventRec;  
                           var ConnStatus : TSPXStatus) : Byte;
```

Purpose

Return the status of an existing connection.

Description

This function can be called after calling SPXEstablishConn, even if it has not completed successfully. SPXGetConnStatus can also be called after a connection is established. SPXGetConnStatus cannot be called immediately after SPXListenForConn. You must wait until SPXListenForConn completes and the connection is set up.

SPXGetConnStatus returns zero if the SPXEvent refers to a valid SPX connection. When the function returns zero, the structure pointed to by ConnStatus is initialized to contain information about the connection. See the description of TspcxStatus earlier in this section for details.

A status code is returned in the function result:

```
0x00 Success.  
0xEE No such connection.
```

Syntax

```
function SPXListenForConn; (SPXEvent : PspcxEventRec; LocalSocket :  
                           Word;  
                           WaitForCompletion : Boolean) : Byte;
```

Purpose

Listen for an SPX connection from a sending partner.

Description

This is the counterpart of the SPXEstablishConn function. To establish a connection, one side must listen with SPXListenForConn, and the other side must call with SPXEstablishConn.

SPXListenForConn uses the SPXEvent parameter as a work area, similar to the way that file operations use a file variable. You must allocate the buffers pointed to by SPXEvent by calling SPXAllocEventRec before calling SPXListenForConn, but you don't need to initialize the buffers. SPXEvent must remain continuously accessible as long as the SPX connection is active.

You must specify a socket number (LocalSocket) for the connection. This socket is used for communicating with the local SPX driver, and must be the same socket that the partner specifies in the RemoteSocket parameter to SPXEstablishConn. See IPXOpenSocket for information regarding socket numbers. SPXListenForConn opens the specified socket if it isn't already open.

When WaitForCompletion is False, SPXListenForConn returns immediately after submitting the request to establish a connection. In this case, the calling application can continue with other tasks, but it must poll the SPXEventComplete function to determine when the event is complete.

If WaitForCompletion is True, SPXListenForConn itself polls until the event is complete.

Generally, WaitForCompletion should be False when calling SPXListenForConn, since there's no guarantee that another station will be attempting to establish the connection.

If you pass True for WaitForCompletion and SPXListenForConn returns zero (i.e. successful), then it automatically stores the SPX connection ID in the SPXEvent variable in the ConnID field. If you pass False so that SPXListenForConn returns immediately, a subsequent successful call to SPXEventComplete stores the connection ID in the same field.

Call SPXCancelListenForConn to cancel an SPXListenForConn call that is in progress. See the demonstration programs NSSEND.PAS and SPX2WAY.PAS for further examples.

A status code is returned in the function result:

```
$00 Event submitted successfully.  
$EF Local connection table is full.  
$FE The socket table is full.
```

SPXGetConnStatus / SPXListenForConn

See Also

[SPXCancelListenForConn](#)

[SPXEstablishConn](#)

SPXPacketReceived / SPXReactivateECB

Syntax

```
function SPXPacketReceived, (SPXEvent : PspXEventRec; var Index :  
    Byte;  
                                var DataType : Byte; var CompletionCode :  
                                Byte;  
                                var DataPtr : Pointer) : Boolean;
```

Purpose

Return True if an SPX packet was received.

Description

An application that is waiting to receive SPX messages must call this routine in a loop. SPXPacketReceived checks the internal queue of received messages and returns True when the queue is not empty. The queue is updated in the background by an SPX event service routine.

When SPXPacketReceived returns True, the Index parameter is set to the internal index number of the listen buffer holding the received message. Your application doesn't need to use this number except to pass it to SPXReactivateECB, which indicates that the application has processed the message and the listen buffer can be reused by SPX.

The variable pointed to by DataType is also initialized when a message is received. DataType always contains the value zero for messages sent by your application. Non-zero values indicate a message sent by SPX itself. The most common non-zero value is 0xFE, which is a packet used by SPX to indicate end of connection. The packet is generated when the other side of a connection has called SPXTerminateConn.

The variable pointed to by CompletionCode holds the final status of the IPX event record used to manage this particular packet. Any non-zero value indicates that there was an error receiving the packet. (This should occur only in rare situations since SPX performs its own error detection and automatic retry.)

The variable pointed to by DataPtr gets a pointer to the actual data buffer. Data buffers were allocated when SPXAllocEventRec was called, and DataPtr returns one of these pointers. You can pass the address of any pointer in this position, and then use that pointer to process the received message.

Example

See NSSEND.PAS and SPX2WAY.PAS for examples.

See Also

SPXReactivateECB

Syntax

```
procedure SPXReactivateECB; (SPXEvent : PspXEventRec; Index : Byte);
```

Purpose

Reactivate the specified listen buffer.

Description

After an application receives a message (via SPXPacketReceived) and completely processes the message, it must call SPXReactivateECB to inform SPX that the listen buffer can be reused. If it doesn't, the listen buffers will eventually be completely used, and the sender will not be able to send additional messages. SPXReactivateECB also updates the internal queue structures used in the SPXEvent variable.

The Index parameter must be the same value that was returned by SPXPacketReceived. If Index is outside of the valid range of indexes, SPXReactivateECB does nothing.

SPXSend / SPXServicesAvail

Syntax

```
function SPXSend; (SPXEvent : PspcxEventRec; WaitForCompletion : _____  
Boolean;  
DataPacketSize : Word; DataPacket : PPacket) : Byte;
```

Purpose

Send a data packet using SPX services.

Description

SPXEvent must have been allocated previously by a call to SPXAllocEventRec, and an SPX connection must have been established by a successful, complete call to SPXEstablishConn or SPXListenForConn. The DataPacket pointer must have been allocated by calling IPXAllocPacket.

The packet is any user-defined data type up to 534 bytes in size. Pass the size of the packet as the DataPacketSize parameter. The actual length of the message is not known to the receiver, so if packets of varying sizes are sent, the application should store the packet length somewhere within the packet itself. The packet length should not exceed the maximum size specified by the receiver when it called SPXAllocEventRec.

The SPXSend event is asynchronous in the same sense that establishing the connection is. Therefore, if WaitForCompletion is False, an SPXSend must be followed by repeated calls to SPXEventComplete. Similarly, the DataPacket buffer must remain static until the SPXSend is complete.

See Also

SPXEstablishConn

Syntax

```
function SPXServicesAvail; (var Version : Word; var MaxSPXConn : Word;  
var AvailSPXConn : Word) : Boolean;
```

Purpose

Determine if SPX services are available and, if so, return version information. *Call this first!*

Description

You *must* call this function before using any of the SPX functions described in this section. SPXServicesAvail internally calls IPXServicesAvail, which initializes an internal pointer that is used to make calls to the IPX driver.

SPXServicesAvail returns True if SPX is available. In this case it also returns SPXVersion, which is the SPX version number with the major part in the high byte. MaxSPXConn is the maximum number of SPX connections possible. AvailSPXConn is the currently available number of SPX connections.

Example

```
if not SPXServicesAvail(Version, MaxConn, AvailConn) then begin  
  Writeln('SPX not available'); Halt;  
end;  
Writeln('SPX version ', Version shr 8, '.', Version and $FF);  
Writeln('Connections ', MaxConn, ' maximum, ', AvailConn, '  
available');
```

Checks for SPX and reports on SPX parameters if available.

See Also

IPXServicesAvail

SPXTerminateConn

Syntax

```
procedure SPXTerminateConn, (SPXEvent : PSpXEventRec);
```

Purpose

Terminate an existing SPX connection.

Description

SPXTerminateConn sends a packet to the other end of the connection indicating that the connection should be broken. SPX automates the receiver's acknowledgment of the message and also completes the disconnection process. The associated connection ID is returned to the free pool.

SPXTerminateConn internally waits for the local SPX driver to report that the connection was successfully terminated. Then the routine closes the local socket that was being used for communicating to the SPX driver and clears out the relevant fields in the SPXEvent variable. On return from SPXTerminateConn, the SPXEvent variable can either be reused to establish another connection, or can be freed by calling SPXFreeEventRec.

No harm is done if both sides of a connection call SPXTerminateConn. Only one of the sides will succeed in this case, the other will receive an error code which SPXTerminateConn takes to mean that the connection is broken.

See Also

SPXAbortConn

SPXEstablishConn

NetBIOS stands for Network Basic Input/Output System. It is an application programming interface (API) that supports data exchange between workstations on a network. NetBIOS was introduced by IBM in 1984, and is in widespread use today. It has become an industry standard protocol and has been implemented by most modern microcomputer network operating systems. As such, a program written using NetBIOS services will run unmodified on a wide variety of networks.

An extended version of NetBIOS was introduced by IBM in 1985 and is called NetBEUI (NetBIOS Extended User Interface). The NETBIOS unit can be used with NetBEUI also, but it does not support the extended functions of NetBEUI.

The NETBIOS unit isolates the programmer from the details of issuing the NetBIOS function calls. It is designed to make writing NetBIOS-compatible Pascal programs easy. (Note that NETBIOS in uppercase characters refers to the unit supplied with B-Tree Filer; the mixed case spelling NetBIOS refers to the operating system extension itself.)

Unlike the routines in the NetWare units, the NETBIOS unit's routines are largely independent of the network operating system. In other words, they will work on any network that implements NetBIOS compatible functions. (Novell supplies a NetBIOS emulator with all versions of NetWare. Note, however, that each workstation must load additional programs--typically NETBIOS.EXE and possibly INT2F.COM--to activate the emulator.)

In contrast to NetWare, the NetBIOS interface is compact and focused only on station to station communications. The complete NetBIOS API defines only 19 functions, and the NETBIOS unit implements and documents almost all of them.

A fundamental part of NetBIOS programming is the concept of a "name." Each workstation can have up to 17 names, each 16 bytes long. One of these names is the "permanent node name." The permanent node name is the physical network adapter card's own unique signature. It is programmed into the hardware and cannot be changed by the application. Because this name always exists, it is sometimes convenient to refer to it during other NetBIOS calls.

Each station's adapter card (or NetBIOS emulator) also supports a "local name table" that holds up to 16 software-selectable names. Each can be a "unique name," which the station reserves for its exclusive use on the network, or a "group name," which other stations can share. Case is significant when comparing NetBIOS names. When a name is added to the table, the NetBIOS broadcasts its intentions to claim the name to all other stations on the network. If a unique name was requested, and another station already has stored the same name in its local table, the request fails. When a name is successfully added to the name table, NetBIOS returns the position in the table where it resides. This "name number" is used by many NetBIOS commands as a quick way of referring to a name that's known to be in the table.

The simplest way you can use NetBIOS to communicate from station to station is with a "." A datagram is just a small block of raw data. The maximum size of a datagram is implementation specific, but the most common limit is 512 bytes. A datagram message can be sent to a unique name, a group name, or everyone on the network (with a broadcast datagram). Datagrams impose very little overhead, and are quite easy to use. However, like NetWare's IPX services, datagrams aren't automatically acknowledged by the receiver's NetBIOS and they are lost if the destination isn't ready to receive them.

A second, more robust, NetBIOS protocol is called a "session." Session packets are automatically acknowledged upon receipt, providing for guaranteed delivery of the data. Sessions can also transfer larger chunks of data--up to 64KB, or even larger by using advanced techniques. Of course, with this reliability comes extra overhead. Another limitation of sessions relative to datagrams is that they are essentially one-to-one connections--broadcasts aren't allowed.

The NETBIOS unit supports both datagrams and sessions. It provides these functions:

- determine whether NetBIOS is installed
- reset the network adapter
- add a unique name to the local name table
- add a group name to the local name table
- remove a name from the local name table
- send a datagram
- receive a datagram
- establish a session
- send a session message
- receive a session message
- close a session

NetBIOS calls are issued in one of two ways. In both cases, the register pair ES:BX points to an initialized record called a Control Block (NCB). Then either an int \$5C, or an int \$21 (DOS) function \$2A is issued. The NETBIOS unit initializes the NCB and performs the interrupt for you. (NETBIOS uses the int \$5C rather than the int \$21 method.)

Many NetBIOS calls can operate in a "no-wait" mode. This means that the call returns immediately after the request is submitted and doesn't wait for completion (similar to the way IPX performs under NetWare). In this case, the NCB variable passed as an argument to the NETBIOS routine must remain undisturbed in memory until the call is completed. Care must be taken not to reuse the NCB and, if the NCB is a local variable, not to exit the routine where it was declared until the NetBIOS function is done. When no-wait calls are used, the safest methodology is to declare one NCB per NetBIOS request, declare NCBs as global variables, and check the CmdComplete field of the NCB before reusing it.

The following references on NetBIOS are useful:

C Programmer's Guide to NetBIOS

W. David Schwaderer, Howard W. Sams & Co., 1988.

Inside NetBIOS

J. Scott Haugdahl, Architecture Technology Corp, Minneapolis, MN.

NetBIOS Applications Development Guide

IBM Manual #68X2270.

Datagram Services

The following examples demonstrate low level datagram services. The sender and receiver must agree on the names by which they refer to each other. In some cases, it is appropriate to extract these names from the workstation environment; for this example, arbitrary names are coded into the application.

The sender's side of a datagram message looks as follows:

```

type
    Str80    = String[80];
    PStr80   = ^Str80;

const
    PacketSize = sizeof(Str80); {bytes sent in each packet}
    SenderName = 'SENDER';      {sender name string}
    ReceiverName = 'RECEIVER';  {receiver name string}

var
    Status      : Byte;
    NameNumber  : Byte;
    SendBuf     : PStr80;

{add sender's name to local name table}
Status := NetBiosAddName(SenderName, NameNumber);

{allocate data packet for sender}
SendBuf := PStr80(NetBiosAllocPacket(PacketSize));

{initialize the packet data}
SendBuf^ := 'This is a datagram';

{send datagram}
Status := NetBiosSendDG(NameNumber, ReceiverName, PacketSize,
SendBuf);

{free data packet buffer}
NetBiosFreePacket(PnbPacket(SendBuf));

{delete the sender's name}
Status := NetBiosDeleteName(SenderName);

```

This code sets up and sends a single datagram, which can be as large as 512 bytes. Since adding the sender's name can take a significant amount of time (while the driver broadcasts the name around the network, looking for conflicts), a real application should add and delete it one time only, not once per message. Real code should also check the status results, which equal zero if a function was successful, or else a NetBIOS error code. NetBiosAllocPacket returns Nil if insufficient memory is available for the packet.

This example uses a wait call, NetBiosSendDG, to send the datagram. This means that the routine does not return until the NetBIOS driver acknowledges sending the packet. When sending datagrams, a wait call is usually appropriate since the driver releases the packet quickly and does not wait for a response from the recipient.

The receiver's side looks like this:

```

type
    Str80    = String[80];
    PStr80   = ^Str80;

const
    PacketSize = sizeof(Str80); {bytes sent in each packet}
    SenderName = 'SENDER';      {sender name string}
    ReceiverName = 'RECEIVER';  {receiver name string}

var
    Status      : Byte;

```

```

    GrpNameNumber : Byte;
    ReceiveBuf     : PStr80;
    ReceiveNCB     : PNCB;

{add receiver's name to local name table}
Status := NetBiosAddGroupName(ReceiverName, GrpNameNumber);

{allocate data packet and NCB}
ReceiveBuf := PStr80(NetBiosAllocPacket(PacketSize));
ReceiveNCB := NetBiosAllocNCB;

{post a request to receive a datagram}
NetBiosReceiveDGNowait(ReceiveNCB, Nil, GrpNameNumber, PacketSize,
ReceiveBuf);

{wait for the datagram to arrive}
while (not NetBiosCmdCompleted(ReceiveNCB, Status)) do begin
    {provide a chance for user to break out}
end;

{message is now available in ReceiveBuf}

{free data packet buffer and NCB}
NetBiosFreePacket(PnbPacket(ReceiveBuf));
NetBiosFreeNCB(ReceiveNCB);

{delete the receiver's name}
Status := NetBiosDeleteName(ReceiverName);

```

In this case the receiver adds a group name, which can match names already stored by other stations on the network. Group names are particularly useful in combination with datagrams, since one message can be received by multiple stations who happen to be listening when the datagram is sent.

This code calls a no-wait service to listen for the datagram. This is the usual approach to receiving messages, since there is no guarantee that a message will be received, and it's important to be able to break out if the user becomes impatient or the program times out. To use the no-wait service, the program must allocate an NCB to hold the status of the pending event. After posting the listen, the program loops calling NetBiosCmdCompleted until it returns a non-zero value or the loop is broken. Once NetBiosCmdCompleted returns non-zero, the event is complete and the Status variable indicates the final status of the receive.

Note that the receiver is not told how long the message was. It knows only that the message is no longer than PacketSize bytes. If the length exceeds PacketSize, NetBIOS returns an error code (\$06) and the receiver can call NetBiosReceiveDGNowait again to obtain the overflow.

Session Services

NetBIOS session services are more reliable and allow longer messages than the datagram services. Before messages can be sent and received, a session must be established. The complete sequence of steps to be performed by the sender is shown in the following example:

```

type
    Str80 = String[80];
    PStr80 = ^Str80;
const
    PacketSize = size(Str80);    {bytes sent in each packet}

```

```

    SenderName = 'SENDER';           {sender name string}
    ReceiverName = 'RECEIVER';       {receiver name string}
var
    NameNumber : Byte;
    Status : Byte;
    SendSN : Byte;
    SendNCB : PNCB;
    SendBuf : PStr80;

{add sender's name to name table}
Status := NetBiosAddName(SenderName, NameNumber);

{allocate buffers}
SendNCB := NetBiosAllocNCB;
SendBuf := PStr80(NetBiosAllocPacket(PacketSize));

{open session from sender's side}
NetBiosOpenNoWait(SendNCB, Nil, ReceiverName, SenderName, 5, 5);
while (not NetBiosCmdCompleted(SendNCB, Status)) do begin
    {provide a chance for user to break out}
end;

{store the session number}
SendSN := SendNCB^.LSN;

{initialize the packet data}
SendBuf^ := 'This is a session message';

{send the message}
Status = NetBiosSend(SendSN, PacketSize, SendBuf);

{close the session}
Status := NetBiosHangup(SendSN);

{free the buffers}
NetBiosFreeNCB(SendNCB);
NetBiosFreePacket(SendBuf);

{remove the sender's name}
Status := NetBiosDeleteName(SenderName);

```

The sender adds its name to the name table, then allocates buffers to be used for opening the session and sending packets. It calls `NetBiosOpenNoWait` to open the session. Here it uses a no-wait service because there is no guarantee that the receiver is available and listening. Once the NetBIOS command is completed, the session is open and the session number can be obtained from the LSN field of the NCB. The rest of the code uses the session number to send messages.

The two values of 5 passed to `NetBiosOpenNoWait` specify the timeout period (in half second increments) on subsequent sends and receives. Thus, if a sent message is not acknowledged by the receiver within 2.5 seconds, the `NetBiosSend` command fails with a status code of \$05 (command timed out). If zero is passed for these timeout periods, NetBIOS will retry forever.

Once the session is open, the sender initializes its message buffer and sends the message by calling `NetBiosSend`. It's generally safe to use a wait service here, since the receiver is already waiting and NetBIOS will time out if the session is unexpectedly terminated.

Messages sent using session services can be up to 64KB long. The limit is much lower when using NetBIOS sessions in Windows applications, however. Without increasing certain SYSTEM.INI parameters, session messages are limited to 512 bytes.

The session is closed by calling NetBiosHangup, then the buffers are freed and the name table is restored.

The receiver's side of the session is similar:

```

type
  Str80 = String[80];
  PStr80 = ^Str80;
const
  PacketSize = size(Str80);      {bytes sent in each packet}
  SenderName = 'SENDER';        {sender name string}
  ReceiverName = 'RECEIVER';    {receiver name string}
var
  NameNumber : Byte;
  Status : Byte;
  RecvSN : Byte;
  RecvNCB : PNCB;
  RecvBuf : PStr80;

  {add receiver's name to name table}
  Status := NetBiosAddName(Receiver, NameNumber);

  {allocate buffers}
  RecvNCB := NetBiosAllocNCB;
  RecvBuf := PStr80(NetBiosAllocPacket(PacketSize));

  {open session from receiver's side}
  NetBiosListenNoWait(RecvNCB, Nil, ReceiverName, SenderName, 5, 5);
  while (not NetBiosCmdCompleted(RecvNCB, Status)) do begin
    {provide a chance for user to break out}
  end;

  {store the session number}
  RecvSN := RecvNCB^.LSN;

  {receive a message}
  NetBiosReceiveNoWait(RecvNCB, Nil, RecvSN, PacketSize, RecvBuf);
  while (not NetBiosCmdCompleted(RecvNCB, Status)) do begin
    {provide a chance for user to break out}
  end;

  {message is available in RecvBuf}

  {close the session}
  Status := NetBiosHangup(RecvSN);

  {free the buffers}
  NetBiosFreeNCB(RecvNCB);
  NetBiosFreePacket(RecvBuf);

  {remove the receiver's name}
  Status := NetBiosDeleteName(ReceiverName);

```

Unlike datagram communications, sessions establish a one-to-one connection between sender and receiver. Therefore, it's better to add a *unique* receiver name rather than a group name as shown in

the datagram examples. The code calls NetBiosListenNoWait to open the receiver's side of the session. As before, it loops until the command is complete or a timeout occurs.

Once the session is open, there is no difference between sender and receiver. Either side can send or receive messages. This example code, however, just shows how the receiver gets one message by calling NetBiosReceiveNoWait. It uses a no-wait service because it doesn't know when the sender will start sending. If there is a significant delay (longer than the receive timeout specified in NetBiosListenNoWait), the NetBiosReceiveNoWait call will complete with a status code of \$05 (command timed out). In this case the receiver should call NetBiosReceiveNoWait again to post another listen request.

Only one side of the session needs to call NetBiosHangup. It doesn't hurt for both to do so, but one will succeed and the other will receive error \$0A (session already closed).

Post-Event Routines

Instead of calling NetBiosCmdCompleted repeatedly to see whether a no-wait routine is completed, it's possible to use a "post-event" routine. In this case, NetBIOS automatically calls a specified routine when the event is complete. In the meantime, the application can go on about its business (as long as the NCB variable remains in scope while the event is in progress). All of the no-wait NetBIOS routines take a parameter that specifies the address of a post-event routine. With Nil in this position, no such routine is used. If an address is passed in this position, it must point to an interrupt procedure, which is declared as follows.

```
procedure PostEventHandler(LastError : Byte; N : PNCB);
```

Since this routine is completely analogous to a hardware interrupt handler, it must be written with the same caution. DOS services may not be available when the routine is activated (since DOS is not reentrant); the stack size may be very limited; time-critical foreground processes may be in progress at the time of the call. Maskable interrupts are off when the post-event routine is activated.

LastError is the NetBIOS completion code for the event that just finished. N is the event's NCB.

The post-event routine can issue NetBIOS requests. Probably the most common situation is installing a post-event routine for receiving NetBIOS packets. The post-event routine should save the contents of the packet and then call NetBiosReceiveNoWait to receive another message.

In a Windows application, the post-event routine commonly posts a user-defined message to the application to say that a NetBIOS packet was received.

Declarations

Constants

```
DefaultAdapterNum; : Byte = 0;
```

The network adapter number used for all calls in the NETBIOS unit. Possible values are 0 or 1, with 0 being the primary adapter.

```
NBESuccess                = $00;
NBEInvalidBufferLength    = $01;
NBEInvalidCommand         = $03;
NBETimedOut               = $05;
NBEIncomplete             = $06;
NBELocalNoAckFailed       = $07;
NBEInvalidLSN             = $08;
NBENoResourceAvail        = $09;
```

```

NBESessionClosed      = $0A;
NBECmdCancelled       = $0B;
NBEDuplicateName      = $0D;
NBENametableFull      = $0E;
NBENametableActive    = $0F;
NBELocalSessionTableFull = $11;
NBESessionNoListen    = $12;
NBEIllegalNameNumber  = $13;
NBECannotFindName     = $14;
NBENoAnswer           = $14;
NBEInvalidName        = $15;
NBENameInUseOnRemote  = $16;
NBENamedeleted        = $17;
NBESessionAbnormal    = $18;
NBENameConflict       = $19;
NBEIncompatibleDevice = $1A;
NBEInterfaceBusy      = $21;
NBETooManyCommands    = $22;
NBEInvalidLanA        = $23;
NBECmdCompletedWhileCancel = $24;
NBEReservedName       = $25;
NBENotValidCancel     = $26;
NBESystemError        = $40;
NBEHotCarrierFromRemote = $41;
NBEHotCarrier         = $42;
NBENoCarrier          = $43;
NBECmdPending         = $FF;
NBEDPMIError          = $FE;
NBEUnexpectedAdaptClose = $FD;

```

NetBIOS error codes returned in the RetCode field of an NCB. See the functions in this chapter for more information about how they apply to each NetBIOS command. Many NetBIOS commands can generate errors \$21 and \$22. When this occurs, the appropriate response is to wait a while and try again. Commands can also generate errors \$40-\$FD. These errors indicate a hardware failure or an internal NetBIOS error. The appropriate response is to halt the application and repair the hardware. Error code \$FE is specially defined by the NETBIOS unit for use in protected mode applications. It is returned in three situations: 1) if insufficient free DOS memory exists to allow allocating a temporary NCB in a wait service; 2) if the NCB address or packet buffer address passed to a function was not correctly allocated in DOS memory; or 3) if the DPMI (DOS Protected Mode Interface) service used to call the NetBIOS driver fails.

```
NBNameMax; = 16;
```

The maximum length of a NetBIOS name.

```
NetBiosReenterError; : Boolean = False;
```

When using post-event routines in protected mode, a DPMI real-mode callback is used to translate the real mode NetBIOS post-event call into protected mode. If the post-event handler you write is servicing receipt events, you should call NetBiosReceiveNoWait to post the NCB back to NetBIOS to receive the next packet. If there is a high volume of packet traffic, the next packet could arrive before the post-event handler is finished. If this is the case, the real mode callback stub that the NETBIOS unit installs sets NetBiosReenterError to True and exits. Rather than try and cure the problem, it is best to design your application so that NetBiosReenterError never becomes True. This can usually be done by posting a pool of NCBs to receive packets. A pool of 10 NCBs is usually sufficient, even for the most demanding of packet traffic.

Types

```
CallNameType; = array[1..NBNameMax] of Char;
```

Internal format used by NetBIOS to store names.

```
NBNameStr; = String[NBNameMax];
```

Type for a NetBIOS name.

```
NetBiosName; = record
    Name      : CallNameType;
    Number    : Byte;
    Status     : Byte;
end;
```

Information about a NetBIOS name returned by the NetBiosInfo function.

```
NetBiosPostRoutine; = procedure (LastError : Byte; N : PNCB);
```

Procedure prototype for a NetBIOS post-event routine. LastError is the completion code of the finished event. N is the event's NCB.

```
PnbPacket; = pointer;
```

Pointer to a data packet. All data packets *must* be allocated by calling NetBiosAllocPacket.

```
PNCB = ^TNCB
TNCB; = record
    Command      : Byte;
    RetCode       : Byte;
    LSN           : Byte;
    NameNum       : Byte;
    Buffer         : Pointer;
    BufLen        : Word;
    RemName       : CallNameType;
    LocName       : CallNameType;
    RTO           : Byte;
    STO           : Byte;
    PostRoutine   : Pointer;
    LanANum       : Byte;
    CmdComplete   : Byte;
    Reserved      : array[1..14] of Byte;
end;
```

The control block. This 64 byte record structure is the heart of all NetBIOS requests. Although the routines in the NETBIOS unit generally manage these fields for you, a brief description of each member may be of interest.

Command specifies which NetBIOS command to execute. RetCode is the final result of the command; many of the possible values are shown in the list of error codes earlier in this section. RetCode is valid only after CmdComplete does not equal \$FF. LSN is the local session number (if any) associated with the command. NameNum is a NetBIOS name table number associated with the command. Buffer is a pointer to data associated with the command. BufLen is the length in bytes of the data buffer. RemName is usually the remote name associated with the command, while LocName is the local name. RTO is the receive timeout value, in half-second units. STO is the send timeout value. PostRoutine is a pointer to a routine to call upon completion of a no-wait command (the NETBIOS unit installs a stub that sets up the program environment and then calls your post-event handler, if you are using one). LanANum is the number of the network adapter to use; in the NETBIOS unit it is always assigned the value of the global variable DefaultAdapterNum. CmdComplete is a flag that equals \$FF while the command is pending, and changes to a different value thereafter. Under Windows, the NETBIOS unit expands the standard NCB to hold extra fields that are used by the post-event handler stub.

```
PNetBiosStatus = ^TNetBiosStatus;
TNetBiosStatus; = record
    PermanentNodeName : Array [1..6] of char; {hardware node name}
    ExtJumpers         : Byte;
    SelfTest           : Byte;
    ProtocolMajor      : Byte;                {major and minor
version numbers}
    ProtocolMinor      : Byte;
    ReportingPeriod    : Word;                {dynamic status of
```

```

driver}
  CRCCount          : Word;
  AlignmentErrors   : Word;
  Collisions        : Word;
  TransmitAborts    : Word;
  Transmits         : LongInt;
  Receives          : LongInt;
  Retransmits       : Word;
  ResourceDepletion : Word;
  ReservedArea1     : Array [1..8] of Byte;
  FreeCommandBlocks : Word;           {available pending
commands}
  CurrentMaxNCBs    : Word;           {max NCBs Driver
configured for}
  HardwareMaxNCBs   : Word;           {max NCBs driver can
support}
  ReservedArea2     : Array [1..4] of Byte;
  Sessions          : Word;           {current number of
active sessions}
  CurrentMaxSessions : Word;           {max sessions driver
configured for}
  HardwareMaxSessions : Word;          {max sessions driver can
support}
  MaxPacketSize     : Word;           {largest packed in
bytes}
  NameCount         : Word;           {number of names in
adapter table}
  NetBiosNames      : Array [1..16] of NetBiosName;
end;

```

Information returned by the NetBiosInfo function. Much of the information is applicable only to IBM's PC-LAN implementation of NetBIOS.

```

  PPostHandler = ^TPostHandler;
  TPostHandler; = record
    ...
  end;

```

A post-event handler and a pointer to it. The fields of this type are used internally by the NETBIOS unit's post-event stub routine. The layout of this type varies for different compiler targets.

NetBiosAddGroupName / NetBiosAddName

Syntax

```
function NetBiosAddGroupName; (NameToAdd : NbNameStr;  
                               var NameNumber : Byte) : Byte;
```

Purpose

Add a group name to the local NetBIOS name table.

Description

Other nodes can be using the same name as a group name, but not as a unique name. NetBIOS repeatedly broadcasts this name across the network. If no conflicting response is received after an adequate number of retries, NetBiosAddGroupName succeeds. This function does not return until the command is complete.

NameToAdd is any ASCII string up to 16 characters long. IBM imposes several restrictions on the name, however. It cannot begin with a null (#0), asterisk (*), or the three letters 'IBM'. The sixteenth character cannot be ASCII #31. NetBiosAddGroupName pads the string with nulls to reach 16 characters. Case is significant when comparing NetBIOS names.

NameNumber returns the local name table slot assigned to the name. This number is used to call the datagram routines.

A status code is returned in function result:

```
$00 Name added successfully.  
$0D Duplicate name in local table.  
$0E Name table is full.  
$15 Invalid name.  
$16 Name in use on another station.  
$19 Name conflict detected. (duplicate names elsewhere)
```

See Also

NetBiosAddName

NetBiosDeleteName

Syntax

```
function NetBiosAddName; (NameToAdd : NbNameStr; var NameNumber :  
                          Byte) : Byte;
```

Purpose

Add a unique name to the local NetBIOS name table.

Description

NetBIOS repeatedly broadcasts this name across the network. If no matching response is received after an adequate number of retries, NetBiosAddName succeeds. This function does not return until the command is complete.

NameToAdd is any ASCII string up to 16 characters long. IBM imposes several restrictions on the name, however. It cannot begin with a null (#0), asterisk (*), or the three letters 'IBM'. The sixteenth character cannot be ASCII #31. NetBiosAddName pads the string with nulls to reach 16 characters. Case is significant when comparing NetBIOS names.

NameNumber returns the local name table slot assigned to the name. This number is used to call the datagram routines.

A status code is returned in function result:

```
$00 Name added successfully.  
$0D Duplicate name in local table.  
$0E Name table is full.  
$15 Invalid name.  
$16 Name in use on another station.  
$19 Name conflict detected. (duplicate names elsewhere on network)
```

NetBiosAddGroupName / NetBiosAddName

See Also

~~NetBiosAddGroupName~~
NetBiosOpen

~~NetBiosDeleteName~~
NetBiosOpenNoWait

NetBiosAllocNCB / NetBiosAllocPacket

Syntax

```
function NetBiosAllocNCB; : PNCB;
```

Purpose

Allocate and return a pointer to an .

Description

You must allocate a NetBIOS control block (NCB) prior to calling any no-wait function from the NETBIOS module. The NCB must remain accessible to the NetBIOS driver until the operation is complete.

For protected mode applications, the NCB *must* be allocated by calling NetBiosAllocNCB. NetBiosAllocNCB calls the GlobalDosAlloc routine to allocate memory that is accessible from both the real and protected modes of the CPU. This is necessary because the NetBIOS driver is written for DOS real mode operation. For these applications, NetBiosAllocNCB returns a protected mode pointer (in selector:offset format) that you can dereference in your application without any conversion. The NETBIOS module converts it to a real mode pointer (segment:offset format) prior to passing control to the NetBIOS driver.

For DOS and Windows applications, NetBiosAllocNCB essentially calls GetMem to allocate the NCB on the heap. Alternatively, in real mode DOS and Windows applications only, you can declare an NCB as a global or local variable and pass its address to any no-wait routine by using the '@' operator. This approach is not recommended, however, since it limits the portability of the code.

NetBiosAllocNCB returns Nil if there is insufficient free memory to satisfy the request.

See Also

NetBiosAllocPacket

NetBiosFreeNCB

Syntax

```
function NetBiosAllocPacket;(Size : Word) : PnbPacket;
```

Purpose

Allocate and return a pointer to a .

Description

Protected mode applications must call this function to allocate all packet buffers that are passed to a function in the NETBIOS unit. The reasoning is equivalent to that described for NetBiosAllocNCB (which actually just calls NetBiosAllocPacket). The pointer returned by NetBiosAllocPacket is a protected mode pointer that can be used by protected mode applications without any conversion.

Real mode DOS and Windows applications can also call NetBiosAllocPacket. They can also declare packet buffers as global or local variables. The preferred approach is to call NetBiosAllocPacket.

See Also

NetBiosAllocNCB

NetBiosFreePacket

NetBiosAllocPost / NetBiosCancelRequest

Syntax

```
function NetBiosAllocPost; (Handler : NetBiosPostRoutine) :  
  PPostHandler;
```

Purpose

Allocate and return a pointer to a .

Description

Applications that make use of post-event handlers must call this routine to create a special post-event control block. This function takes the address of a post-event procedure of type NetBiosPostRoutine and creates a special stub routine to call it. This stub routine is the actual post-event handler that NetBIOS calls when an event completes. The routine sets up the program environment and then call the Handler procedure.

The variable that is returned by NetBiosAllocPost is different for each compiler target, but in all three environments it contains enough data to ensure that the stub can perform its task. In protected mode, a DPML real mode callback is allocated for each call to NetBiosAllocPost. Remember that the DPML server is only guaranteed to provide 16 callbacks per process, and other parts of your application will require some.

If an error occurs (such as an out of memory condition or a DPML error), NetBiosAllocPost returns Nil.

See Also

NetBiosFreePost

Syntax

```
function NetBiosCancelRequest; (N : PNCB) : Byte;
```

Purpose

Cancel a pending request.

Description

N is the NCB of the command that is to be cancelled. This routine can be used to cancel the operation of any NETBIOS functions that has NoWait in its name. All other functions do not return until the operation is complete, so NetBiosCancelRequest is irrelevant to them.

A status code is returned in function result:

- \$00 Command cancelled.
- \$24 Command completed while cancel occurring.
- \$26 Command not valid to cancel.

See Also

NetBiosCmdCompleted

NetBiosFreePost

NetBiosClearNCB / NetBiosCmdCompleted

Syntax

```
procedure NetBiosClearNCB; (N : PNCB);
```

Purpose

Initialize an NCB.

Description

This function fills the NCB with zeros and sets its LanANum field to DefaultAdapterNum. This is always required before using an NCB. The higher level functions in the NETBIOS module call this function to initialize all NCBs they use. It is interfaced in case you need to make a direct call to the NetBIOS.

See Also

NetBiosRequest

Syntax

```
function NetBiosCmdCompleted; (N : PNCB; var FinalRetCode : Byte) :  
Boolean;
```

Purpose

Determine whether a no-wait event is complete.

Description

Call this routine repeatedly after using any NETBIOS unit no-wait function without specifying a post-event handler. It returns True and the final status code in FinalRetCode when the event is complete. In the meantime the application can continue with other tasks or cancel the event if desired.

N is the same NCB that was passed to the original no-wait function. This variable must remain in scope and must not be reused until the event completes or is cancelled.

Calling NetBiosCmdCompleted is equivalent to comparing the CmdComplete field of the NCB to NBECommandPending.

See Also

NetBiosListenNoWait
NetBiosReceiveBDGNoWait
NetBiosReceiveNoWait
NetBiosSendDGNoWait

NetBiosOpenNoWait
NetBiosReceiveDGNoWait
NetBiosSendBDGNoWait
NetBiosSendNoWait

NetBiosDeleteName / NetBiosFreeNCB

Syntax

```
function NetBiosDeleteName; (NameToDelete : NDBNameStr) : Byte;
```

Purpose

Delete a unique or group name from the local NetBIOS name table.

Description

If a name is deleted while a session is actively using it, the actual deletion is delayed until the session is terminated, typically by a NetBiosHangup command. NetBiosDeleteName returns error code \$0F in this case. Even though the name table space is not immediately reclaimed, the name is "deregistered," so that datagrams and new sessions cannot use it.

All names added to the name table by an application should be deleted before it exits. NetBIOS does not do this automatically. See the demonstration programs NBSend for an example of using a Turbo Pascal exit procedure to delete the names.

A status code is returned in the function result:

```
$00 Name deleted successfully.  
$0F Name has active sessions. Deletion is delayed.  
$15 Invalid name.
```

See Also

NetBiosAddGroupName

NetBiosAddName

Syntax

```
procedure NetBiosFreeNCB; (N : PNCB);
```

Purpose

Free an .

Description

All NCB's allocated using NetBiosAllocNCB should be freed by calling NetBiosFreeNCB. Otherwise heap space that is no longer in use remains allocated.

See Also

NetBiosAllocNCB

NetBiosFreePacket / NetBiosFreePost / NetBiosHangUp

Syntax

```
procedure NetBiosFreePacket; (P : PnbPacket);
```

Purpose

Free a .

Description

All packet buffers allocated using NetBiosAllocPacket should be freed by calling NetBiosFreePacket. Otherwise heap space that is no longer in use remains allocated.

See Also

NetBiosAllocPacket

Syntax

```
procedure NetBiosFreePost; (P : PPostHandler);
```

Purpose

Free a .

Description

All post-event handlers allocated by NetBiosAllocPost should be freed by calling NetBiosFreePost. Otherwise heap space that is no longer in use remains allocated. In protected mode, a DPMI callback also remains allocated if you don't call NetBiosFreePost.

See Also

NetBiosAllocPost

Syntax

```
function NetBiosHangUp; (SessionNumber : Byte) : Byte;
```

Purpose

Terminate an existing session.

Description

The function does not return until the command is complete. Either side of a session can call this routine to terminate the session.

A status code is returned in the function result:

```
$05 Command timed out.  
$08 Invalid local session number.  
$0A Session already closed.  
$0B Command was cancelled.  
$18 Session ended abnormally.
```

NetBIOS terminates any pending NetBIOS commands that use the specified session when this routine is called, although this can generate a delay before NetBiosHangUp returns.

See Also

NetBiosDeleteName

NetBiosOpen

NetBiosInfo / NetBiosInstalled

Syntax

```
function NetBiosInfo( NS : PNetBiosStatus; Name : NNameStr;  
                    MaxNames : Byte) : Byte;
```

Purpose

Return information about a driver or adapter.

Description

Name specifies the NetBIOS driver or adapter that NetBiosInfo queries for information. If Name equals '*', the default local adapter is used. If Name is a node, unique, or group name of a different adapter, information is returned from that adapter.

NS points to a buffer that is at least 60 plus 18*MaxNames bytes in size (the default size for a TNetBiosStatus variable assumes MaxNames is 16). The tail of the NetBiosStatus struct is a variable length array that contains information about each NetBIOS name stored in the specified adapter. In a protected mode applications, NS must be allocated by calling NetBiosAllocPacket.

The typical value to pass for MaxNames is 16, although some NetBIOS drivers allow a larger number of names. If MaxNames is smaller than the actual number of names, NetBiosInfo returns information about the names that will fit.

The interpretation of many of the fields in the TNetBiosStatus structure depends on the particular NetBIOS implementation. Comments in the type definition for NetBiosStatus earlier in this section describe the fields that are interpreted the same in most implementations.

The name table at the end of NetBiosStatus is always interpreted the same. In each NetBiosName struct, the Name member is the actual name, Number is the name number, and Status is a set of bitmapped flags that specify the name's state. The high bit of Status is set if the name is a group name, clear if it is a unique name. The low three bits of Status are interpreted as follows:

000	the name is being registered
100	the name is registered
101	the name is deregistered
110	the name is a detected duplicate
111	the name is a detected duplicate that is being deregistered

The other bits of Status are not currently used by NetBIOS.

A status code is returned in the function result:

\$00	Status returned.
\$05	Command timed out.
\$06	Buffer too small; partial information returned.

Syntax

```
function NetBiosInstalled; : Boolean;
```

Purpose

Return True if NetBIOS or a compatible emulator is installed.

Description

A check is made to see whether the int \$5C vector contains the Nil address or points into the PC ROM BIOS. If either is the case, it can be concluded that NetBIOS is not loaded. Otherwise, an invalid NetBIOS request (\$7F) is issued. This command returns NBEInvalidCommand in NCB.RetCode if NetBIOS is installed. There is a slight chance that this technique will conflict with another int \$5C handler, but since this interrupt is reserved for NetBIOS by IBM, it is quite unlikely.

Example

```
if not NetBiosInstalled then begin  
  Writeln('This program requires a NetBIOS driver');
```


NetBiosInfo / NetBiosInstalled

```
    Halt;  
end;
```

Halts an application if NetBIOS is not installed.

NetBiosListen / NetBiosListenNoWait

Syntax

```
function NetBiosListen( RemoteName, LocalName : NBNameStr;  
                      SendTimeout, RecTimeout : Byte;  
                      var SessionNumber : Byte) : Byte;
```

Purpose

Listen for a call to initiate a .

Description

The local station must call NetBiosListen before the remote station calls NetBiosOpen. To make a successful connection, each command is generally called within a loop (which should also provide an opportunity for the user to abort). Once the session is created, either station can send or receive messages. See the OpenSession function in the demo program NBSEND.PAS for a reliable routine to open a session.

LocalName specifies an existing name in the local adapter's name table. RemoteName specifies a name in the table of the caller. Both ends must agree on the names in order for a session to be created. Generally, the names should be unique, since a session cannot connect more than two stations. It is also possible for both ends of the session to be on a single workstation.

SendTimeout and RecTimeout specify how long (in half-second increments) the NetBIOS will retry when sending and receiving messages. A value of zero means to retry forever. Some NetBIOS implementations ignore these parameters, so you must use NetBiosSendNoWait or NetBiosReceiveNoWait.

Upon successful completion of this routine, SessionNumber contains a local session number. This number is used in later calls to send or receive messages, and to disconnect the session.

A status code is returned in the function result:

```
$00 Session established.  
$05 Command timed out.  
$09 Remote session table full.  
$11 Local session table full.  
$15 Invalid name.  
$18 Session ended abnormally.  
$19 Name conflict detected. (duplicate names elsewhere on network)
```

Example

See the introduction to this section.

See Also

NetBiosListenNoWait

NetBiosOpen

Syntax

```
procedure NetBiosListenNoWait( N : PNCB; PostEvent : PPostHandler;  
                              RemoteName, LocalName : NBNameStr;  
                              SendTimeout, RecTimeout : Byte);
```

Purpose

Listen for a call to initiate a session; don't wait for connection.

Description

This routine works like NetBiosListen, except that it returns immediately instead of waiting for the connection to be established or for timeout to occur.

The application should either specify a Pascal post-event routine and convert it into a PostEvent handler with NetBiosAllocPost, or should loop calling NetBiosCmdCompleted to see whether the session has been established. The NCB pointed to by N is initialized by this call; its associated variable must remain valid until the event completes or is cancelled. When the command is complete, the session number is found in the LSN field of the NCB.

NetBiosListen / NetBiosListenNoWait

See Also

[NetBiosCmdCompleted](#)

[NetBiosListen](#)

NetBiosOpen

Syntax

```
function NetBiosOpen(RemoteName, LocalName : NDNameStr;  
    SendTimeOut, RecTimeOut : Byte;  
    var SessionNumber : Byte) : Byte;
```

Purpose

Call another station to initiate a session.

Description

The remote station must call NetBiosListen before the local station calls NetBiosOpen. To make a successful connection, each command is generally called within a loop (which should also provide an opportunity for the user to abort). Once the session is created, either station can send or receive messages. See the OpenSession function in the demo program NBSend.PAS for a reliable routine to open a session.

LocalName specifies an existing name in the local adapter's name table. RemoteName specifies a name in the table of the listener. Both ends must agree on the names in order for a session to be created. Generally, the names should be unique, since a session cannot connect more than two stations. It is also possible for both ends of the session to be on a single workstation.

SendTimeOut and ReceiveTimeOut specify how long (in half-second increments) the NetBIOS will retry when sending and receiving messages. A value of zero means to retry forever. Some NetBIOS implementations ignore these parameters, so you must use NetBiosSendNoWait or NetBiosReceiveNoWait.

Upon successful completion of this routine, SessionNumber contains a local session number. This number is used in later calls to send or receive messages, and to disconnect the session.

A status code is returned in the function result:

```
$00 Session established.  
$05 Command timed out.  
$09 Remote session table full.  
$11 Local session table full.  
$12 No listen is outstanding.  
$14 Cannot find name called, or no answer.  
$15 Invalid name.  
$18 Session ended abnormally.  
$19 Name conflict detected (duplicate names elsewhere on network).
```

The strategy for choosing NetBIOS names is similar to choosing unique socket numbers under NetWare. The easiest, and usually the best, strategy is to code the name directly into the application, but to make that name installable by the network administrator. In the unlikely event of a conflict with another application running on the network, the administrator can choose a different name. Alternative strategies involve the use of broadcast datagrams or shared files to establish an acceptable set of NetBIOS names.

See Also

NetBiosListen

NetBiosOpenNoWait

NetBiosOpenNoWait

Syntax

```
procedure NetBiosOpenNoWait; (N : PNCB; PostEvent : PPostHandler;  
                             RemoteName, LocalName : NbNameStr;  
                             SendTimeout, RecTimeout : Byte);
```

Purpose

Call another station to initiate a session; don't wait for connection.

Description

This routine works like NetBiosOpen, except that it returns immediately after posting the event.

The application should either specify a Pascal post-event routine and convert it into a PostEvent handler with NetBiosAllocPost, or should loop calling NetBiosCmdCompleted to see whether the session has been established. The NCB pointed to by N is initialized by this call; its associated variable must remain valid until the event completes or is cancelled. When the command is complete, the session number is found in the LSN field of the NCB.

Example

See the introduction to this section.

See Also

NetBiosCancelRequest
NetBiosListenNoWait

NetBiosCmdCompleted
NetBiosOpen

NetBiosReceive / NetBiosReceiveNoWait

Syntax

```
function NetBiosReceive; (SessionNumber : Byte; PacketSize : Word;  
    Packet : PnbPacket) : Byte;
```

Purpose

Receive data during a session.

Description

SessionNumber must have been initialized by a prior call to NetBiosCall or NetBiosListen. NetBiosReceive waits for the command to complete, either by receiving the data or by timing out with an error.

PacketSize specifies the maximum size of the data packet (0..65521) and Packet is the buffer to receive the data. In protected mode applications, Packet *must* be allocated by calling NetBiosAllocPacket, and it is advisable to do so in real mode DOS and Windows applications as well. It is essential that the Packet variable be at least as large as the PacketSize parameter. Otherwise, memory can be overwritten, probably leading to a program crash. If the incoming packet is larger than PacketSize, NetBIOS generates an error but does not overwrite memory.

A status code is returned in the function result:

```
$00 Message received.  
$05 Command timed out.  
$06 Incomplete received message (buffer too small).  
$08 Invalid local session number.  
$0A Session was closed.  
$0B Command was cancelled.  
$18 Session ended abnormally.
```

If an error \$06 is returned, the receiver can request the remainder of the packet by immediately issuing another NetBiosReceive call.

Example

See the introduction to this section.

See Also

NetBiosReceiveNoWait

NetBiosSend

Syntax

```
procedure NetBiosReceiveNoWait; (N : PNCB; PostEvent : PPostHandler;  
    SessionNumber : Byte; PacketSize :  
    Word;  
    Packet : PnbPacket);
```

Purpose

Receive data during a session; don't wait for completion.

Description

This routine works like NetBiosReceive, except that it does not wait to receive the message before returning.

The application should either specify a Pascal post-event routine and convert it into a PostEvent handler with NetBiosAllocPost, or should loop calling NetBiosCmdCompleted to see whether the message has been received. The NCB pointed to by N is initialized by this call; its associated variable must remain valid until the event completes or is cancelled. The same is true for the buffer pointed to by Packet.

Example

See the introduction to this section.

See Also

[NetBiosReceive](#)

NetBiosReceive / NetBiosReceiveNoWait

[NetBiosSendNoWait](#)

NetBiosReceiveBDG / NetBiosReceiveBDGNoWait

Syntax

```
function NetBiosReceiveBDG (ReceiverNameNum : Byte; DatagramSize : Word;  
                                Datagram : PnbPacket) : Byte;  
procedure NetBiosReceiveBDGNoWait (N : PNCB; PostEvent : PPostHandler;  
                                ReceiverNameNum : Byte;  
                                DatagramSize : Word;  
                                Datagram : PnbPacket);
```

Purpose

Receive a broadcast datagram.

Description

These commands work just like NetBiosReceiveDG and NetBiosReceiveDGNoWait, but they receive only those messages sent by NetBiosSendBDG and NetBiosSendBDGNoWait. See NetBiosReceiveDG and NetBiosReceiveDGNoWait for more information.

See Also

NetBiosSendBDG

NetBiosReceiveDG / NetBiosReceiveDGNoWait

Syntax

```
function NetBiosReceiveDG; (ReceiverNameNum : Byte; DatagramSize :  
Word;  
Datagram : PnbPacket) : Byte;  
procedure NetBiosReceiveDGNoWait; (N : PNCB; PostEvent : PPostHandler;  
ReceiverNameNum : Byte; DatagramSize  
: Word;  
Datagram : PnbPacket);
```

Purpose

Receive a normal .

Description

These functions receive datagrams that were sent to the NetBIOS name with number ReceiverNameNum in the local name table. Any other station could have transmitted the datagram. The transmission could have been directed to a unique name or to a group name shared by many stations. Although this command does not receive broadcast datagrams, it can be made to intercept any normal datagram if the value \$FF is specified for ReceiverNameNum.

The call to NetBiosReceiveDG or NetBiosReceiveDGNoWait must be pending when the sender transmits the message. If not, the message is lost.

If you use NetBiosReceiveDG, the call does not return until a message is received. Use this option with care, because there is no time out value associated with datagram reception. In most cases, using NetBiosReceiveDGNoWait is the better option. Then the routine returns immediately. The calling application can go ahead with other work while waiting for a message to appear, as long as the NetBIOS control block N and the data buffer Datagram remain undisturbed in memory. When a message finally arrives, the NetBIOS writes status and data to them.

In protected mode applications, the Datagram buffer must be allocated by calling NetBiosAllocPacket. The maximum size of a datagram is implementation specific, but is typically 512 bytes. Be sure that the size of the buffer is at least as large as DatagramSize to prevent NetBIOS from overwriting memory.

For real mode DOS and Windows applications, it is recommended to allocate the Datagram packet with NetBiosAllocPacket as this makes your code more portable between compiler targets.

A status code is returned in the function result of NetBiosReceiveDG (or the FinalRetCode parameter of NetBiosCmdCompleted when NetBiosReceiveDGNoWait is called):

```
$00 Datagram received.  
$06 Incomplete received message.  
$0B Command was cancelled.  
$13 Illegal name number.  
$17 Name was deleted.  
$19 Name conflict detected.
```

If error \$06 is received, the message was larger than DatagramSize. The remainder of the message is lost.

See Also

NetBiosReceiveBDG

NetBiosSendDG

NetBiosRequest / NetBiosResetAdapter

Syntax

```
procedure NetBiosRequest; (N : PNCB);
```

Purpose

Issue a direct NetBIOS call.

Description

This routine sets ES:BX to point to the NCB and then calls interrupt \$5C. In protected mode applications, the NCB must have been allocated previously by calling NetBiosAllocNCB. The NCB must have been initialized previously by calling NetBiosClearNCB and then setting the appropriate fields. Applications won't typically need to call this low-level routine. It is interfaced in case you need to make a direct call to the NetBIOS that is not supported by the NETBIOS unit.

See Also

NetBiosClearNCB

Syntax

```
function NetBiosResetAdapter; (SessionCount : Byte; CommandCount :  
Byte) : Byte;
```

Purpose

Reset a adapter card.

Description

The global variable DefaultAdapterNum, which defaults to 0, specifies the adapter card to reset. This command was originally designed to reset the hardware network adapter for IBM's PC LAN; NetBIOS emulators may treat the command differently. One of its functions is to clear the local name and session tables; all NetBIOS versions should perform this action.

SessionCount is the number of simultaneous sessions that the current workstation will allow. CommandCount is the maximum number of pending commands. Specify 0 for each parameter to keep the current settings. Unless the NETBIOS unit's no-wait services are being used, the command count should not matter.

Note that other programs might be using the NetBIOS adapter or driver, and if you call NetBiosResetAdapter, you will destroy their list of names in the name table. This is especially a concern with Windows for Workgroups because it uses NetBIOS to communicate with other workstations on the network. Calling NetBiosResetAdapter in this situation is *not* recommended.

A status code is returned in the function result:

```
0x00 Successful reset.  
0x23 Invalid adapter number.
```

NetBiosSend / NetBiosSendNoWait

Syntax

```
function NetBiosSend; (SessionNumber : Byte; PacketSize : Word; _____  
    Packet : PnbPacket) : Byte;
```

Purpose

Send data during a session.

Description

SessionNumber must have been initialized by a prior call to NetBiosCall or NetBiosListen. NetBiosSend waits for the command to complete, either with an acknowledgement from the receiver or by timing out with an error.

PacketSize is the size of the data packet. Although the NetBios standard allows sending up to 65535 bytes in a single packet, Windows applications are typically limited to 512 byte packets.

The address of the data to be sent is passed in Packet. In protected mode applications, Packet must be allocated by calling NetBiosAllocPacket. This is also recommended for real mode and Windows applications.

A status code is returned in the function result:

```
$00 Message sent.  
$05 Command timed out.  
$08 Invalid local session number.  
$0A Session was closed.  
$0B Command was cancelled.  
$18 Session ended abnormally.
```

See Also

NetBiosReceive

NetBiosSendNoWait

Syntax

```
procedure NetBiosSendNoWait; (N : PNCB; PostEvent : PPostHandler;  
    SessionNumber : Byte; PacketSize : Word;  
    Packet : PnbPacket);
```

Purpose

Send data during a session; don't wait for completion.

Description

This routine works like NetBiosSend, except that it does not wait for the message to be sent before returning. Use a post-event handler or poll NetBiosCmdCompleted to determine when the message has been successfully sent and the NCB can be reused.

See Also

NetBiosReceiveNoWait

NetBiosSend

NetBiosSendBDG / NetBiosSendBDGNoWait

Syntax

```
function NetBiosSendBDG; (SenderNameNum : Byte; DatagramSize : Word;  
                        Datagram : PnbPacket) : Byte;  
  
procedure NetBiosSendBDGNoWait; (N : PNCB; PostEvent : PPostHandler;  
                                SenderNameNum : Byte; DatagramSize :  
                                Word;  
                                Datagram : PnbPacket);
```

Purpose

Send a broadcast datagram.

Description

These commands work just like NetBiosSendDG and NetBiosSendDGNoWait, except that the message is sent to all nodes. Only nodes listening with NetBiosReceiveBDG or NetBiosReceiveBDGNoWait will receive the message. See NetBiosSendDG and NetBiosSendDGNoWait for more information.

See Also

NetBiosReceiveBDG

NetBiosSendDG

NetBiosSendDG / NetBiosSendDGNoWait

Syntax

```
function NetBiosSendDG; (SenderNameNum : Byte; ReceiverName : _____
NBNameStr;
                                DatagramSize : Word; Datagram : PnbPacket) :
Byte;
procedure NetBiosSendDGNoWait; (N : PNCB; PostEvent : PPostHandler;
                                SenderNameNum : Byte; ReceiverName :
NBNameStr;
                                DatagramSize : Word; Datagram :
PnbPacket);
```

Purpose

Send a normal .

Description

If ReceiverName is a unique name, the message will go to at most one station. If ReceiverName is a group name, the message can go to many stations at once. ReceiverName can refer to a name within the current workstation; if so, and the station is listening, it receives its own message.

SenderNameNum is the slot number associated with a name previously added to the sender's local name table. The number is used by NetBIOS to manage the transmission. Slot number 1 is always safe to use, since this slot is associated with the adapter's permanent name.

The message is lost if the target station does not have a NetBiosReceiveDG or NetBiosReceiveDGNoWait call pending when the datagram is sent.

If you use NetBiosSendDG, the NetBIOS call does not return until the message is transmitted. This function returns a status code that can be used to determine whether the transmission was successful. (Whether the message was received is another matter; datagram senders are not notified whether a given message was received, unless the receiver explicitly sends a reply. For messages requiring confirmation, use the NetBIOS session services instead.)

If you use NetBiosSendDGNoWait, the call returns immediately. The calling application can go ahead with other work while waiting for the transmission to occur. In the meantime, the NCB variable must remain undisturbed in memory because the NetBIOS will eventually write status information to it.

Datagram contains the address of a message buffer to send. In a protected mode application, it must be allocated by calling NetBiosAllocPacket (this is also recommended for real mode DOS and Windows applications). The maximum size of a datagram is implementation specific, but is typically 512 bytes. DatagramSize specifies the actual number of bytes to send.

A status code is returned in the function result of NetBiosSendDG (or the FinalRetCode parameter of NetBiosCmdCompleted when NetBiosSendDGNoWait is called):

```
$00 Datagram sent.
$01 Invalid buffer length (DatagramSize too large).
$13 Illegal name number (sender name number doesn't exist).
$19 Name conflict detected (duplicate names elsewhere on network).
```

See Also

NetBiosReceiveDG

NetBiosSendBDG

The SHARE unit implements assorted network-related functions. Many will work whenever MS-DOS version 3.1 or later is in use. Others require loading the SHARE.EXE utility provided with the same versions of MS-DOS. Some require that the workstation be attached to an MS-NET compatible network such as 3Com's 3+. Novell's NetWare emulates most of these interfaces, so almost all the functions work with NetWare. Prerequisites are noted in the description for each routine. Many functions state that SHARE.EXE must be loaded, but in truth many network shells incorporate this function within another file.

Here is a list of the capabilities provided by the SHARE unit:

- determine whether SHARE is loaded
- lock and unlock any portion of a file
- set the retry count for locking errors
- flush DOS buffers to disk
- get DOS extended error information
- determine whether a drive or file is on a remote machine
- determine or modify a printer setup string
- get the current workstation name
- determine whether PC LAN is loaded
- get PC LAN version information
- redirect a network drive or printer to a local name
- cancel redirection

All of these concepts, with the possible exception of "redirection," will be clear to programmers. Redirection is used here in a different sense than typical for DOS, where it usually refers to channeling the standard input or output device to a different source or destination. In the context of MS-Net, redirection signifies the process of making a network resource (a printer or disk drive) available to a workstation. Redirection stores a name meaningful to the workstation in a table used by the MS-NET "redirector services," so that MS-Net can later translate a reference to the local name and redirect the resource request across the network. See `GetRedirectionEntry` and `RedirectDevice` for more information.

The following references provide further information about the calls in the SHARE unit:

Advanced MS-DOS Programming, 2nd Edition

Ray Duncan, Microsoft Press, 1988.

DOS Programmer's Reference (4th Edition)

Terry R. Dettman, Que Corporation, 1993.

PC Network Technical Reference

IBM manual #632205.

Most routines return standard DOS error codes. The SHARE unit defines four additional error codes, described later in this section. For a list of DOS error codes, see the `GetExtendedError` function.

Declarations

Constants

```
shErrNoDosMemory      = $FFFC;
shErrBadDosVersion    = $FFFD;
shErrFileNotOpen      = $FFFE;
shErrShareNotLoaded   = $FFFF;
```

Non-DOS error codes returned by some SHARE functions. shErrNoDosMemory is returned if no DOS memory is available (in protected mode or Windows). shErrBadDosVersion is returned if the version of DOS does not support the call. shErrFileNotOpen is returned if the File variable is not open. shErrShareNotLoaded is returned if SHARE.EXE or an equivalent is not loaded.

Types

```
DeviceType; = (DevInvalid, DevPrinter, DevDrive);
```

Device types associated with redirection services.

```
LocalStr; = String[15];
```

String type used for the local machine name, and for local redirection names.

```
NetworkStr; = String[127];
```

String type used for the name of a network resource.

```
PCLanOpType; = (LanUnknown, LanRedirector, LanReceiver,
LanMessenger,
               LanServer);
```

Operating modes of an MS-NET or PC LAN workstation.

```
PrnSetupStr; = String[64];
```

String type used for printer setup strings.

Variables

```
DosMajor; : Byte;
DosMinor; : Byte;
```

The major and minor version numbers of MS-DOS. SHARE's initialization block sets these variables.

CancelRedirection

Syntax

```
function CancelRedirection(LocalName : LocalStr) : Word;
```

Purpose

Cancel redirection previously established with RedirectDevice.

Description

LocalName must match the name originally specified in the call to RedirectDevice. This function requires DOS 3.1 or later, an MS-NET compatible network, and that SHARE.EXE be loaded.

A status code is returned in function result. Zero is returned for success; otherwise a DOS error code indicates the reason for failure. See GetExtendedError for a list of error codes.

Example

```
if RedirectDevice(DevDrive, 'Q:', '\\MAIN\DBTEST', '', 2) = 0 then
begin
  Writeln('Drive Q redirected to network');
  if CancelRedirection('Q:') = 0 then
    Writeln('Redirection cancelled for drive Q')
  else
    Writeln('Error cancelling redirection');
end else
  Writeln('Error redirecting drive Q');
```

Assigns drive Q: to the network resource \\MAIN\DBTEST, then cancels the redirection.

See Also

GetRedirectionEntry

RedirectDevice

DosLockRec

Syntax

```
function DosLockRec; (var F; LockPosition, LockLength : LongInt) :  
Word;
```

Purpose

Lock part or all of a file using DOS services.

Description

This function uses DOS call \$5C to lock all or part of a file. It requires DOS 3.0 or later, and that SHARE.EXE be loaded.

The parameter F is untyped to allow any Turbo Pascal file variable to be passed. It is the caller's responsibility to assure that the variable passed is really a file. The file must already be open.

LockPosition specifies the start of the region to lock. The first byte of the file is at position 0. LockLength specifies the number of bytes to lock. It is not an error to lock beyond the end of the file. The same values of LockPosition and LockLength must be used when unlocking the file.

A DOS error code is returned in the function result: zero for success, else a code compatible with those listed under GetExtendedError.

Be sure to remove all locks before closing a file. Failure to do so will lead to unpredictable results.

Example

```
var  
  F : file of Byte;  
  B : Byte;  
...  
if DosLockRec(F, 5, 1) = 0 then begin  
  Seek(F, 5);  
  if IOResult <> 0 then begin  
    Writeln('Seek error');  
    if UnlockDosRec(F, 5, 1) <> 0 then ;  
    Halt;  
  end;  
  B := 12;  
  Write(F, B);  
  if IOResult <> 0 then  
    Writeln('Write error');  
  if UnlockDosRec(F, 5, 1) <> 0 then  
    Writeln('Error unlocking file');  
end else  
  Writeln('Error locking file');
```

Locks the sixth byte of file F, then seeks to that position and writes a new value to the byte there. (Note that the first byte is byte 0.) The lock is removed even if the write to the file fails.

See Also

UnlockDosRec

GetExtendedError

Syntax

~~function GetExtendedError,(var Class, Action, Locus : Byte) : Word;~~

Purpose

Return extended information about the most recent DOS error.

Description

This function requires DOS 3.0 or later. It returns zeros for earlier versions of DOS. Class is the type of the last DOS error (see Error Classes below). Action specifies the recommended action to take (see Recommended Actions below). Locus indicates the device that reported the error (see Error Locus Codes below). The function result returns the most recent DOS error code again. The table of extended error codes that follows can be used to interpret the error codes returned by other functions in the SHARE unit.

Extended Error Codes

\$01 Invalid DOS function number.
\$02 File not found.
\$03 Path not found.
\$04 Too many open files.
\$05 Access denied.
\$06 Invalid handle.
\$07 Memory control blocks destroyed.
\$08 Insufficient memory.
\$09 Invalid memory block address.
\$0A Invalid environment.
\$0B Invalid format.
\$0C Invalid access code.
\$0D Invalid data.
\$0E Unknown unit.
\$0F Invalid disk drive.
\$10 Attempted to remove current directory.
\$11 Not same device.
\$12 No more files.

Critical error codes mapped by DOS

\$13 Disk write-protected.
\$14 Unknown unit.
\$15 Drive not ready.
\$16 Unknown command.
\$17 Data CRC error.
\$18 Bad request structure length.
\$19 Seek error.
\$1A Unknown media type.
\$1B Sector not found.
\$1C Printer out of paper.
\$1D Write fault.
\$1E Read fault.
\$1F General failure.

Codes new to DOS version 3

\$20 Sharing violation.
\$21 Lock violation.
\$22 Invalid disk change.
\$23 FCB unavailable.
\$24 Sharing buffer exceeded.
\$32 Unsupported network request.

GetExtendedError

\$33 Remote machine not listening.
\$34 Duplicate name on network.
\$35 Network name not found.
\$36 Network busy.
\$37 Device no longer exists on network.
\$38 NetBIOS command limit exceeded.
\$39 Error in network adapter hardware.
\$3A Incorrect response from network.
\$3B Unexpected network error.
\$3C Remote adapter incompatible.
\$3D Print queue full.
\$3E Not enough space for print file.
\$3F Print file cancelled.
\$40 Network name deleted.
\$41 Network access denied.
\$42 Incorrect network device type.
\$43 Network name not found.
\$44 Network name limit exceeded.
\$45 NetBIOS session limit exceeded.
\$46 File sharing temporarily paused.
\$47 Network request not accepted.
\$48 Print or disk redirection paused.
\$50 File already exists.
\$52 Cannot make directory.
\$53 Fail on critical error.
\$54 Too many redirections.
\$55 Duplicate redirection.
\$56 Invalid password.
\$57 Invalid parameter.
\$58 Network device fault.
\$59 Function not supported by network.
\$5A Required system component not installed.

Error Classes (Class)

\$01 Out of resource (e.g., disk space or handles).
\$02 Temporary problem (e.g., file locked).
\$03 Authorization problem.
\$04 Internal error in system software.
\$05 Hardware failure.
\$06 System software failure (e.g., missing configuration file).
\$07 Application program error.
\$08 File or item not found.
\$09 Invalid type or format of file or item.
\$0A File or item locked.
\$0B Wrong or bad disk.
\$0C Item already exists.
\$0D Unknown error.

Recommended Actions (Action)

\$01 Retry reasonable number of times, then prompt to abort.
\$02 Retry reasonable number of times with delays between, then prompt to abort.
\$03 Get corrected information from user.
\$04 Clean up (e.g., release locks, close files) and abort application.
\$05 Abort application immediately.

GetExtendedError

\$06 Ignore error.
\$07 Retry after user intervention to correct error.

Error Locus Codes (Locus)

\$01 Unknown.
\$02 Block (disk) device.
\$03 Network.
\$04 Serial device.
\$05 Memory.

Example

```
if GetExtendedError(Class, Action, Locus) <> 0 then begin
  Write('Error location: ');
  case Locus of
    $01 : Writeln('Unknown');
    $02 : Writeln('Disk device');
    $03 : Writeln('Network');
    $04 : Writeln('Serial device');
    $05 : Writeln('Memory');
  end;
end;
```

Reports the locus of the last DOS error.

GetMachineName / GetPrinterSetup

Syntax

```
function GetMachineName; (var MachineName : LocalStr;  
                          var MachineNum : Byte) : Word;
```

Purpose

Return the workstation's machine name and NetBIOS name index.

Description

This function requires DOS 3.1 or later and an MS-NET compatible network.

MachineName is the workstation's machine name. MachineNum is the index into the machine's NetBIOS name table.

The function returns a DOS error code, or zero if successful. It may return zero even if the machine has no defined name. In this case, MachineName will be the empty string and MachineNum will be zero.

Example

```
if GetMachineName(MachName, MachNum) then  
  Writeln('Workstation name: ', MachName)  
else  
  Writeln('GetMachineName function not supported');
```

Reports the name of the current workstation.

Syntax

```
function GetPrinterSetup; (var SetupStr : PrnSetupStr; RDLIndex :  
Word) : Word;
```

Purpose

Return the printer setup string for the specified device in the redirection index list.

Description

This function returns zero if it is successful, otherwise a DOS error code. It requires DOS 3.1 or later and an MS-NET compatible network.

SetupStr returns the current printer setup string. The printer setup string is sent prior to printing each file on the specified network printer. Thus, different stations can customize the operating mode of the printer.

RDLIndex is first established when RedirectDevice is called to redirect the local printer port to a network device. The first redirected resource obtains redirection index 0, the next index 1, and so on. You can get the index by calling GetRedirectionEntry for each index and comparing the returned information with the desired device.

This function returns non-zero if RDLIndex doesn't specify a redirected printer or if the function is not supported.

Example

```
if GetPrinterSetup(SetupStr, 2) then  
  if SetPrinterSetup(NewSetupStr, 2) then begin  
    {Print a file}  
    ...  
    if not SetPrinterSetup(SetupStr, 2) then  
      Writeln('Unable to restore printer setup string');  
  end;
```

Saves the current printer setup string, activates a new setup string, prints a file, then restores the original setup. Assumes that redirection index 2 points to a printer device.

GetMachineName / GetPrinterSetup

See Also

GetRedirectionEntry	RedirectDevice
SetPrinterSetup	

GetRedirectionEntry / GetTempFileName

Syntax

```
function GetRedirectionEntry; (RDLIndex : Word; var LocalName :           
LocalStr;
                                var NetName : NetworkStr; var Parameter
                                : Word;
                                var Dev : DeviceType) : Word;
```

Purpose

Return information about the specified redirection list entry.

Description

This function requires DOS 3.1 or later, an MS-NET compatible network, and that SHARE.EXE be loaded.

RDLIndex is an index into the redirection list. The first redirection entry is number 0.

GetRedirectionEntry returns the other parameters. LocalName is a printer name like 'LPT1' or a drive name like 'G:'. NetName is the name of the network resource associated with the local name. Parameter is the same user-defined value assigned to the redirection entry when it was created. Dev returns either DevInvalid, DevPrinter, or DevDrive. If DevInvalid is returned then the device is not valid (i.e., the index into the redirection list was not mapped).

DOS error codes are returned in the function result. See GetExtendedError for a list of the possible error codes.

Example

```
Index := 0;
while GetRedirectionEntry(Index, LocalName, NetName,
                          UserParam, Dev) = 0 do begin
    if Dev <> DevInvalid then
        Writeln(LocalName, ' is mapped to the network resource ',
NetName);
    inc(Index);
end;
```

Reports on all redirected network resources.

See Also

CancelRedirection

RedirectDevice

Syntax

```
function GetTempFileName; (PathName : PathName;
                            var TempFileName : PathName) : Word;
```

Purpose

Return a file name guaranteed to be unique in the specified directory.

Description

This function requires DOS 3.0 or later.

PathName is an existing drive and directory to hold the temporary file. TempFileName returns the full pathname of the temporary file. The function result returns a DOS error code (see GetExtendedError for a list).

The file is created and immediately closed by this call. The application should refer to the returned name to rewrite or reset the file using an appropriate Turbo Pascal file variable.

Example

```
var
    F : File;
...

```

GetRedirectionEntry / GetTempFileName

```
if GetTempFileName('C:\', TempFileName) = 0 then begin
  Assign(F, TempFileName);
  Rewrite(F, 1);
end;
```

Creates a unique file in the root directory of drive C:, then opens it for writing.

IBMPC LANLoaded / IsDriveLocal

Syntax

```
function IBMPC LANLoaded; (var Lan : PCLanOpType) : Boolean;
```

Purpose

Return True if the IBM PC LAN program is loaded.

Description

If PC LAN is loaded, then the parameter Lan contains a value of type PCLanOpType. This type indicates the mode in which the workstation is operating. Valid modes are LanRedirector, LanReceiver, LanMessenger, or LanServer. See the PC LAN documentation for more information on these operating modes. Novell's Advanced NetWare may pass the test for PC LAN. (It does so if Novell's INT2F TSR is loaded.) When determining the type of a network operating system, check first for Novell. See NETINFO.PAS for an example.

See Also

ShareInstalled

Syntax

```
function IsDriveLocal; (Drive : Byte) : Boolean;
```

Purpose

Determine whether the specified drive is local to the current workstation.

Description

This function requires DOS 3.1 or later.

Drive is the number of the desired drive (0 = default, 1 = A, etc.). It is the application's responsibility to assure that Drive specifies a valid drive number. IsDriveLocal returns True if an invalid drive is specified. The recommended approach to determining whether a drive is valid is to switch to the drive by using DOS function \$0E and then to determine whether the current drive has been updated by using DOS function \$19. Turbo Professional and Object Professional provide the ValidDrive function that encapsulates these calls.

Example

```
Write('Current drive is ');
if IsDriveLocal(0) then
  Writeln('local')
else
  Writeln('a network drive');
```

Reports on the status of the default drive.

See Also

IsFileLocal

IsFileLocal / RedirectDevice

Syntax

```
function IsFileLocal; (var F) : Boolean;
```

Purpose

Determine whether the specified file is local to the current workstation.

Description

This function requires DOS 3.1 or later.

F must be a Turbo Pascal file variable of any type, already opened. IsFileLocal returns True when passed an invalid file variable.

Example

```
assign(F, 'TEST.DAT');
reset(F);
if IOResult <> 0 then Halt;
if not IsFileLocal(F) then
    Writeln('TEST.DAT is on a network drive');
```

Reports whether TEST.DAT is on a network drive.

Syntax

```
function RedirectDevice; (TypeOfDev : DeviceType; LocalName :
LocalStr;

                                NetworkName : NetworkStr; Password :
NetworkStr;

                                Parameter : Word) : Word;
```

Purpose

Associate a local name with a network printer or disk.

Description

This function requires DOS 3.1 or later, an MS-Net compatible network, and that SHARE.EXE be loaded.

TypeOfDev is either DevPrinter or DevDrive to specify printer or drive redirection, respectively.

For printer redirection, LocalName is one of 'PRN', 'LPT1', 'LPT2', or 'LPT3'. For drive redirection, LocalName is a drive designator such as 'F:'. After successful redirection, program references to these names will access network resources rather than local ones.

NetworkName specifies the name of the network resource. For example, to redirect the local drive F: to directory 'WORK' on an MS-Net server named 'MAIN', NetworkName should be set to '\\MAIN\WORK' and LocalName should be 'F:'. The syntax for specifying network directories may vary on different networks.

Password might be required by the network to gain access to the specified resource. Specify an empty string if no password is required.

Parameter is ignored by the network. Any value specified here will be returned by a call to GetRedirectionEntry, perhaps for use by the application.

The function result returns a DOS error code. See GetExtendedError for a list of the possible error codes.

This function also works with Novell's Advanced NetWare. See NETINFO.PAS for an example.

It is good programming practice to restore the initial redirection settings when an application terminates. This is possible by using GetRedirectionEntry at program startup and then resetting any changed values with RedirectDevice at termination. Note, however, that Password values are never returned by GetRedirectionEntry, so to restore a password-protected resource requires that the application have prior knowledge of the password.

See Also

CancelRedirection

IsFileLocal / RedirectDevice

GetRedirectionEntry

SetPrinterSetup / ShareInstalled

Syntax

```
function SetPrinterSetup, (SetupStr : PfnSetupStr, RDLIndex : Word) :  
Word;
```

Purpose

Define a printer setup string for the specified device in the redirection list.

Description

This function requires DOS 3.1 or later and an MS-Net compatible network.

SetupStr is the new printer setup string, which is sent prior to printing each file on the specified network printer. Thus, different stations can customize the operating mode of the printer.

RDLIndex is first established when RedirectDevice is called to redirect the local printer port to a network device. The first redirected resource obtains redirection index 0, the next index 1, and so on. You can get the RDLIndex by calling GetRedirectionEntry for each index and comparing the returned information with the desired device.

This function returns a DOS error code. See GetExtendedError for a list of error codes.

See Also

GetPrinterSetup

Syntax

```
function ShareInstalled; : Boolean;
```

Purpose

Return True if the DOS file-sharing module SHARE.EXE is installed.

Description

This function requires DOS 3.0 or later.

See Also

IBMPCLanLoaded

UnlockDosRec / UpdateFile

Syntax

```
function UnlockDosRec; (var F; FilePosition, FileLength : LongInt) :  
Word;
```

Purpose

Unlock all or part of a file using DOS services.

Description

This function requires DOS 3.0 or later, and that SHARE.EXE be loaded.

The parameter F is untyped to allow any Turbo Pascal file variable to be passed. The variable passed must be an open file. FilePosition specifies the start of the region to unlock and FileLength specifies the number of bytes to unlock. FilePosition and FileLength must exactly match the values used when the region was locked. You cannot, for example, specify

```
Status := UnlockDosRec(F, 0, FileSize(F));
```

to remove all locks at once. Be sure to remove all locks before closing a file. Failure to do so will lead to unpredictable results.

A DOS error code is returned in the function result. See GetExtendedError for a list of the possible error codes.

See Also

DosLockRec

Syntax

```
function UpdateFile; (var F) : Word;
```

Purpose

Flush an open file to disk, assuring that previous writes are saved.

Description

This function requires DOS 2.0 or later. The method used is to force a duplicate of the file handle, then close the duplicate. This is significantly faster than closing and reopening the file. The original file remains open after the call. For some networks, any locks that were in place before the call to UpdateFile are lost after the call. See the discussion of IsamFlushDOS33 in Chapter 5 for more information.

The parameter F is untyped to allow any Turbo Pascal file variable to be passed. It is the caller's responsibility to assure that the variable passed is an open file. If the file is a Turbo Pascal text file, the application should call Turbo's Flush procedure first to assure that the Turbo text buffers are flushed as well.

A DOS error code is returned in the function result. See GetExtendedError for a list of the possible error codes

D. Network Demonstration Programs

There are several network demonstration programs supplied with B-Tree Filer. Each demonstrates a different aspect of network programming and shows how to use the individual network-aware units. All the programs are command line driven.

- NETINFO is the most all encompassing demonstration program. It can be run on any system; it attempts to determine what network is active and to report information about it. The most complete information is provided if you are running on a NetWare system.
- NBSEND implements a method to transfer a group of files between two workstations on a network by using NetBIOS session services.
- NISEND does the same thing as NBSEND, but uses IPX services instead.
- NSSEND does the same thing as NBSEND, but uses SPX services instead.
- SPX2WAY shows how to use SPX services to implement a simple chat-type program. It is somewhat easier to understand than the NSSEND program.
- BINDLIST is a NetWare bindery listing program.
- TTSFILER is an example of how to use NetWare Transaction Tracking Services in a B-Tree Filer application.

NETINFO accepts the following command line syntax:

```
NETINFO [/P]
```

At this time, the only option available is /P, which activates a detailed printer status report on NetWare installations. NETINFO can distinguish between the following network operating systems:

- NetWare
- NetWare with the NetBIOS emulator loaded
- NetBIOS
- PC LAN
- Other or no network

Because so many networks attempt to be compatible (or semi-compatible) with one another, it is possible that NETINFO may become confused when presented with an unknown network. Take a look at the logic used by the DetermineNetwork function in NETINFO.PAS for help in recognizing your target system.

NETINFO reports the most information when NetWare is detected. In this case, it first writes out information about the workstation:

- shell type (NETX or VLM) and version number
- workstation date/time

Then it displays a list of servers with the following information for each:

- the server name and handle
- NetWare version
- the server date/time
- whether transaction tracking is available
- workstation connection number
- workstation broadcast message mode

Finally it displays the drive mapping information, followed by printer information (including the capture flag details if you use the /P option).

Since NetBIOS is a much lower level protocol, the only information NETINFO reports in this case is the workstation machine name and printer setup strings for any network printers.

In all cases, NETINFO reports the device redirection list, since this function can safely be called even without a network. (See GetRedirectionEntry in _8.C for more information about network redirection.)

NBSEND can be used to copy a file from one workstation directly to another, without using the server as an intermediary. It uses NetBIOS services to send the messages.

NBSEND works only if you have a NetBIOS driver or adapter loaded on your system. It is designed to work between two such workstations on a network. The two workstations are differentiated as a "sender" and as a "receiver." It is assumed that the sender will send a group of files to the receiver. The sender can specify the files to send (these files can be anywhere on the workstation's drives and in any directory), however the receiver can only write the transferred files into the current directory (although the full path name is transmitted for each file, NBSEND ignores the path information).

NBSEND accepts the following command line syntax:

```
NBSEND [/Cxxx] [/R] [FileMask [FileMask...]]
```

The /Cxxx option specifies the amount of time in seconds the program waits to receive a connection request from the partner.

On the sending station, call NBSEND with at least one FileMask, where FileMask can specify a single file or a group of files using DOS wildcards. On the receiving station, call NBSEND with the /R option, but without any file masks.

For example, to send all files in the \TEST directory from workstation A to workstation B, enter the following command line on workstation A:

```
NBSEND /C60 \TEST\*.*
```

and the following command line on workstation B:

```
NBSEND /C60 /R
```

These command lines give a 60 second leeway for the two instances of NBSEND to connect.

Each instance of the program adds a unique name to its local NetBIOS name table and then tries to connect to the partner. The sender is the active partner (it sends out periodic open session requests). The receiver is the passive partner (it just listens for the open session request).

Once the NetBIOS session is established, the sender transmits the specified files to the receiver one by one, using the maximum packet size that the two workstations agree on. The receiver overwrites any files it needs to without warning.

Press <Esc> at any time to abort the transfer. The receiver deletes the final partial file.

Once the file transfer is complete, either normally or aborted by the user, the session is closed down and the name added to the NetBIOS name table is deleted.

NISEND can be used to copy a file from one workstation directly to another, without using the server as an intermediary. It uses IPX services to send the messages. As in NBSEND, one workstation must be the sender and the other the receiver.

NISEND accepts the following command line syntax:

```
NISEND [/Cxxx] [/R] [FileMask [FileMask...]]
```

The /Cxxx option specifies the amount of time in seconds the program waits to receive a connection request from the partner.

On the sending station, call NISEND with at least one FileMask, where FileMask can specify a single file or a group of files using DOS wildcards. On the receiving station, call NISEND with the /R option, but without any file masks.

For example, to send all files in the \TEST directory from workstation A to workstation B, enter the following command line on workstation A:

```
NISEND /C60 \TEST\*.*
```

and the following command line on workstation B:

```
NISEND /C60 /R
```

These command lines give a 60 second leeway for the two instances of NISEND to connect.

NISEND sets up an IPX session in the following manner. Two sockets are required, the first is the regular channel for transmitting data and the second is used only as a acknowledgement or handshake socket. The receiver listens on the regular socket for an IPX message. The sender posts a listen event on the handshake socket and then regularly broadcasts a message on the regular data socket. Once the receiver gets the broadcast message, it responds it by sending a message on the handshake socket. The sender gets this acknowledgement message (which contains the receiver's address). The session is then assumed to be set up and the sender starts transmitting files through the data socket. The packet size for transfer is 512 bytes.

Press <Esc> at any time to abort the transfer. The receiver deletes the final partial file.

Once the transfer is complete, either normally or aborted by the user, NISEND cleans up by closing all sockets.

NSSEND can be used to copy a file from one workstation directly to another, without using the server as an intermediary. It uses SPX services to send the messages. As in NBSEND, one workstation must be the sender and the other the receiver.

NSSEND accepts the following command line syntax:

```
NSSEND [/Cxxx] [/R] [FileMask [FileMask...]]
```

The /Cxxx option specifies the amount of time in seconds the program waits to receive a connection request from the partner.

On the sending station, call NSSEND with at least one FileMask, where FileMask can specify a single file or a group of files using DOS wildcards. On the receiving station, call NSSEND with the /R option, but without any file masks.

For example, to send all files in the \TEST directory from workstation A to workstation B, enter the following command line on workstation A:

```
NSSEND /C60 \TEST\*.*
```

and the following command line on workstation B:

```
NSSEND /C60 /R
```

These command lines give a 60 second leeway for the two instances of NSSEND to connect.

NSSEND sets up an SPX session in the following manner. Three sockets are required, the first is the sender's SPX socket, the second is the receiver's SPX socket, and the third is used for an IPX handshake socket. The sender posts an SPX "listen for connection" event on its SPX socket, and then regularly broadcasts an IPX message on the IPX handshake socket. The receiver listens for an IPX message on the IPX handshake socket, and once it receives one it posts an SPX "establish connection" event on its SPX socket. If this is received by the sender, the SPX session is set up. The sender then starts transmitting files through the SPX connection. The packet size for transfer is 512 bytes.

Press <Esc> at any time to abort the transfer. The receiver deletes the final partial file.

Once the transfer is complete, either normally or aborted by the user, NSSEND cleans up by terminating the SPX session and closing all sockets.

SPX2WAY is a simple chat-type program that uses SPX communications to transmit messages from one workstation to another. The user interface is extremely simple so that the underlying use of SPX services is not hidden by too much extraneous interface code.

One workstation is assumed to be in control of the chat session (the sender), the other can be viewed as the 'slave' or receiver. This distinction is only used to initially establish the session. Once it is established, either station can send messages and terminate the conversation.

SPX2WAY accepts the following command line syntax:

```
SPX2WAY [/Cxxx] [/R]
```

The /C option determines the amount of time in seconds the program waits for a connection request from the partner station. The default time is 60 seconds. The /R option must be specified by the receiver. The workstation that controls the conversation does not specify the /R option.

Once the SPX session is set up, either workstation can send a message to the other by typing it and pressing <Enter>. Messages received from the partner workstation are automatically displayed.

Either workstation can terminate the conversation by pressing the <Esc> key. This action causes SPX2WAY to terminate the SPX session and close all sockets.

BINDLIST is a program that exercises the NWBIND unit by listing all the objects in the bindery for a server, and listing all properties for each object.

BINDLIST accepts the following command line syntax:

```
BINDLIST [ServerName]
```

ServerName is optional and specifies the server whose bindery is to be listed. If this parameter is missing, the default server is used.

BINDLIST makes two assumptions. The first is that under NetWare 4.x you must be using bindery emulation on the server. If you are not, then the program fails with an error message. The other assumption is that the calling workstation must be at least attached to the required server. The workstation does not have to be logged on; the bindery can still be accessed, but a reduced report is generated.

Because the listing that BINDLIST produces can be long, you may find it helpful to use the normal DOS redirection commands on the command line to redirect the output to a file, for example:

```
BINDLIST >C:\BINDERY.LOG
```

This redirects the bindery listing for the default server to a file called C:\BINDERY.LOG.

The listing that BINDLIST produces has the following format:

```
A2000018 User          JULIAN
  Static object  Security: write-supervisor; read-logged
  Properties...
  GROUPS_I'M_IN
  Static set    Security: write-supervisor; read-logged
  Contents:     EVERYONE
  LOGIN_CONTROL
  Static item   Security: write-supervisor; read-object
  Contents:     00 00 00 00 00 00 00 00 FF [..... ]
                00 00 FF 00 00 00 00 FF FF [.. ... ]
                FF FF FF FF FF FF FF FF FF [      ]
                FF FF FF FF FF FF FF FF FF [      ]
                FF FF FF FF FF FF FF FF FF [      ]
                FF FF FF FF FF FF FF FF FF [      ]
                FF FF FF FF FF FF FF FF FF [      ]
                5E 0A 0A 08 2F 33 00 00 [^.../3..]
                7F FF FF FF 00 00 00 00 [□ ....]
                00 00 00 00 00 00 00 00 [.....]
                00 00 00 00 00 00 00 00 [.....]
  ...
```

The ID (in hex), type, and name for the object are shown first, followed by whether the object is static or dynamic and the security flags. Then all the properties for that object are listed. The property name is given, followed by its persistence value and its security. The property value is then shown. For set type properties, the object IDs in the set are resolved to their correct names. For item type properties, the property value is shown in a hex dump type display, together with the equivalent ASCII characters.

Please note that a complete listing of all objects and their properties is dependent on the security level of the user using BINDLIST. If the user does not have sufficient rights to the bindery, some objects might be missing and some properties might be hidden.

TTSFILER is an example program that shows how to use Novell's Transaction Tracking Services (TTS) in a B-Tree Filer application. To run it you need access to a Novell NetWare server that is configured for TTS.

The program first creates a new fileblock on the server. You might need to change the name of this demonstration fileblock for your network; alter the constant TestFileName at the beginning of the program. Once the fileblock is created, the program sets the NetWare transactional flag for both the data and index files, and then opens the fileblock as normal.

TTSFILER then goes into a loop adding randomly generated records and their keys. For each record added, a transaction is begun (nwTTSBegin) after the fileblock is locked and prior to the fileblock being updated. Once the record and its keys are added, the transaction is ended (nwTTSEnd) and the fileblock unlocked.

Because the data is generated randomly, there is a small finite chance that one of the records will have a duplicate key, in which case the add key routine fails. The transaction is aborted (nwTTSAbort) and the B-Tree Filer engine is notified by calling BTInformTTSAbortSuccessful.

After all the records are added, the program quickly checks all the transactions to see whether they completed. This is not done for each transaction because the server takes more than 5 seconds to report that a single transaction has completed. Allowing the server to process many transactions asynchronously, one after the other, is more efficient.

At the end of the program, the fileblock is closed and the transactional bit for the data and index files is set off. The output from the program is fairly minimal, just enough to verify the steps TTSFILER executes.

9. Appendix

A. Error Codes

The following table summarizes all error codes that can be returned in the IsamError variable. See the READ.1ST file for additional error codes that may have been added since the manual was printed. There is also a file called ERRORS that the install program extracts into your main B-Tree Filer directory which has the full up-to-date list of error codes.

The format of the error table below is

NNNNN (C)	List of routines that can generate the error
	Brief description of the error

NNNNN is the error number, and C is the error class as returned by BTIsamErrorClass.

For more information about each error, the detailed documentation about each corresponding routine. "General" errors occur in low-level FILER procedures and cannot be correlated with a higher level interfaced procedure except from the context of the calling program. Also be aware of the interfaced variables IsamDOSError and IsamDOSFunc, which can be used to determine the cause of many low level DOS errors. See the description of these variables in Chapter 5.

8000 (4)	user	The range 8000..8999 is reserved for user-defined codes, BONUS units, etc.
9011 (4)	general	Unexpected end of file. This indicates a corrupted dBase file. This error code is assigned to the identifier DEEOF in the DBIMPEXP unit.
9012 (4)	general	Out of memory. This error code is assigned to the identifier DEEOM in the DBIMPEXP unit.
9013 (4)	general	Either the file is not a dBASE file, or the dBase version is not supported (only dBASE III and IV files are supported). This error code is assigned to the identifier DEBV in the DBIMPEXP unit.
9014 (4)	general	Corrupted memo file. This error code is assigned to the identifier DECMF in the DBIMPEXP unit.
9015 (4)	general	Record size too large (> 64KB). This error code is assigned to the identifier DERSTL in the DBIMPEXP unit.
9016 (4)	general	Invalid CType (conversion field type). Generally means that there was a field type in the dBASE file that isn't supported. This error code is assigned to the identifier DEWCT in the DBIMPEXP unit.
9017 (4)	dBaseImport	Error writing type definition include file. This error code is assigned to the identifier DEEWD in the DBIMPEXP unit.

- 9018 (4) dBaseImport, dBaseExport
Error converting a field. This error code is assigned to the identifier DEECF in the DBIMPEXP unit.
- 9019 (4) CompleteDBaseList, CompleteIsamList, dBaseImport, dBaseExport
List header not initialized or bad parameter. This error code is assigned to the identifier DELHNI in the DBIMPEXP unit.
- 9020 (4) general
Too many B-Tree Filer fields to convert. dBase data files have a limit of either 128 or 255 fields. This error code is assigned to the identifier DETMF in the DBIMPEXP unit.
- 9021 (4) general
Invalid field type. This error code is assigned to the identifier DEWFT in the DBIMPEXP unit.
- 9022 (4) general
Field width too large. This error code is assigned to the identifier DEFCTL in the DBIMPEXP unit.
- 9023 (4) general
Too many decimal places specified. This error code is assigned to the identifier DETMD in the DBIMPEXP unit.
- 9024 (4) general
Field type version conflict. You requested a conversion to a dBASE floating type for a dBASE III file, however floating types are a dBASE IV type. This error code is assigned to the identifier DEFTVC in the DBIMPEXP unit.
- 9025 (4) general
Auto relation field is not allowed. You cannot create a dBASE file with an auto relation field. This error code is assigned to the identifier DEARFNA in the DBIMPEXP unit.
- 9026 (4) general
File contains no memo fields. This error code is assigned to the identifier DEFCNMF in the DBIMPEXP unit.
- 9027 (4) general
Error opening dump file. This error code is assigned to the identifier DEEODF in the DBIMPEXP unit.
- 9028 (4) general
Error writing to dump file. This error code is assigned to the identifier DEEWDF in the DBIMPEXP unit.
- 9029 (4) general
Error closing dump file. This error code is assigned to the identifier DEECDf in the DBIMPEXP unit.
- 9030 (4) general
Programming error (you probably passed an invalid value to a routine). This error code is assigned to the identifier DEPE in the DBIMPEXP unit.
- 9031 (4) general
Field name already exists. This error code is assigned to the identifier DEFNAE in the DBIMPEXP unit.

- 9032 (4) dBaseExport
No field defined; your field list is empty. This error code is assigned to the identifier DENFD in the DBIMPEXP unit.
- 9500 (4) general
The range 9500..9899 is reserved for IOResult error codes. Subtract 9500 to get the true IOResult code, then refer to the Turbo Pascal Programmer's Guide or a DOS reference manual.
- 9900 (2) general
Invalid path name. In a Windows environment, this error can also mean that the drive is not ready. The reason for this is that a Windows function call to DOS returns error code 3 if the drive is not ready (normally DOS error code 3 means invalid path name). Therefore it is possible that a retry could execute successfully.
- 9901 (4) general
Too many open files. Use ExtendHandles from ISAMTOOL to expand the application's file handle table or, if that fails, alter the FILES= line in CONFIG.SYS to allow more system-wide file handles.
- 9902 (4) general
Current directory is full.
- 9903 (1) general
File not found.
- 9904 (4) general
Invalid file descriptor. The file handle stored in the variable pointed to by the IsamFileBlockPtr is not recognized by DOS. Either the fileblock was never opened, the fileblock was closed, or there is a memory overwrite in the application.
- 9905 (4) general
Read request exceeds 64KB.
- 9906 (4) general
Write request exceeds 64KB.
- 9907 (4) general
Error returning file size.
- 9908 (4) general
Invalid file access mode.
- 10000 (4) BTInitIsam
Number of page buffers is less than MaxHeight.
- 10001 (3) BTAddRec
Serious I/O error in save mode. The fileblock is still in a usable state but the last operation was not completed.
- 10002 (3) BTDeleteRec
Serious I/O error in save mode. The fileblock is still in a usable state but the last operation was not completed.
- 10003 (3) BTAddKey
Serious I/O error in save mode. The fileblock is still in a usable state but the last operation was not completed.

- 10004 (3) BDeleteKey
 Serious I/O error in save mode. The fileblock is still in a usable state but the last operation was not completed.
- 10005 (3) BtDeleteAllKeys
 Serious I/O error in save mode. The fileblock is still in a usable state but the last operation was not completed.
- 10010 (4) BTOpenFileBlock
 Index file probably corrupt (wasn't closed properly after last modification).
- 10020 (4) BTCreateFileBlock
 Record length out of range (less than 21 bytes).
- 10030 (4) BTOpenFileBlock, BTCreateFileBlock
 Insufficient memory for key descriptors.
- 10040 (4) general, VREC unit
 Insufficient memory to expand the internal variable length record buffer.
- 10050 (4) BTCreateFileBlock
 Invalid number of indexes specified (<0 or >MaxNrOfKeys).
- 10055 (4) BTCreateFileBlock
 Key length out of range (<1 or >MaxKeyLen).
- 10060 (4) BTOpenFileBlock
 Too many indexes (>MaxNrOfKeys). The number of indexes for a fileblock is held in the system record (record 0).
- 10065 (2) general
 Attempt to write to a fileblock opened in read-only mode.
- 10070 (4) general
 File read error. B-Tree Filer requested DOS to read a certain number of bytes from a file. DOS successfully read some bytes, but it was less than the number requested. Common reasons for this error include:
 - Dialog file left corrupted by a previous program run (delete the dialog file at the DOS command line);
 - BTGetRec specified an invalid record number (larger than BTFileLen), this can also happen if you have a corrupted variable length record fileblock;
 - two different programs compiled with different MaxNrOfWorkStations constants are trying to access the same fileblock;
 - a physical hard disk error;
 - a NetWare NETX shell bug (see the conditional compilation define LockBeforeRead in _2.A).
- 10075 (4) general
 File write error. B-Tree Filer requested DOS to write a certain number of bytes to a file. DOS wrote some of the buffer, but it wasn't the amount requested. This could be a hard disk error or the disk could be full.
- 10080 (4) BTCloseFileBlock
 Fileblock is not open.
- 10090 (4) BTCreateFileBlock
 Insufficient memory for an IsamFileBlock variable.

- 10100 (4) BTOpenFileBlock
Insufficient memory for an IsamFileBlock variable.
- 10110 (2) general
Drive not ready.
- 10120 (4) BTOpenFileBlock
Index header corrupted. Cannot calculate key length.
- 10121 (4) BTOpenFileBlock
MaxKeyLen now incompatible with fileblock.
- 10122 (4) BTOpenFileBlock
The page size for the fileblock is greater than MaxPageSize.
- 10125 (4) BTAddKey
Key length too long (> the maximum length for its index).
- 10130 (4) BTPutRec
Attempt to write to record number zero, the system record.
- 10135 (4) BTDeleteRec
Attempt to delete record number zero or a record number greater than the number of records in the file. Record 0 is the system record and cannot be deleted.
- 10140 (4) general
Unexpected DOS error (check IsamDOSError and IsamDOSFunc). Sometimes this error can be generated through a lock error.
- 10150 (4) general
Out of handles when flushing a network fileblock.
- 10160 (4) BTCloseFileBlock
Fileblock not correctly closed. This can be caused by an unlock operation failing (through an attempt to release the workstation number) or by DOS reporting an error when closing one of the files.
- 10164 (4) general
Invalid key number (< 1 or > the maximum index for this fileblock).
- 10170 (4) BTAddRec, BTDeleteRec
Free record list corrupt. Deleted records form a linked list, with the reference to the first deleted record being in the system record (record 0). The links are record reference numbers, and B-Tree Filer has found a reference number which is zero, greater than the number of records or less than -1 (which is used as the free record list terminator). This is probably due to a memory overwrite, or to BTPutRec being called for a reference number after a call to BTDeleteRec for the same reference number.
- 10180 (4) general
Attempt to repair fileblock failed.
- 10190 (4) ExtendHandles
Requires DOS 3.3 or later.
- 10191 (4) ExtendHandles
Insufficient memory for new file handle table.
- 10192 (4) ExtendHandles
Unable to obtain new file handle table from DOS.

- 10200 (1) BTFindKey
No matching key found.
- 10205 (1) BTGetRecReadOnly
Data record is locked, but the record has been read (except for the first four bytes).
- 10210 (1) BTSearchKey
No key found and no larger keys are available in the index.
- 10215 (1) ReIndexFileBlock, Reorg(V)FileBlock, Rebuild(V)FileBlock
The data file is defective. RestructFileBlock must be called to recreate it.
- 10220 (1) BTDeleteKey
Key to delete was not found.
- 10230 (1) BTAddKey
Cannot add duplicate key. If the index is primary (duplicates are not allowed) then there is a key in the index that already has the value you are trying to add. If the index is secondary (duplicate keys are allowed) then you are trying to add a key and reference number combination that already exists.
- 10240 (1) BTNextDiffKey
No larger key found.
- 10245 (1) BTPrevDiffKey
No smaller key found.
- 10250 (1) BTNextKey
No larger key found.
- 10255 (1) BTNextKey
Sequential access not allowed.
- 10260 (1) BTPrevKey
No smaller key found.
- 10265 (1) BTPrevKey
Sequential access not allowed.
- 10270 (1) BTFindKeyAndRef
No matching key and record number found.
- 10280 (1) BTGetApprKeyAndRef
Index empty.
- 10285 (1) BTGetApprRelPos
Index empty.
- 10306 (2) BTOpenFileBlock
Network fileblock cannot be opened because the maximum number of workstations have the fileblock open. Increase MaxNrOfWorkstations.
- 10310 (4) BTInitIsam
Network initialization error. The B-Tree Filer network detection code did not find the specified network.
- 10315 (4) BTExitIsam
Network exit error. This error should not occur.

10322 (4) BTCloseFileBlock
Attempt to remove read lock failed.

10323 (4) BTCloseFileBlock
Attempt to remove record lock failed.

10330 (2) BTLockFileBlock, BTLockAllOpenFileBlocks
Attempt to lock fileblock failed.

10332 (2) BTReadLockFileBlock, BTReadLockAllOpenFileBlocks
Attempt to read lock fileblock failed.

10335 (2) BTLockRec
Attempt to lock record failed.

10337 (4) BTLockRec
Insufficient memory to expand lock list.

10340 (4) BTUnlockFileBlock, BTUnlockAllOpenFileBlocks
Attempt to unlock fileblock failed.

10341 (4) general
Attempt to remove fileblock read lock failed.

10342 (4) BTOpenFileBlock
Attempt to unlock fileblock failed.

10345 (4) BTUnlockRec
Attempt to unlock record failed.

10355 (2) BTOpenFileBlock
A lock prevents the operation.

10356 (4) BTOpenFileBlock
Insufficient memory for network support record.

10360 (2) BTOpenFileBlock
Cannot repair the dialog file because other workstations have the fileblock open.

10390 (2) BTFindRecRef
Unexpected DOS error occurred (check IsamDOSFunc and IsamDOSError).

10397 (2) general
Fileblock is read-only or a lock prevents the operation.

10398 (4) general
Fileblock must be locked for this operation.

10399 (2) general
A lock prevents the operation.

10410 (1) RestructFileBlock, Reorg(V)FileBlock, Rebuild(V)FileBlock
Data file not found.

10411 (4) RestructFileBlock, ReIndexFileBlock, Reorg(V)FileBlock,
Rebuild(V)FileBlock
Insufficient memory for work buffer.

- 10412 (4) RestructFileBlock, ReIndexFileBlock, Reorg(V)FileBlock,
 Rebuild(V)FileBlock
 Section length exceeds 64KB.
- 10415 (4) general, VREC unit
 Too many sections in a variable length record (total length exceeds \$FFF0 bytes).
- 10420 (4) BTGetApprKeyAndRef, BTGetApprRecRef
 Relative position or scale invalid.
- 10425 (4) BTGetApprRelPos, BTGetApprRecPos
 Relative position or scale invalid.
- 10430 (4) general
 Fileblock repair not allowed in read-only mode.
- 10435 (4) BTInitIsam
 When using EMS, index page larger than 16KB.
- 10440 (4) BTOpenFileBlock
 Cannot create dialog file in read-only mode. The dialog file needs to be created (or recreated),
 but the fileblock is open in read-only mode.
- 10445 (4) general
 Fileblock pointer or descriptor corrupted. Either the IsamFileBlockPtr variable is nil, or the
 pointer does not point to an IsamFileBlock variable. Either the fileblock has never been
 opened, or it has been closed, or there is a memory overwrite in the application.
- 10446 (4) general
 B-Tree Filer was called recursively from a user function such as a BuildKey routine. Such a
 recursive call is not allowed when EMS page buffers are used.
- 10450 (4) BTInitIsam
 BTInitIsam was called twice.
- 10455 (4) general
 BTInitIsam was not called. The B-Tree Filer engine is not active.
- 10460 (4) RestructFileBlock, ReIndexFileBlock, Reorg(V)FileBlock,
 Rebuild(V)FileBlock
 Reorganization was aborted by the progress status routine (IsamOK was set to False).
- 10465 (4) RestructFileBlock, Reorg(V)FileBlock, Rebuild(V)FileBlock
 Restructure aborted. Both the DAT and SAV files already exist.
- 10470 (4) ReIndexFileBlock, Reorg(V)FileBlock, Rebuild(V)FileBlock
 Reindexing was aborted by the BuildKey routine returning with IsamOk set to False. This is
 because it could not create the required key.
- 10475 (4) RestructFileBlock, Reorg(V)FileBlock, Rebuild(V)FileBlock
 Restructuring was aborted by the ChangeDatS routine returning with IsamOk set to False. This
 is because it could not restructure the current record.
- 10480 (4) BTOpenFileBlock
 The network's locking routine is not functioning properly (a portion of the dialog file was
 successfully locked twice. Either no network is present, or the wrong one was specified by
 BTInitIsam.

B. Converting from Borland's Database Toolbox

Because one of the original goals of B-Tree Filer was to remove the limitations of the Database Toolbox, it is likely that many users of B-Tree Filer will be converting applications previously written to use the Toolbox. The B-Tree Filer programming interface and naming conventions are similar to those of the Toolbox, making conversion as painless as possible. Nevertheless, the improvements in B-Tree Filer could not occur without some change. This section documents the important differences. Of course, B-Tree Filer offers many added capabilities as well. To learn more about those, please refer to the prior reference sections of this manual.

This section assumes that the first step in your conversion will be to convert to a single-user application using B-Tree Filer. This approach gets your application up and running as quickly as possible with a minimum of debugging. Making this assumption also allows us to focus on the real differences between B-Tree Filer and the Database Toolbox and not get sidetracked by the new issues of multi-user programming. When you are ready for a multi-user conversion, see Chapter 4 for complete information about B-Tree Filer's multi-user capabilities.

For the remainder of this section, Borland's Database Toolbox is abbreviated as "Dbox," and TurboPower's B-Tree Filer as "Filer." Dbox refers to Borland's version for Turbo Pascal 4.0 and later, which provides some new capabilities compared to earlier versions. Just the routines from Borland's low-level TACCESS.PAS unit are referred to; the higher-level unit TAHIGH.PAS poses similar conversion questions.

There are three categories of differences to consider when converting a Dbox application:

- routines with different names or calling conventions
- routines with different behavior
- data and index file formats

As a result of these differences, you'll need to make various changes to your source code and convert your data files to switch over to Filer. Each of these is discussed later in this section.

There is one suggestion that probably goes without saying. Before you start your conversion, make a complete backup copy of your existing source and data files. In case something goes wrong, you'll feel much better if you have them.

Naming and Calling Conventions

The first change you need to make is to remove TACCESS from your USES statements and replace it with FILER. Be sure to replace it in all units--leaving even one reference to TACCESS in your program can lead to duplicate identifier names and the possibility for an endless variety of bugs that are very difficult to identify.

The names of Filer's routines are similar to those of Dbox, but Filer prefixes almost all of its routine names with the letters 'BT'. In some cases (for example, AddRec), you'll just need to add the prefix characters to get the Filer name (BTAddRec). In other cases, the names are quite different. The following list shows naming differences between Dbox and Filer that go beyond the initial prefix letters.

Dbox	Filer	Category
DataFile/IndexFile	IsamFileBlockPtr	Type
MakeFile/MakeIndex	BTCreateFileBlock	Procedure
OpenFile/OpenIndex	BTOpenFileBlock	Procedure
CloseFile/CloseIndex	BTCloseFileBlock	Procedure
FlushFile/FlushIndex	BTFlushFileBlock	Procedure
OK	IsamOK	Variable
TaStatus	IsamError	Variable

The first five differences stem from Filer's grouping of the data and index files as a single logical entity, the fileblock. Each DataFile and all associated IndexFiles will be replaced with a single IsamFileBlockPtr. Similarly, each call to MakeFile and all associated calls to MakeIndex will be replaced with a single call to BTCreateFileBlock. The same applies to each of the other file management routines. Filer allows 100 indexes or more for each fileblock, and uses just a single file handle to manage all of the indexes. Filer's use of a pointer to refer to each fileblock cuts the use of global data space compared to Dbox.

The final two rows reflect an attempt to make the identifiers names more explicit and self-consistent. If you don't want to rename all references to OK and TaStatus in your program, there's a fast fix. Just edit FILER.PAS and insert the following lines immediately after the declaration for IsamError:

```
OK : Boolean absolute IsamOK;
TaStatus : Integer absolute IsamError;
```

Then your references to OK and TaStatus will get the values in Filer's variables. Remember, however, that the error codes returned in IsamError are completely different from those of TaStatus. See "Toolbox Behavior" later in this section for further details.

To minimize the manual effort required to perform name substitutions, a program named UPGRADE.EXE is provided. It automatically replaces Dbox's names with those of Filer. It also inserts comments in the source code to indicate where manual conversion efforts should be focused. A substitution file named DBOX.UPG is used to specify the exact replacements. DBOX.UPG is a simple text file; you can view or modify it with a standard text editor.

The UPGRADE program is described in more detail in Appendix C. The typical command line you'll use to upgrade a Dbox program is

```
UPGRADE /A /O OutputDir /S DBOX.UPG ProgramName
```

This upgrades all units of the program ProgramName, sending modified output to the directory OutputDir, using the substitution file DBOX.UPG.

Probably the most time-consuming portion of your conversion will result from the fact that Filer stores all indexes in a single file. Because of this, every Filer routine that deals with keys has an additional parameter that specifies which index is affected. Here is an example of one pair of corresponding routines.

Dbox

```
procedure AddKey(var IndexF : IndexFile; var DataRef : LongInt; var
Key);
```


Filer

```

procedure BTAddKey(IFBPtr : IsamFileBlockPtr; Key : Word; DataRef :
LongInt;
                    Key : IsamKeyStr);

```

In making the conversion, you'll want to assign unique sequential numbers (starting with 1) to each IndexFile associated with a given DataFile. Then just replace the first parameter to each existing index routine with the new fileblock name followed by the number you've assigned to that index file. All of the following Dbox routines must be modified in this manner: AddKey, DeleteKey, FindKey, SearchKey, NextKey, PrevKey, and ClearKey.

The preceding procedure declarations point out another difference. Whereas Dbox uses an untyped VAR parameter for the key string, Filer specifies a typed value parameter. In this case, Filer's approach is less restrictive and should not require you to modify existing applications. For a few specific key routines, however, there are important behavioral differences that are described in "Toolbox Behavior" later in this section.

Another difference between Dbox and Filer lies on the border between syntactic and behavioral. Dbox requires the existence of an include file, TACCESS.DEF, which specifies various constants describing the configuration of the B-tree. Filer does not use such an include file. The following table shows the constants specified by TACCESS.DEF and the action you should take for each one.

MaxDataRecSize

not used by Filer. Just refer to the SizeOf() your data record wherever a record length is required.

MaxKeyLen

edit FILER.CFG to specify the desired value in a constant of the same name.

PageSize

FILER.CFG defines two similar constants, CreatePageSize and MaxPageSize. Both have the value 62. It is not recommended that you change them, but you can. You can probably set Filer's constants to the same value you used for Dbox. Be sure to read the description of these constants in Chapter 5 before changing FILER.CFG.

PageStackSize

not used by Filer. The index page buffer size of Filer is determined at runtime. This allows applications to take full advantage of available memory. With Filer, the page stack can buffer more than 64KB and use expanded (EMS) memory when available. See BTInitIsam in Chapter 5 for more information.

Order

not used by Filer. This constant is just one-half of PageSize.

MaxHeight

FILER.CFG defines such a constant having the value 8. It is not recommended that you change it, but you can. You can probably set Filer's constant to the same value you used for Dbox. Be sure to read the description of constant MaxHeight in Chapter 5 before changing FILER.CFG.

The reason you shouldn't change CreatePageSize, MaxPageSize, and MaxHeight is this: a frequent bug in Dbox applications stems from databases that grow beyond the size limit implicitly calculated from CreatePageSize, MaxPageSize, and MaxHeight. Since the overhead of using larger values for

these constants is small, why not select them so you never need to worry about this problem? The default values in Filer allow up to 68 billion entries in the B-tree, with a minimum index file size of about 6KB and minimum page buffer heap space of about 20KB (when EMS is used, the minimum is reduced to just a few hundred bytes). If the space usage is a concern, the first constant you should consider changing is MaxKeyLen, which controls the overhead in a fairly linear fashion. If this isn't enough, change CreatePageSize, MaxPageSize, and MaxHeight only with great caution.

Toolbox Behavior

In certain situations, there are differences in the behavior of Dbox and Filer. These are discussed in the order you would encounter them as you trace the logical flow of your program. This list summarizes the more detailed discussion that follows:

- Filer's minimum data record size is 21 bytes compared to Dbox's 18.
- Filer uses a single fileblock to describe the Dbox DataFile and related IndexFiles.
- FileBlocks are allocated on the heap, while DataFiles and IndexFiles are generally stored in the data segment.
- Filer requires a call to BInitIsam to dynamically allocate page space; Dbox allocates space based on a constant.
- Filer never halts after errors; you must check IsamOK after each call to a Filer routine. Filer's error codes differ from Dbox's, but both agree in assigning zero to success.
- After you create a new database with Filer, you must explicitly open it. Dbox leaves the database open after creating it.
- When you open an existing database with Filer, you don't need to specify the record size or the keys again.
- Dbox modifies the key string passed as a parameter to FindKey; Filer doesn't.
- Dbox returns the associated record number when DeleteKey is called to delete a primary (non-duplicate) key; Filer doesn't.
- Dbox's EraseFile and EraseIndex routines take *open* file variables as parameters; Filer's DeleteFileBlock routine takes a DOS pathname and erases the data and index files, presumed to be closed.

Both Dbox and Filer require that you specify a record structure for the data stored in each data file. The minimum acceptable size for a Dbox record is 18 bytes; for Filer it is 21 bytes. If your record is smaller than 21 bytes, you'll need to pad it to make up the difference. (The lower limit is set by the information stored in the reserved record--number 0--of the data file. Filer simply needs to store more there to manage reliable multi-user databases.) The record size change is handled automatically when you convert your data file, as described in "Data and Index File Formats" later in this section.

The biggest behavioral difference is Filer's concept of a fileblock, which groups the data file and all associated index files in one logical entity. A fileblock is similar to Dbox's high-level DataSet, but the fileblock does not limit the number and type of keys as does the DataSet. Converting to fileblocks requires the syntactic changes already described. If the basic operations for managing your database (adding a record and all its keys, modifying a record, and so on) are isolated in modular fashion, you'll have no trouble in making the IndexFile to fileblock transition. If not, you'll have more edits to do, but the difficulty of conversion is no greater.

Filer makes more frequent use of pointers in its management of data. Whereas DataFile and IndexFile variables use space in the segment where they are declared (about 150 and 200 bytes,

respectively), Filer's IsamFileBlockPtr consumes only four bytes where it is declared and allocates the actual storage space on the heap. This approach reduces the strain on the 64K data segment. If your application restricts the heap with the `{ $M }` compiler directive, you may find that you need to increase the maximum heap space during the conversion.

Although both Dbox and Filer store the B-tree page buffers on the heap, Filer dynamically allocates it while Dbox sets the size based on the constant `PageStackSize`. With Filer's approach, application performance improves automatically when more system memory is available. Filer's method also lets it allocate page buffers larger than 64KB, leading to even better buffering when plenty of memory is available.

Another important difference occurs in the way the two toolboxes handle errors. Dbox halts for many errors, and returns with `OK` set to `False` for others. Filer never halts as a result of an error. It consistently sets `IsamOK` to `False`, and initializes `IsamError` to a specific value no matter what error occurs. Filer therefore puts more responsibility to check for errors on the programmer, but offers much more flexibility in reacting to errors. Dbox isn't very explicit about what situations lead to fatal errors. You should add a check of `IsamOK` after *every* Dbox call when you perform the conversion.

The Dbox manual doesn't say much about the `TaStatus` variable either. This variable holds a code that refers to the most recent error. Because Filer returns a completely different set of error codes in `IsamError`, you'll need to map the numbers accordingly wherever your program refers to `TaStatus`. See Appendix A for a complete list of the Filer error codes. Note that the value 0 does indicate success in both cases.

Because Filer doesn't halt after errors, there is no need for an error handling routine like Dbox's `TAErrorProc`. You may want to move some of the contents of any existing error routine into a general purpose exit procedure. You should remove all references to the variable `TAErrorProc`.

The remaining items refer to specific Dbox procedures.

`OpenFile` and `OpenIndex` both require that you respecify the data record length or key length. Filer's `BTOpenFileBlock` routine does not allow this, because the appropriate information is already stored in the data file header. The two main parameters to `BTOpenFileBlock` are the fileblock pointer variable, and the root name of the data and index files to open. Several additional boolean parameters to `BTOpenFileBlock` specify whether the fileblock is being opened for network operation, read-only mode, or a special safety mode called save mode. During initial conversion, just set all of these parameters to `False`.

The `FindKey` routine in Dbox passes the key to search for as a `VAR` parameter, and uses the specified variable as a buffer area. As a result, the variable passed is overwritten when the search fails. Filer passes the key as a value parameter, which means that the caller's parameter is never modified. This difference requires no conversion activity unless your application relies on the modified value returned by `FindKey` when the search fails. If so, you should call Filer's `BTSearchKey` routine when the `FindKey` search fails. `BTSearchKey` does return the nearest key string to the caller.

The `DeleteKey` routine in Dbox returns the record number associated with the deleted key if the index file contained primary (non-duplicate) keys. `BTDeleteKey` in Filer does not return the record number. It expects you to have determined the record number through a previous find key operation.

The EraseFile and EraseIndex routines take as parameters an open DataFile or IndexFile, respectively. Filer's BTDeleteFileBlock requires that the associated fileblock already be closed. It takes a DOS pathname as a parameter, and adds the 'DAT' and 'IX' extensions to that name before deleting the files.

Data and Index File Formats

Although both Filer and Dbox store data records one immediately after the other in the data file, the formats differ in one important respect. Both reserve the very first record of the data file to hold system information, and the organization of that information differs for the two. Hence, you must convert your Dbox data files in order to use them with Filer.

Practically speaking, the data file conversion doesn't cost anything, because the indexes must be totally rebuilt anyway. Filer's method of storing all indexes in a single file means that the existing Dbox index files are useless. They are rebuilt from scratch during the conversion.

Filer offers the perfect tool to perform the conversion--the REORG unit described in _6.E. It is not possible to write a general purpose utility to convert your files, because that would require knowing how your keys are defined. However, the Convert program is a useful template to show how to write your own conversion program in just a few lines of code. Convert is in the file CONVERT.PAS, which is in the \FILER\TOOLS directory.

```

program Convert;
uses
  Filer, Reorg;
type
  DataRecord =
    record
      RecDeleted : LongInt;
    this}
      LastName : String[20];
      CustNum : String[10];
      {...}
    end;
const
  DataFileName = 'MYDATA';
  NrOfKeys = 2;
data file}
var
  IID : IsamIndDescr;
  Pages : LongInt;
  RecordsAdded : LongInt;
added}
  RecordsKeyed : LongInt;
keyed}
{$F+}
function BuildKey(var DatS; KeyNr : Integer) : IsamKeyStr;
begin
  if RecordsKeyed = 0 then
    Writeln;
  with DataRecord(DatS) do
    case KeyNr of
      key}
        1 : BuildKey := CustNum;

```

```

        2 : BuildKey := LastName;    {!!}
        {...}
    end;
    {Keep status counter running}
    inc(RecordsKeyed);
    if RecordsKeyed and 15 = 0 then
        Write('^M, RecordsKeyed);
    end;

function ChangeDat(var DatSold; var DatSNew; Len : Word) : Boolean;
begin
    if LongInt(DatSold) = 0 then begin
        {Record hasn't been deleted}
        ChangeDat := True;
        DataRecord(DatSNew) := DataRecord(DatSold);
        {Keep status counter running}
        inc(RecordsAdded);
        if RecordsAdded and 15 = 0 then
            Write('^M, RecordsAdded);
        end else
            {Record is deleted, don't add it}
            ChangeDat := False;
    end;
    {$F-}

procedure InitIID;
begin
    {!! Specify each index type here}
    IID[1].KeyL := 10;                {Maximum length of key string}
    IID[1].AllowDupK := False;        {False for a primary key}
    IID[2].KeyL := 20;
    IID[2].AllowDupK := True;         {True for a secondary key}
end;

begin
    Pages := BTInitIsam(NoNet, 10000, 0); {Allocate heap space for
page buffers}
    if not IsamOK then begin
        Writeln('Not enough memory available');
        Halt;
    end;
    RecordsAdded := 0;
    RecordsKeyed := 0;
    InitIID;
    ReorgFileBlock(DataFileName, SizeOf(DataRecord), NrOfKeys,
                    IID, SizeOf(DataRecord),
                    @BuildKey, @ChangeDat);

    Writeln;
    if IsamOK then
        Writeln(RecordsAdded, ' records converted')
    else
        Writeln('Convert failed. IsamError = ', IsamError);
end.

```

The main things you need to add to the template are the declaration of your data record, the function to build each key string, and the initialized structure that describes each key. Lines that you need to modify are indicated with double exclamation points in the listing.

If you need to use non-default values of MaxKeyLen, CreatePageSize, MaxPageSize, or MaxHeight, be sure to modify FILER.CFG for the correct values before you compile Convert.

After you customize and compile the Convert program, copy your existing Dbox data file onto a new file with the extension 'DAT' and a name consistent with the one you specified in Convert. The REORG unit requires the 'DAT' extension; anyway, you'll want to keep your original files safely hidden away in case something goes wrong. Then just run Convert. It reports status information while it runs. When it's done, you have new data and index files ready to run with B-Tree Filer.

C. Converting from earlier versions of B-Tree Filer

This section describes the differences between B-Tree Filer version 5.20 and earlier versions. The major changes and enhancements can be summarized as follows:

Single-user and Network Routines Merged

All routines in the FILER and VREC units now automatically differentiate between single-user and network fileblocks. This makes it easier and cleaner to write applications that work either way, and also easier to write higher level routines and objects that manage fileblocks in a general fashion. Most of the routines in these units were renamed to avoid ambiguity within existing applications. All such routines now begin with the 'BT' prefix. The process of converting your applications is described below.

EMS Support

In real mode, the index buffers of B-Tree Filer can now be completely or partially allocated from expanded memory, thus promising higher performance and freeing more of the DOS 640K for other purposes. This is done via the new BTInitIsam function, which combines the functions of the older InitNetIsam and GetPageStack routines.

Read-only Modes

The new fileblock opening procedure, BTOpenFileBlock, accepts new parameters ReadOnly and AllReadOnly. When ReadOnly is True, the current workstation is prevented from modifying the fileblock, and the FILER unit itself won't attempt to change the fileblock when closing it. When AllReadOnly is True, no workstation can modify the fileblock. B-Tree Filer uses this flag to optimize its own page buffering. The first option is useful for managing fileblock security; the second one is especially useful for CD-ROM applications.

As a result of these changes, B-Tree Filer 5.20 and later versions are not source code compatible with earlier versions. For example, the AddRec and AddNetRec routines are now replaced with a single procedure, BTAddRec, which works like one or the other depending on how the fileblock was opened. Two methods are provided to upgrade your earlier applications.

ISCOMPAT and VRCOMPAT Units

ISCOMPAT is for FILER unit compatibility and VRCOMPAT is for VREC compatibility. By adding these to the Uses statements of your existing applications, your program should compile and run as before, with just a few exceptions. Although these units provide the easiest technique to get up and running with Filer 5.20 or later, you don't get any of the advantages of the cleaner calling interface, so it is not recommended to use them.

Some upgrade problems are not solved by using ISCOMPAT and VRCOMPAT. The following procedures were moved to the new ISAMTOOL unit. Add ISAMTOOL to your Uses statement if you call any of these routines.

```
IsamErrorMessage  
ExtendHandles
```

The following procedures no longer exist.

```
PreAllocateFileblock  
PreAllocateVRecFileBlock  
WSNrLogIn  
WSNrLogOut
```

The following variables are now in the ISCOMPAT unit, but their interpretation is somewhat different than in B-Tree Filer 5.06 (the only previous version in which they were available).

```
InternalDosError - set for every DOS call, not just 10140 error
calls
InternalDosFunction - now word instead of byte, initialized for
every call
```

The following typed constants are no longer supported.

```
AbortReorg
NextDontUseKey
```

Both of these affect the operation of the REBUILD, REORG, VREBUILD, and VREORG units. You can get the effect of AbortReorg by setting IsamReXUserProcPtr to the address of a routine that checks for a user request to stop the reorganization. This routine then simply sets IsamOK to False to abort the operation. There is no one-to-one replacement for NextDontUseKey. If you don't want to add keys for a given index during a rebuild, set AddNullKeys to False and have the BuildKey routine return an empty string for each record in that index. Or call BTDeleteAllKeys for that index after reorganization is complete. Note, however, that you should now be using the RESTRUCT and REINDEX units.

A number of FILER's internal identifiers were changed or removed from the interface section. It's unlikely that these will affect you, but here is a list of the identifiers you can no longer access:

```
const      DummyFillChar
procedure  ExpandFileBlock
var        FuncIsamChangeAttrToShareable
var        FuncIsamExitNet
var        FuncIsamLockRecord
var        FuncIsamUnLockRecord
var        IsamDriveNotReadyError
var        IsamInitializedNet
var        IsamForceFlushOfMark
var        IsamLockError
type       IsamLockedListBlock
type       IsamLockedListBlockPtr
type       IsamLockedListElement
var        IsamNetEmu
var        IsamNetRequested
var        IsamNrOfWS
var        IsamOFBLPtr
var        IsamPageStackSize
var        IsamSRlPtr
type       IsamStackRec
type       IsamStackRecPtr
const      LockedListBlockSize
const      NetFileAttr
const      NovellFileAttr
```

UPGRADE Program

UPGRADE is a small utility that applies multiple identifier name substitutions to Turbo Pascal programs or units. It reads a substitution file (FILER.UPG in this case) and then parses the specified program and makes replacements as appropriate. UPGRADE can thus automatically handle most of the rote work of moving to B-Tree Filer 5.20 or later versions. It also inserts comments into the resulting source text where changes will require additional effort on your part.

You should not use the ISCOMPAT and VRCOMPAT units if you're going to run UPGRADE on your application.

Call UPGRADE as follows:

```
UPGRADE [options] ProgramName[.PAS]
```

The following options can be specified:

```
/A          upgrade all units listed in Uses statement
/O OutputDir specify the drive or directory to receive upgraded
files
/S SubstFile specify a substitution file (default FILER.UPG)
```

UPGRADE automatically finds FILER.UPG if it's in the current directory, in the directory where UPGRADE.EXE is found (DOS 3.0 or later), or on the DOS PATH. You can specify a different substitution file pathname with the /S option.

Specify the /A option to upgrade an entire program, as opposed to a single unit. UPGRADE then processes all units listed in the main program's Uses statement, as well as any units listed in the Interface Uses statement of any unit. It does not process units listed only in an Implementation Uses statement.

If you don't specify an output directory, UPGRADE overwrites the original files with the modified output. The original files are renamed with the extension .S00, unless there is already a file with that extension, in which case .S01, .S02, and so on, are used. You should send the modified output file to a separate directory.

UPGRADE substitutes without regard for normal Pascal scope rules. For example, if you happen to have declared a local variable named AddKey, UPGRADE changes its name to BTAddKey, even in your declaration. You can scan through FILER.UPG to see whether this effect will cause problems for you. Such substitutions don't affect the code generated by the compiler, but could affect the readability of the program.

Wherever UPGRADE knows that you'll need to make additional changes, it inserts a comment of the form:

```
{**new param ISOLock:Boolean**}
```

You can search for these comments and make the appropriate changes manually, or just recompile your program and let the compiler find remaining issues. In some cases, the comments are for your information only and do not require a change in order to compile the program. See FILER.UPG for a list of the comments. Note that FILER.UPG is just a text file; you can modify it to perform additional substitutions or use different comments.

Error Codes

If your application reacts to specific IsamError values, be sure to compare the codes you use to the values in Appendix A and update them accordingly. B-Tree Filer now uses common error checking routines at the entry and exit points of almost all of its interfaced routines. As a result, many errors that previously generated differing codes for different procedures (e.g. 10165..10179, Invalid Key Number) now return the same code in all cases (10164 for this example).

The error class 99 (unknown error) was removed. A value of 4 (hard error) is now returned by BTIsamErrorClass in case of an unknown error.

Other Minor Changes

BTCreateFileBlock leaves the fileblock closed and unallocated on the heap, unlike MakeFileBlock.

The typed constant SearchForSequentialDefault now has a default value of True rather than False.

The maximum number of indexes per fileblock was reduced from 750 to 254. This new limit, which still exceeds practical requirements by a large margin, provides for a simpler and faster internal design.

In versions of B-Tree Filer prior to 5.05, index routines automatically truncated key strings whose length exceeded the maximum for a given index. Now, BTAddKey returns error 10125 if the key is too long. BFindKey (and related key searching routines) no longer truncate the specified search key; if the search key is longer than the key in the index file, an exact match is not returned.

Workstation numbers are now determined automatically by B-Tree Filer when BOpenFileBlock is called. Different numbers are allocated for different fileblocks. Use BTGetInternalDialogID (described in Chapter 5) to get the workstation number for a particular open fileblock.

Identifiers

A

AbortSort (MSORT) 222
AbortSort (MSORTP) 231
AddBrowseCommand 277
AddFieldNode 242
AddNullKeys 81
AdjustHorizOfs 323
AdjustWindow 291
ASCIIZeroKeys 36
AutoScaleMouse 269
AutoSort 223
AutoSortInfo 225

B

BadOPBrOptions 290
BcdToKey 205
BiggestDataItem 221
BKtype 275
BKXxx constants 273
BRCurrentlyLocked 285
BRFilterError 285
BRLRowElitStrArr 286
BRLRowElitString 286
BRNoFilterResult 285
Browse 278
BrowseAgain 280
BrowseHelpIndex 276
BrowseHelpPtr 276
BrowseKeyPtr 276
BrowseKeySet 274
BrowseMouseEnabled 269
BrowseMousePage 269
BrowseReadKey 282
BRUserStatStart 285
BTAddKey 89
BTAddRec 90
BTAddVariableRec 179
BTAdjustVariableRecBuffer 179
BTaReclsLocked 90
BTClearKey 91
BTCloseAllFileBlocks 91
BTCloseFileBlock 92
BTCreateFileBlock 93
BTCreateVariableRecBuffer 180
BTDataFileName 94
BTDatRecordSize 94
BTDeleteAllKeys 95
BTDeleteFileBlock 96
BTDeleteKey 97
BTDeleteRec 98
BTDeleteVariableRec 181
BTExitIsam 99
BTFileBlockIsLocked 100
BTFileBlockIsOpen 101

BTFileBlockIsReadLocked 102
BTFileLen 103
BTFindKey 103
BTFindKeyAndRef 104
BTFindRecRef 106
BTFlushAllFileBlocks 106
BTFlushFileBlock 107
BTForceNetBufferWriteThrough 107
BTForceWritingMark 108
BTFreeRecs 109
BTGetAfterNextUsedAddRecRef 109
BTGetAllowDupKeys 110
BTGetApprKeyAndRef 111
BTGetApprRecPos 112
BTGetApprRecRef 112
BTGetApprRelPos 113
BTGetInternalDialogID 114
BTGetKeyLen 115
BTGetNextUsedAddRecRef 115
BTGetRec 116
BTGetRecordInfo 117
BTGetRecReadOnly 118
BTGetSearchForSequential 118
BTGetStartingLong 119
BTGetVariableRec 181
BTGetVariableRecLength 182
BTGetVariableRecPart 182
BTGetVRecPartReadOnly 183
BTGetVRecReadOnly 183
BTIndexFileName 119
BTInformTTSAbortSuccessful 120
BTInitIsam 121
BTIsamErrorClass 124
BTIsamLockRecord 124
BTIsamUnLockRecord 125
BTIsInitialized 125
BTIsNetFileBlock 126
BTKeyExists 127
BTKeyRecordSize 129
BTLockAllOpenFileBlocks 130
BTLockFileBlock 131
BTLockRec 132
BTMinimumDatKeys 132
BTNetSupported 133
BTNextDiffKey 134
BTNextKey 135
BTNextRecRef 136
BTNoCharConvert 136
BTNoNetCompiled 137
BTNrofKeys 137
BTOpenFileBlock 138
BTOtherWSChangedKey 140
BTPeekXxx routines 88
BTPrevDiffKey 141
BTPrevKey 142
BTPrevRecRef 143
BTPutRec 143
BTPutVariableRec 184
BTReadLockAllOpenFileBlocks 144

- BTReadLockFileBlock 145
- BTReclsLocked 146
- BTree52 39
- BTReleaseVariableRecBuffer 185
- BTSearchKey 146
- BTSearchKeyAndRef 147
- BTSetCharConvert 148
- BTSetDosRetry 149
- BTSetSearchForSequential 150
- BTSetVariableRecBuffer 185
- BTUnlockAllOpenFileBlocks 151
- BTUnlockAllRecls 151
- BTUnlockFileBlock 152
- BTUnlockRec 152
- BTUsedKeys 153
- BTUsedRecls 154
- BuildARow 282
- BuildBrowseScreenRow (OPBROW) 291
- BuildBrowseScreenRow (TVBROWS) 308
- BuildBrowseScreenRow (WBROWSER) 323
- BuildRow (OPBROW) 291
- BuildRow (TVBROWS) 308
- BuildRow (WBROWSER) 323
- ByteToKey 205

C

- CalcMaxWidth 324
- CallNameType 442
- CanCallLowBrowser 324
- CancelRedirection 464
- CBInterior 306
- CBrowserView 306
- ChangeBounds 308
- ChangeDatSNoChange 190
- CharHandler 292
- CloseDBaseFiles 244
- CloselsamFiles 244
- CompleteDBaseList 245
- CompletsamList 246
- ConnectLowBrowser 325
- CreateListHeaderOpenFileBlock 247
- CreateListHeaderUseDBaseFiles 248
- CreatePageSize 84
- CStyleDescendingKey 211
- CStyleNumKey 211
- CurRow 286

D

- DataBuffer 286
- DatExtension 81
- DBaseErrorMessage 249
- DBaseErrorNr 240
- DBaseExport 250
- DBaseFileName 240
- DBaseImport 253
- DBaseUsedErrorMessages 240
- DBaseVersion 240

- DBXxx constants 238
- DCXxx constants 238
- DebugEMSHeap 158
- DecideCase 240
- DefaultAdapterNum 441
- DefaultMergeName 231
- DefOPBrOptions 290
- DelayTimeOnGetRec 286
- DeleteTheFont 325
- DescendingKey 204
- DeviceType 463
- DEXxx constants 238
- DiaExtension 81
- DialogError 285
- DisableBrowseMouse 282
- DisableFiltering 283
- DisplayARow 283
- DoManualInitEMSHeap 167
- Done (OPBROW) 292
- Done (TVBROWS) 309
- Done (WBROWSER) 325
- DoneMergeSort 231
- DosLockRec 465
- DosMajor 463
- DosMinor 463
- DoSort 225
- Draw 309

E

- ElementCompareFunc 230
- ElementIOProc 230
- EMSDisturbance 35
- EMSHardErrorFuncPtr 167
- EMSHeapErrorFuncPtr 167
- EMSHeapInitialized 167
- EMSMaxAvail 168
- EMSMemAvail 168
- EMSPointer 167
- EnableBrowseMouse 283
- EnableFilter 326
- EnableFiltering 284
- EnumFct_DecideWrite 240
- ErrorReaction 240
- ERXxx constants 239
- ExitEMSHeap 169
- ExtendHandles 213
- ExtToKey 205

F

- FilterIsOn 326
- FirstUserInit 326
- FixToVarFileBlock 201
- FontInfo 320
- FreeEMSMem 169
- FreeListHeader 255
- FuncBuildKey 188
- FuncChangeDatS 188

542 Identifiers

G

GenKeyStr 286
GetBrowserTextRect 327
GetCurNrOfLines 327
GetCurrentDatRef (OPBROW) 292
GetCurrentDatRef (TVBROWS) 309
GetCurrentDatRef (WBROWSER) 328
GetCurrentKeyNr (OPBROW) 293
GetCurrentKeyNr (TVBROWS) 309
GetCurrentKeyNr (WBROWSER) 328
GetCurrentKeyStr (OPBROW) 293
GetCurrentKeyStr (TVBROWS) 310
GetCurrentKeyStr (WBROWSER) 329
GetCurrentRec (OPBROW) 293
GetCurrentRec (TVBROWS) 310
GetCurrentRec (WBROWSER) 329
GetElement (MSORT) 226
GetElement (MSORTP) 232
GetEMSMem 170
GetExtendedError 466
GetFooter 329
GetHeader 330
GetHeaderFooterColor 330
GetHighLightColor 330
GetLineNrFromY 331
GetLowHighKey 331
GetMachineName 469
GetNormalColor 331
GetPalette 310
GetPrinterSetup 469
GetRedirectionEntry 470
GetRowAreaRect 332
GetSortStatus 232
GetSuppressTimer 332
GetTempFileName 470
GetTextOutPosY 332
GetThisRec (OPBROW) 294
GetThisRec (TVBROWS) 311
GetThisRec (WBROWSER) 333
GRunLength 221

H

HandleChar 333
HandleEvent 311
HandlesToUseForAlloc 167
HardError 320
Heap6 39
HelpForBrowse 274
HighKey 287

I

IBMPCLanLoaded 471
Init (OPBROW) 294
Init (TVBROWS) 312, 318
Init (WBROWSER) 322, 333
InitAllUnits 38
InitCustom 295

- InitEMSHeap 171
- InitMergeSort 232
- IntFct_ReXUser 241
- IntFct_WriteTDef 241
- IntToKey 205
- InvertString 214
- IPXAddress 345
- IPXAllNodes 414
- IPXAllocateEventRec 419
- IPXAllocPacket 419
- IPXCancelEvent 420
- IPXCloseSocket 420
- IPXEventComplete 421
- IPXEventServiceRoutine 416
- IPXFreeEventRec 421
- IPXFreePacket 421
- IPXInternetAddress 422
- IPXListen 422
- IPXMaxDataSize 414
- IPXOpenSocket 423
- IPXOpenUniqueSocket 423
- IPXRelinquish 424
- IPXSend 425
- IPXServicesAvail 426
- IsamAssign 87
- IsamBlockRead 87
- IsamBlockReadRetLen 87
- IsamBlockWrite 87
- IsamClearOK 87
- IsamClose 87
- IsamCompiledNets 86
- IsamCopyFile 87
- IsamDelay 87
- IsamDelayBetwLocks 82
- IsamDelete 87
- IsamDOSError 86
- IsamDOSFunc 86
- IsamError 86
- IsamErrorMessage 215
- IsamExists 87
- IsamExtractFileNames 87
- IsamFBlockTimeOutFactor 82
- IsamFile 84
- IsamFileBlockName 84
- IsamFileBlockPtr 85
- IsamFileName 85
- IsamFileNameLen 82
- IsamFlush 87
- IsamFlushDOS33 82
- IsamForceExtension 87
- IsamGetBlock 87
- IsamIndDescr 85
- IsamInstallInt24Handler 43
- IsamKeyStr 85
- IsamLockTimeOut 82
- IsamLongSeek 87
- IsamLongSeekEOF 87
- IsamOK 86
- IsamPutBlock 87

- IsamRemoveInt24Handler 43
- IsamRename 87
- IsamReset 87
- IsamRewrite 88
- IsamReXUserProcPtr 86
- IsamVRecBufSize 178
- ISBrowser 289
- IsDriveLocal 471
- IsFileLocal 472
- IsFilteringEnabled 284
- IxExtension 81

K

- KeyNr 286
- KeyToBcd 207
- KeyToByte 207
- KeyToExt 207
- KeyToInt 207
- KeyToLong 207
- KeyToReal 207
- KeyToShort 207
- KeyToWord 207

L

- LastFileName 221
- LastVarRecLen 286
- LengthByteKeys 36
- ListHeader 240
- LocalStr 463
- LockBeforeRead 38
- LockError 285
- LongToKey 205
- LowKey 287
- LowWinBrowser (OPBROW) 290
- LowWinBrowser (TVBROWS) 306
- LowWinBrowser (WBROWSER) 321
- lwOptionsAreOn 296
- lwOptionsOff 296
- lwOptionsOn 296
- lwSelectOnClick 289
- lwSuppressUpdate 289

M

- ManualInitEMSHeap 158
- MapEMSPtr 171
- MaxCols (BROWSER) 274
- MaxCols (LOWBROWS) 285
- MaxEltsPerRow 285
- MaxEMSHeapPages 167
- MaxHeapToUse 221
- MaxHeight 82
- MaxKeyLen 83
- MaxNetworks 349
- MaxNrOfKeys 83
- MaxNrOfWorkStations 83
- MaxPageSize 84
- MaxRows 274

544 Identifiers

- MaxSelectors 229
- MaxVariableRecLength 178
- MedianThreshold 229
- MergeInfo 233
- MergeInfoRec 230
- MergeNameFunc 230
- MergeOrder (MSORT) 221
- MergeOrder (MSORTP) 230
- MergeSort 234
- MinEMSHeapPages 167
- MinimizeUseOfNormalHeap 83
- MinimumHeapToUse 235
- MinRecsPerRun 230
- MinRows 275
- MouseDnMark 269
- MouseUpMark 269
- MouseX1 269
- MouseX2 269
- MoveToHorizPos 334
- MoveToRelPos 334
- MsgExtension 81
- MsNet 34
- MSortIOResult 221
- MSortStatus 221

N

- NBEXxx constants 441
- NBNameMax 442
- NBNameStr 442
- NetBiosAddGroupName 445
- NetBiosAddName 445
- NetBiosAllocNCB 446
- NetBiosAllocPacket 446
- NetBiosAllocPost 447
- NetBiosCancelRequest 447
- NetBiosClearNCB 448
- NetBiosCmdCompleted 448
- NetBiosDeleteName 449
- NetBiosFreeNCB 449
- NetBiosFreePacket 450
- NetBiosFreePost 450
- NetBiosHangUp 450
- NetBiosInfo 451
- NetBiosInstalled 451
- NetBiosListen 452
- NetBiosListenNoWait 452
- NetBiosName 442
- NetBiosOpen 453
- NetBiosOpenNoWait 454
- NetBiosPostRoutine 442
- NetBiosReceive 455
- NetBiosReceiveBDG 456
- NetBiosReceiveBDGNoWait 456
- NetBiosReceiveDG 457
- NetBiosReceiveDGNoWait 457
- NetBiosReceiveNoWait 455
- NetBiosReenterError 442
- NetBiosRequest 458

- NetBiosResetAdapter 458
- NetBiosSend 459
- NetBiosSendBDG 460
- NetBiosSendBDGNoWait 460
- NetBiosSendDG 461
- NetBiosSendDGNoWait 461
- NetBiosSendNoWait 459
- NetSupportType 85
- NetworkStr 463
- NoError 285
- NoErrorCheckEMSHeap 158
- NoErrorHandler 255
- NoNet 34
- NoNetMode 275
- NoReXUser 255
- Novell 34
- NrOfRows 287
- nwbAddObjectToSet 360
- nwbChangeObjectSecurity 360
- nwbChangePassword 361
- nwbChangePropertySecurity 361
- nwbCloseBindery 362
- nwbCreateObject 362
- nwbCreateProperty 363
- nwbDeleteObject 363
- nwbDeleteObjectFromSet 364
- nwbDeleteProperty 364
- nwbGetBinderyAccessLevel 365
- nwbGetObjectID 365
- nwbGetObjectName 366
- nwbIsObjectInSet 366
- nwbOpenBindery 367
- nwbReadPropertyValue 367
- nwbRenameObject 368
- nwbScanObject 368
- nwbScanProperty 369
- nwbVerifyPassword 369
- nwbWritePropertyValue 370
- nwbXxx constants 358
- nwCancelCapture 396
- nwCloseSema 384
- nwDecSema 384
- nwDefaultServer 351
- nweaXxx constants 375
- nwEndCapture 396
- nwEnumQueues 396
- nwEnumServers 351
- nwExamineSema 384
- nwfaXxx constants 375
- nwFlushCapture 397
- nwfXxx constants 375
- nwGetBannerName 397
- nwGetBroadcastMessage 372
- nwGetBroadcastMode 372
- nwGetCaptureFlags 398
- nwGetConnInfo 351
- nwGetConnNo 352
- nwGetConnNoForUser 352
- nwGetFileAttr 376

- nwGetInternetAddress 353
- nwGetNetworkList 353
- nwGetNumPrinters 398
- nwGetServerInfo 354
- nwGetServerTime 354
- nwIncSema 385
- nwIntr 347
- nwIsCaptured 398
- nwIsLoggedIn 354
- nwJcXxx constants 392
- nwLockRecord 378
- nwOpenSema 385
- nwParseFileName 379
- nwpErrXxx constants 392
- nwpfXxx constants 392
- nwPrintASAP 392
- nwqAbortPrintJobFile 399
- nwqChangePrintJob 399
- nwqChangePrintJobPos 400
- nwqClosePrintJobFile 400
- nwqCreatePrintJobFile 401
- nwqEnumPrintJobs 402
- nwqGetPrintJob 402
- nwqRefreshPrintJob 403
- nwqRemovePrintJob 403
- nwSendBroadcastMessage 373
- nwSendMessageToConsole 373
- nwsErrXxx constants 383
- nwServerCall 347
- nwServerFromName 355
- nwServerVersion 355
- nwSetBannerName 404
- nwSetBroadcastMode 374
- nwSetCaptureFlags 404
- nwSetFileAttr 381
- nwSetServerTime 356
- nwShellType 347
- nwShellVersion 348
- nwStartCaptureToFile 405
- nwStartCaptureToQueue 405
- nwTTSAAbort 387
- nwTTSAvailable 387
- nwTTSTBegin 388
- nwTTSTDisable 388
- nwTTSTEnable 389
- nwTTSTEnd 389
- nwTTSTIsCommitted 390
- nwUNCtoNetWare 381
- nwUnlockRecord 381
- nwUpperStr 346
- nwXxx constants 345

O

- OpBrCommands 290
- OpBrKeySet 289
- OptimumHeapToUse 235

P

- Pack4BitKey 208
- Pack5BitKeyUC 208
- Pack6BitKey 208
- Pack6BitKeyUC 208
- PascalStyleNumKey 211
- PathName 221
- PCLanOpType 463
- PerformFilter (OPBROW) 297
- PerformFilter (TVBROWS) 313
- PerformFilter (WBROWSER) 334
- PhysicalNodeAddress 345
- PnbPacket 442
- PosClientCorruption 335
- PostCompletePage (OPBROW) 298
- PostCompletePage (TVBROWS) 314
- PostCompletePage (WBROWSER) 335
- PPacket 414
- PreCompletePage (OPBROW) 298
- PreCompletePage (TVBROWS) 314
- PreCompletePage (WBROWSER) 336
- PrnSetupStr 463
- ProcBTCharConvert 85
- ProcErrorHandler 239
- ProcessPostCommand 299
- ProcessPreCommand 299
- ProcessSelf 299
- ProgrammingError 320
- PutElement (MSORT) 226
- PutElement (MSORTP) 235

R

- ReadDataRecord 275
- RealToKey 205
- RebuildFileBlock 196
- RebuildVFileBlock 196
- RedirectDevice 472
- RefreshFunc 275
- RefreshPeriod 275
- ReIndexFileBlock 193
- ReorgFileBlock 198
- ReorgVFileBlock 198
- RestoreEMSCtxt 172
- RestructFileBlock 191
- RetriesOnGetRec 287
- RetriesOnLock 275
- RowRec (BROWSER) 276
- RowRec (LOWBROWS) 286
- RowsToJump 275
- RowString 286

S

- SaveEMSCtxt 172
- SavExtension 81
- ScrollBarAttr 269
- ScrollBarAutoSize 270
- ScrollBarCol 270

546 Identifiers

ScrollBarHt 270
 ScrollBarUp 270
 ScrollMark 270
 ScrollVertChar 270
 SearchForSequentialDefault 84
 SetAndUpdateBrowserScreen (OPBROW)
 300
 SetAndUpdateBrowserScreen (TVBROWS)
 315
 SetAndUpdateBrowserScreen
 (WBROWSER) 336
 SetCharValues 337
 SetDimAttr 300
 SetHeaderFooter (OPBROW) 301
 SetHeaderFooter (TVBROWS) 315
 SetHeaderFooter (WBROWSER) 337
 SetHeaderFooterAttr 301
 SetHighLightAttr 302
 SetKeyNr (OPBROW) 302
 SetKeyNr (TVBROWS) 316
 SetKeyNr (WBROWSER) 337
 SetLowHighKey (OPBROW) 302
 SetLowHighKey (TVBROWS) 316
 SetLowHighKey (WBROWSER) 338
 SetMargins 338
 SetPrinterSetup 473
 SetSuppressTimer 338
 SetTheFont 339
 SetUpdateInterval 303
 SetupWindow 339
 ShareInstalled 473
 shErrXxx constants 463
 ShortToKey 205
 ShowErrorOccured (OPBROW) 303
 ShowErrorOccured (TVBROWS) 316
 ShowErrorOccured (WBROWSER) 339
 ShowFilterWorking 340
 SliderAttr 270
 SpecialTask 284
 SPXAbortConn 426
 SPXAllocEventRec 427
 SPXCancelListenForConn 428
 SPXECBsListening 428
 SPXEstablishConn 429
 SPXEventComplete 430
 SPXFreeEventRec 430
 SPXGetConnStatus 431
 SPXListenForConn 431
 SPXMaxDataSize 414
 SPXMaxPoolCount 414
 SPXPacketReceived 432
 SPXReactivateECB 432
 SPXRetryCount 414
 SPXSend 433
 SPXServicesAvail 433
 SPXTerminateConn 434
 SPXWatchDog 414
 StartAutoRel 239
 StdDecideWrite 256

StdEMSInstCheck 159
 STemp 221
 SwapThreshold 230

T

TBIinterior 306
 TBrowserScrollBar 306
 TBrowserView 306
 TBrowserWindow (TVBROWS) 307
 TBrowserWindow (WBROWSER) 321
 TipxECB 415
 TipxEventRec 416
 TipxFragment 415
 TipxHeader 415
 TNCB 442
 TNetBiosStatus 443
 TnwBannerJob 393
 TnwBannerName 393
 TnwBroadcastMode 371
 TnwCaptureFlags 393
 TnwConnInfo 349
 TnwConnList 349
 TnwEnumPrintJobFunc 394
 TnwEnumQueueFunc 394
 TnwEnumServerFunc 349
 TnwErrorCode 346
 TnwFileHandle 375
 TnwFormName 393
 TnwNetworkList 350
 TnwObjectStr 346
 TnwPasswordStr 359
 TnwPrinter 394
 TnwPrintJob 394
 TnwPropStr 346
 TnwPropValue 359
 TnwRegisters 346
 TnwSemaName 383
 TnwServer 346
 TnwServerInfo 350
 TnwServerName 350
 TnwShellType 346
 TnwUpperStr 346
 TnwVolumeName 375
 ToLetFreePages 167
 TotalCharHeight 340
 TPostHandler 444
 TspxEvtRec 416
 TspxHeader 417
 TspxPoolECB 417
 TspxStatus 417

U

ucOPBrowse 289
 UnlockDosRec 474
 Unpack4BitKey 210
 Unpack5BitKeyUC 210
 Unpack6BitKey 210

- Unpack6BitKeyUC 210
- UpdateBrowserScreen (OPBROW) 304
- UpdateBrowserScreen (TVBROWS) 317
- UpdateBrowserScreen (WBROWSER) 341
- UpdateFile 474
- UsedFileBlock 287
- UseEMS 221
- UseEMSHeap 35
- UseFilerDLL 36
- UseOPCRT 38
- UseOPEMS 158
- UseReadLock 275
- UserMousePtr 270
- UseScrollBar 270
- UseSeparator 341
- UseTPCRT 38
- UseTPEMS 158
- UsingEMS 221

V

- VariableRecs 287
- VarRecMaxReadLen 287
- VersionStr 84
- vImCall 348
- vImVersion 348
- VoidFct_CharArrConvert 241
- VoidFct_ErrorHandler 241

W

- WordToKey 205
- WorkStatus 241
- WriteNoTypeDef 256
- WritePascalTypeDef 256
- WriteStringOut 341
- WSXxx constants 239

Index

0

3Com 462
8087 chip 205

A

Adding
 data 60
 memo field 201
ADDRESS 12
ADRESSEN 15, 17, 20
API 435
Artisoft LANtastic 2
ASCIIZ 36, 203

B

Background task hook 271
Bindery 344
 access level 365
 add object 360
 close 362
 create object 362
 create property 363
 delete object 363, 364
 delete property 364
 object ID 365
 object name 366
 object present 366
 open 367
 password 361, 369
 read property 367
 rename object 368
 scan 368, 369
 security 360, 361
 write property 370
BINDLIST 481
Branches 47
Broadcast
 get message 372
 mode 372, 374
 receive datagram 456
 send datagram 460
 send message 373
Browser 261, 263
 add command 277
 browse 278
 browse again 280
 build a row 282, 291, 308, 323
 change view 308
 change window size 291
 character width and height 337
 colors 300, 301, 302, 330, 331
 command processing 299
 connect 325

 data rectangle 332
 display records 309
Browser
 display row 283
 dispose 292, 309, 325
 error 303, 316, 339
 filter 297, 302, 313, 316, 326, 331, 334, 338, 340
 first user init 326
 font 325, 339
 footer 329
 get current key string 293, 310, 329
 get current number of lines 327
 get current record 293, 310, 329
 get current record number 292, 309, 328
 get index number 293, 309, 328
 get palette address 310
 get record 294, 311, 333
 handle character 292, 333
 handle event 311
 header 330
 header, footer 301, 315, 337, 341
 horizontal offset 323
 initialize 294, 295, 312, 318, 322, 333
 keystrokes 282
 mouse 282, 283
 mouse support 38
 network update 303
 Object Professional 288
 object-oriented 285
 ObjectWindows Library 319
 options 296
 page construction 298, 314, 335, 336
 pixel coordinate 332
 record filtering 283, 284
 row height 340
 row number 331
 scroll 334
 set current record 300, 315, 336
 set key number 302, 316, 337
 special task key 284
 text margins 338
 text rectangle 327
 timer 332, 338
 Turbo Vision 305
 update screen 304, 317, 335, 341
 valid 324
 width 324
 window 339
BTDEFINE.INC 34, 263
BTFILER.HLP 23
BTFILER.TPH 23
BTFWIN.HLP 23
Buffer, flush 107, 108
B-tree 47
B-Tree Filer's capacity 1
B-Tree Isam 1

C

- Cabling 344
- Capture
 - abort 396
 - banner name 404
 - banner page 397
 - close 396
 - flags 398, 404
 - flush 397
 - number of printers 398
 - start 405
 - test 398
- Compiler options 40
- CompuServe 31
- Conditional defines 34
- CONFIG.SYS 55, 213
- Configuring B-Tree Filer 33
- Context-sensitive help hook 271
- CONVERT.PAS 497
- Converting
 - from Database Toolbox 492
 - from earlier versions of Filer 499
 - to variable length records 200
- Corrupt
 - database 186
 - fileblock 79
- CRT, using 38
- Cut/paste 27

D

- Data orphan 53
- Data record
 - add 90
 - definition of 57
 - delete 98
 - find 106
 - free 109
 - get info 117
 - read 116, 118
 - read first four bytes 119
 - relative position 112
 - size 94
- Data reference 109, 115
- Database
 - B-Tree 48
 - data file 46
 - definition of 45
 - record definition 57
- Database Toolbox 492
- Datagram
 - definition of 435
 - receive 457
 - send 461
- DB2ISAM 257
- DBase conversion
 - add field node 242
 - close files 244

- error 249, 255
- DBase conversion
 - export 250
 - field definition list 245, 246, 255
 - import 253
 - open files 247, 248
 - progress routine 255
 - record filter 256
 - type definition file 256
- DBIMPEXP 236
- Dbox 492
- Deadlock 66, 71
- Degree 47
- Deleted record 57, 62, 181
- Demonstration programs 11
- Dialog file 55
- Dialog ID 114
- Directory structure 6
- Disk requirements 2
- DLL, using Filer in 36
- DOS 3.1 record locking 2
- DOS error code variable 86
- DOS function code variable 86
- Drive, local 471
- Dynamic sockets 406

E

- EMS
 - allocate 170
 - amount available 168
 - definition of 157
 - error codes 165
 - for heap storage 157
 - for sorting 216, 220
 - free 169
 - init 171
 - largest available block 168
 - map 171
 - memory usage 35
 - releasing 165
 - restore context 172
 - save context 172
- EMSHEAP 157
- Enz EDV-Beratung GmbH 1
- Error
 - class 124
 - codes 483
 - codes for EMS 165
 - extended information 466
 - get message string 215
 - handling 56
 - variables 86
- Event service routine 415
- Exclusive access 66
- Expanded memory 216
- Extended error codes 466
- Extensions 81

F

FASTUPD.nnn 3

File

- attributes 376, 381
- buffer flushing 82
- convert name 381
- extend handles 213
- extensions 81
- flush 474
- get unique name 470
- local 472
- lock 378, 465
- name 81, 84
- name length 82
- parse name 379
- server 344
- unlock 381, 474
- variable length format 177

Fileblock 55

- close 59, 91, 92
- create 58, 93
- data file name 94
- definition of 45
- delete 96
- delete record 61
- flush 106, 107
- index file name 119
- length 103
- lock 100, 102, 130, 131, 144, 145
- maintenance 177
- name 84
- network 126
- open 101, 138
- pointer 85
- reindex 193
- repairs 79
- restructure 190, 191
- unlock 151, 152

Filer

- early versions 499
- exit 99
- initialize 121

Filtering records 266

First record in the data file 46

FIXTOVAR 200

Flushing the file buffer 82

G

Group name 435, 445

Groupware 343

Guaranteed delivery of packets 407

H

Handles

- extend 213
- per fileblock 80

Heap

- management 121
- using EMS 157

Height of the tree 47, 82

Help 22

HIBROWS 285

Hidden swap files 23

Hotkey (POPHELP) 23

I

IBM PC LAN 2

Index

- description 85
- entries 48
- file 46, 49
- file size 93, 129
- keys 47, 49, 203

Indexed search 64

Initialization block 38

INSTALL.EXE 3

Installable keyboard hook 271

Installation 3

Integrity of the database 186

Internet address 353

Internetwork Datagram Packet Protocol 406

Internetwork Packet eXchange protocol 406

Interrupt 347

IPX 406

- allocate event rec 419
- available 426
- cancel event 420
- close socket 420
- control 424
- event status 421
- free event record 421
- internet address 422
- open socket 423
- packet buffer 419, 421
- receive 422
- send 425

ISAM2DB 259

ISAMTOOL 212

K

Key 46

- add 89
- C-Style 211
- changed 140
- clear 91
- delete 95, 97
- duplicate 110
- find 103, 104, 127
- get 111
- length 83, 115
- maximum number 83

- next 135
- Key
 - number of 137, 153
 - previous 142
 - record size 129
 - relative position 113
 - search 134, 141, 146, 147, 150
 - types 36
- Key string
 - invert 204, 214
 - length 85
 - pack 208
 - unpack 210
- Keyboard handling 271

L

- Leaf node 51
- Leaves 47
- LHA.EXE 3
- List box 261
- Local name table 435
- Locking 14, 38, 66, 67
 - circumvent 67
 - reacting to 74
 - retries 67
 - timeout 68
- Logical record counts 174
- LOWBROWS 285

M

- Macro Assembler 2
- MAKE 7
- MAKEHELP.EXE 23
- Mapping context 220
- Mapping EMS pointers 162
- MASM 2
- MaxHeight 49
- Maximum file name length 82
- Maximum length of a key 49
- MEDBROWS 285
- Memo field 173, 201
- Memory, out of 40
- Merge
 - file 231
 - order 220, 229
 - phase 220, 229
 - sort 216
- Microsoft MS-NET 2
- Minimum data record size 495
- Mouse
 - in browser 282, 283
 - support 269
- MS-NET 462
- MSORT 217
- MSORTP 227

N

- Nagel, Ralf 1
- NBSEND 343, 477
- NCB 436
 - allocate 446
 - free 449
 - initialize 448
- NetBIOS 435
 - cancel request 447
 - control block 436, 443
 - determine if installed 451
 - driver 451
 - issue request 458
 - name index 469
 - name table 445
 - reset adapter 458
- NETDEMO 11
- NETINFO 343, 476
- NetWare 462
 - API 344
 - shell type 347
 - shell version 348
- Network
 - current 133
 - emulation 80
 - interface 42
 - none 137
 - prerequisites 81
 - user connection number 352
- NISEND 343, 478
- No-wait event complete 448
- Node 47, 48, 406
 - merging 53
 - splitting 52
- Non-indexed search 64
- Novell 2, 34, 343
- NSSEND 343, 479
- Numeric key 89, 203
 - convert to string 205
- NUMKEYS 49, 203
- NWBASE 345
- NWBIND 357
- NWCONN 349
- NWFILE 375
- NWIPXSPX 406
- NWMSG 371
- NWPRINT 391
- NWSEMA 382
- NWTTS 386

O

- Object Professional 15, 263
- Object-oriented browser 263, 285
- ObjectWindows Library 20
- OPBROW 288
- OPCRT, using 38
- Opening a fileblock 59

OPISDEMO 15
Order 47
Overlaying B-Tree Filer units 40
Overlays and EMS 160
OWDEMO 20

P

Packed key string 203, 208
Packet buffer
 allocate 446
 free 450
Packets 406
PACKING.LST 3
Page
 balance 52
 buffer 56, 58
 size 49, 84
Password 472
Pasting help text 27
PC LAN 471
Permanent node name 435
POPHelp 22
Post-event handler 440
 allocate 447
 free 450
Primary key 11, 49, 59
Print capture
 abort 396
 banner name 404
 close 396
 flags 398, 404
 flush 397
 number of printers 398
 start 405
 test 398
Print job
 close 400
 create 401
 delete 399, 403
 information 402, 403
 number on queue 402
 queue position 400
 settings 399
Print queue 344, 396
PRINTDOC.EXE 3
Printer setup string 469, 473
Purchase agreement 30

R

Random access 46
Read locks 66
READ.ME 3
REBUILD 195
Record
 convert 136, 148
 filtering 266
 how many fit 132

 lock 90, 124, 132, 146
 modifying 62
Record
 next 136
 number 46
 number of 154
 previous 143
 unlock 125, 151, 152
 validation 267
 write 143
Redirection 462, 472
 cancel 464
 list entry 470
Refresh function 268
REINDEX 186
REORG 197
Requirements 2
RESTRUCT 186
Retry 149
Root node 51
Run
 buffer 219
 definition of 220, 229
Runtime fees 30

S

Save mode 11, 12, 17, 55, 79
Scan codes 26
Screen refresh 268
Scroll bar 269
Search 118
 example 64
 tree 48
Secondary key 49, 59
Section length 173
Semaphore
 close 384
 decrement 384
 increment 385
 open 385
 value 384
Sequenced Packet eXchange protocol 407
Sequential
 access pointer 63
 data access 46
 pointer 51
Server
 available 351
 call 347
 crashes 186
 date and time 354, 356
 handle 351, 355
 information 354
 login information 351
 network numbers 353
 version 355
 workstation logged in 354

- Session 435
 - listen for session open 452
 - open 453, 454
 - receive data 455
 - send data 459
 - terminate 450
- SHARE 34, 343, 462, 473
- SHELL.CFG 213
- Shift key codes 25
- Site licensing 30
- Socket 406
 - numbers 407
- Sort 223, 225, 234
 - abort 222, 231
 - done 231
 - fileblock 218
 - get element 226, 232
 - heap requirements 235
 - initialize 232
 - merge file 231
 - predict resource usage 225, 233
 - put element 226, 235
 - status 232
- Special task 274, 278, 284
- SPX 407
 - abort connection 426
 - alloc event record 427
 - avail 433
 - cancel listen 428
 - connection status 431
 - establish connection 429
 - event status 430
 - free event record, data buffers 430
 - listen buffer 428
 - receive 431
 - receive packet 432
 - send packet 433
 - terminate connection 434
 - test for packet received 432
- SPX2WAY 480
- Support
 - CompuServe 31
 - telephone 31
- System record 93

T

- TACCESS.DEF 494
- TACCESS.PAS 492
- TASM 2
- Telephone support 31
- Temporary index 266
- Terminology 45, 47
- Timeout lock 82
- Timer 332, 338
- Topologies 344
- TPALLOC 219
- TRAFFIC 68
- Transaction tracking 79

- Tree structure 47
- TSR swapping 22
- TTS
 - abort 120
 - abort transaction 387
 - available 387
 - begin transaction 388
 - disable 388
 - enable 389
 - end transaction 389
 - verify transaction 390
- TTSFILER 482
- Turbo Assembler 2
- Turbo Pascal 2
- Turbo Professional 263, 271, 272
- Turbo Vision 17
- TVBROWS 305
- TVISDEMO 17

U

- Underflow 53
- Unique name 435, 445
- UPGRADE.EXE 493

V

- Validation function 267
- Variable length records 173, 200, 278
 - add 179
 - buffer 179, 180, 185
 - convert to 201
 - delete 181
 - length 182
 - read 181, 182, 183
 - write 184
- Version string 84
- Viewing a database 261
- VLM
 - call 348
 - version 348
- VREBUILD 195
- VREC 173
- VREORG 197

W

- Warranty 30
- WBROWSER 319
- Well-known sockets 406
- WordStar 271, 277
- Workstation
 - connection number 352
 - maximum number 83
 - number 42, 114

X

- Xerox 406, 407
- XMS memory 24

NOTES

NOTES

*B-Tree Filer*TM

Tools for Fast, Compact Multi-User Database Applications

Write fast and capable multi-user database applications using B-Tree Filer and Turbo Pascal for Windows or DOS. Single user applications run even faster using the same units. The database code can be linked directly into your EXE file or obtained from a redistributable Windows DLL.

The B-Tree Filer database engine uses only about 40K bytes of code. Benchmarks for common database structures show that it runs faster than Paradox Engine, Btrieve, and the Borland Database Toolbox.

B-Tree Filer provides complete control over file structure, indexes, and locking. Records can hold any data type, up to 64K bytes total, in fixed or variable length. Up to 100 indexes can be constructed however you like. Multiple users can be managed through write locks, read locks, record locks, read-only files, and exclusive access files. Supports Novell NetWare and any PC network that provides the MS-DOS 3.x locking function.

Comprehensive Tools for All of Your Needs

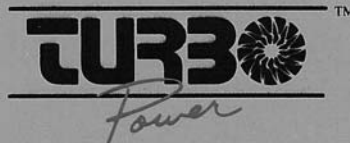
You get more than just the database engine. You also get features such as:

- bonus object-oriented interface to the database engine
- units for sorting, reindexing, and reorganizing databases
- dBase import/export capability
- visual database browsers that support ObjectWindows Library, Turbo Vision, Object Professional, and straight DOS applications
- a broad assortment of NetWare programming interfaces, including printing, IPX/SPX messaging, transaction tracking services, semaphores, and more
- NetBIOS and Share programming interfaces

Full Source Code, No Royalties, and Expert Support

You'll gain the confidence and flexibility of having complete source code. B-Tree Filer also includes comprehensive documentation, pop-up help, and plenty of demo programs. Free technical support is provided by telephone, e-mail, and fax. You pay no royalties.

B-Tree Filer requires Borland Pascal, Turbo Pascal for Windows, or Turbo Pascal. Also requires MS-DOS 3.3 or later and an IBM PC compatible computer. 3.5" disks are provided.



TurboPower Software
P.O. Box 49009
Colorado Springs, CO 80949-9009

©TurboPower Software, 1994