

TurboPower's compact, high-speed client/server database engine

# FlashFiler<sup>TM</sup> 2

The reliable, extensible, client/server database engine  
for Delphi and C++Builder

- *SQL support for easy data retrieval*
- *Internet enabled using TCP/IP*
- *Flexible open architecture allows for expandability and growth*
- *No per-seat licensing fee or hidden costs*



# FlashFiler 2<sup>TM</sup>

TurboPower Software Company  
Colorado Springs, CO

Order line (U.S. and Canada): 800/333.4160  
Elsewhere: 719/471.3898  
Fax: 719/471.9091

[www.turbopower.com](http://www.turbopower.com)

© 1997-2000 TurboPower Software Company. All rights reserved.

First Edition April 1997  
Second Edition March 2000  
Third Edition December 2000

# License 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.

Copyright © 1997-2000 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 FlashFiler. With the exception of FlashFiler Server (FFSERVER.EXE), FlashFiler Explorer (FFE.EXE), FlashFiler Service (FFSERVICE.EXE), FlashFiler Client Configuration Utility (FFCOMMS.EXE), FlashFiler Conversion Utility (FFCNVRTC.EXE and FFCNVRTC.EXE), you may not distribute any of the FlashFiler 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, units, or components that depend upon FlashFiler. However, others who receive your source code, units, or components need to purchase their own copies of FlashFiler in order to compile the source code or to write programs that use your units or components.

The supplied software may be used by one person on as many computer systems as that person uses. Group programming projects making use of this software must 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.

This software and accompanying documentation is deemed to be "commercial software" and "commercial computer software documentation," respectively, pursuant to DFAR Section 227.7202 and FAR 12.212, as applicable. Any use, modification, reproduction, release, performance, display or disclosure of the Software by the US Government or any of its agencies shall be governed solely by the terms of this agreement and shall be prohibited except to the extent expressly permitted by the terms of this agreement. TurboPower Software Company, 15 North Nevada, Colorado Springs, CO 80903-1708.

With respect to the physical media and documentation provided with FlashFiler, 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 media 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 media 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 FLASHFILER 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 and to any additional licensing terms contained in the DEPLOY.HLP file. If you do not agree, you should immediately return the entire FlashFiler package for a refund.

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

<b>Chapter 1: Introduction and Installation</b>	<b>1</b>
What's New in This Release?	4
System Requirements	6
Files Supplied	7
The FlashFiler Class Hierarchy	10
Installation	12
Organization of this Manual	13
Technical Support	16
Suggested Reading	17
<b>Chapter 2: Concepts and Glossary</b>	<b>19</b>
<b>Chapter 3: FlashFiler Server</b>	<b>31</b>
Configuration	32
Operation	33
FlashFiler Service	44
Transaction Journal Recovery	45
Server Scripting	47
Installing New Encryption Routines	52
Deploying FlashFiler Server	54
<b>Chapter 4: FlashFiler Explorer</b>	<b>55</b>
Operating FlashFiler Explorer	56
The Server Context Menu	57
The Database Context Menu	59
The Table Context Menu	67
The Table Browser	72
The Main Menu	75
<b>Chapter 5: Base Classes</b>	<b>77</b>
TffComponent Class	78
TffLoggableComponent Class	80
TffStateComponent Class	82
TffBaseLog Class	88
TffEventLog Component	91
TffObject Class	94
TffStringGrid Component	96

<b>Chapter 6: Client Architecture</b>	<b>103</b>
Setting Up a Data Module to Connect to a FlashFiler Server	104
Creating FlashFiler Tables That Use Calculated and Lookup Fields	107
Using FlashFiler Tables to Create a Master/Detail Relationship	110
Setting Up Data Entry Forms	111
Using Filters	112
Ranges	113
Finding Records	114
BLOB Streams	115
 <b>Chapter 7: FlashFiler Client</b>	 <b>117</b>
Global Variables and Constants	121
Component Helpers	124
TffDBListItem Class	131
TffBaseClient Class	134
TffClient Component	141
TffCommsEngine Component	142
TffSession Component	146
TffBaseDatabase Class	166
TffDatabase Component	177
TffDataSet Class	191
TffBaseTable Class	232
TffTable Component	247
TffQuery Component	251
TffBlobStream Class	262
 <b>Chapter 8: Advanced Architecture</b>	 <b>269</b>
Following the Flow	270
Modifying the FlashFiler Server	275
Creating Your Own Server	277
Finding New Uses For Transports	279
Creating Plugins	300
Extending the Server Engine	310

<b>Chapter 9: Transports and Threads</b>	<b>315</b>
TffBaseTransport Class	321
TffThreadedTransport Class	345
TffLegacyTransport Class	351
TffRequest Class	359
TffThreadPool Class	372
TffThread Class	376
TffPooledThread Class	378
TffTimerThread Class	380
<b>Chapter 10 : Command Handlers</b>	<b>383</b>
TffBaseCommandHandler Class	387
TffIntermediateCommandHandler Class	393
TffServerCommandHandler Class	395
<b>Chapter 11: Server Engines</b>	<b>399</b>
TffBaseServerEngine Class	401
TffIntermediateServerEngine Class	405
TffServerEngine Component	407
TffRemoteServerEngine Component	410
TffBaseSQLEngine Class	413
TffSQLEngine Component	421
TffBaseEngineManager Data Module	428
TffEngineManager Data Module	433
<b>Chapter 12: Plugins and Extenders</b>	<b>439</b>
TffBasePluginCommandHandler Class	447
TffBasePluginEngine Class	450
TffBaseEngineExtender Class	453
TffBaseEngineMonitor Class	456
TffSecurityMonitor Component	460
<b>Chapter 13: SQL Reference</b>	<b>463</b>
Syntax Conventions	464
Naming Conventions	465
Data Types	466
Supported Statements	467
Supported Functions	468
Key Words	471
Syntax Definitions	472

<b>Chapter 14: Troubleshooting and Performance</b>	<b>485</b>
Troubleshooting	486
Performance	493
<b>Chapter 15: Data Conversion</b>	<b>499</b>
Conversion Utilities	500
TffDataConverter Class	508
FlashFiler 1 Dynamic Link Library	516
Conversion Memory Manager	520
<b>Chapter 16: Appendices</b>	<b>523</b>
FlashFiler Data Types	524
Converting Data Types	531
Importing Data into FlashFiler Tables	534
User-Defined Indexes	543
Client Configuration	547
FlashFiler-Specific Routines (FFCLIntf)	553
Batched Record Routines	567
SingleEXE Applications	570
TffDataDictionary Class	572
Crystal Reports Support	601
Example Programs	603
Modifying the FlashFiler Source Code	606
How to Recompile FlashFiler	607
<b>Identifier Index</b>	<b>i</b>
<b>Subject Index</b>	<b>ix</b>

---

# Chapter 1: Introduction and Installation

FlashFiler is a professional, compact, client/server database engine for programmers developing applications with Borland Delphi or C++ Builder. Because of its open architecture and extendibility, FlashFiler easily supports large client/server systems and grows with you as your requirements expand. Because of its small size and speed, the FlashFiler Server engine can also be embedded in your stand-alone applications.

FlashFiler is also an excellent alternative to the Borland Database Engine (BDE). Since FlashFiler is small and doesn't require any external libraries, it is much easier to deploy than other database systems bogged down with run-time files.

FlashFiler is tightly integrated with Borland Delphi and Borland C++Builder, so you can quickly use FlashFiler right out of the box. FlashFiler data access components are TDataSet descendants, allowing third-party and VCL components such as TDataSource or TDBGrid to seamlessly connect with FlashFiler tables.

TurboPower designed and coded FlashFiler from the ground up in highly optimized code, specifically to provide high-speed database functionality in Delphi and C++Builder programs. Its two parts, FlashFiler Server and FlashFiler Client, are both written entirely in Delphi. Unlike other database engines, there are no DLLs, VBXs, or OCXs to worry about, ship, or install. You can compile FlashFiler into complete, self-contained applications. FlashFiler's components bind with your program to prevent distribution hassles. Since FlashFiler is a TurboPower library, you get the source code so you don't have to concern yourself with out-of-date DCU or OBJ files.

FlashFiler also includes FlashFiler Explorer, an application developed using FlashFiler Client. It provides database creation, importing, browsing, and maintenance capabilities. With FlashFiler Explorer, standard maintenance tasks such as changing the structure of a table can be done at the click of a mouse button. We provide the full source code for FlashFiler Explorer so you can use the same techniques in your own client applications.

FlashFiler gives you the security you need. FlashFiler Server can be configured so that all access to data requires the user to log on. For further security, especially across the Internet, data passed between client and server is encrypted. In addition, FlashFiler can encrypt physical data on disk. With FlashFiler's open architecture and free source code, you can extend or modify the security features as necessary.

The following sections describe some of the functionality provided by FlashFiler.



## Client/server architecture

FlashFiler is a client/server database engine. It consists of two parts: client source files that you compile directly into your application, and a server application that performs all file operations for the client. The server engine can optionally be compiled into your application.

## Transaction management

One benefit of the client/server architecture is the use of transactions. FlashFiler Server requires that all data additions, modifications, and deletions be enveloped within a transaction. This ensures all changes to your database are committed, or rolled back in the event of a failure. FlashFiler goes even further by providing transaction journal recovery. If a transaction is made in fail-safe mode and the server machine crashes or is powered down during the commit phase of the transaction, the journal file is recognized the next time the server is brought up and the administrator is prompted to commit or rollback the transaction. See “Chapter 2: Concepts and Glossary” on page 19 and “Chapter 7: FlashFiler Client” on page 117 for more information concerning transactions.

## Indexes

FlashFiler uses composite indexes comprised of keys that are generated from one or more fields in a record. FlashFiler provides classes for building and comparing keys. If you have special indexing needs (e.g., Soundex), you can integrate your own classes for building and comparing keys. See “Chapter 6: Client Architecture” on page 103 for more information concerning indexes and sorting.

## SQL

FlashFiler provides a TffQuery component that allows you to query tables using Structured Query Language (SQL). FlashFiler’s SQL engine supports a subset of the SQL-99 standard. See “Chapter 13: SQL Reference” on page 463 for more information concerning FlashFiler’s SQL support and the SQL engine.

## Large table support

Typically, database engines are limited to tables up to 4 gigabytes in size. Depending on your operating system, file system, and available disk space, with FlashFiler you can create tables that are hundreds of gigabytes or even terabytes in size. See “Chapter 2: Concepts and Glossary” on page 19 for more information about the operating system and file system boundaries.

## Data dictionary

FlashFiler provides a full-featured data dictionary to fully define a table. It defines the files that make up the table, the fields in the records, and the indexes built over the records.

## Data file repair facilities

Although FlashFiler has a reputation for being one of the most reliable database engines, we provide utilities to give you the maximum control over the integrity of your data. FlashFiler provides routines to pack, re-index, and restructure your tables. FlashFiler Explorer—for which code is provided—makes use of these routines, so you can easily see how to add them into your own applications.

## Data conversion utilities

FlashFiler 2 supplies routines to convert existing data from ASCII files, typed binary files, B-Tree Filer tables, FlashFiler 1 tables, or BDE tables to the FlashFiler 2 format. FlashFiler comes with several example programs that perform data conversion, including FlashFiler Explorer. For more information on converting FlashFiler version 1.X tables to FlashFiler 2 format, see “Chapter 15: Data Conversion” on page 499.

## TCP/IP protocol

FlashFiler can use the TCP/IP protocol to communicate between a client application and the server application. This means, for example, that FlashFiler can be used across the Internet.

## Scalability

Out of the box, FlashFiler supports hundreds of client connections using three different protocols. You now have the ability to create your own communication protocols (i.e., transports). Because FlashFiler Server is a component-based engine, you can add new functionality to the Server and react to events within the server engine. Virtually every aspect of the Server is under your control.

---

# What's New in This Release?

People currently using FlashFiler requested many new enhancements and features. Following are some of the highlights. See the README.HLP file for a more comprehensive list.

## SQL

FlashFiler now includes a TffQuery component and SQL engine. With them, a client application can use complex queries and generate reports.

## Expanded manual

We have more than doubled the size of the manual. The original parts of the manual have been refined and expanded. In addition, we have included new material on creating a client application, using the new FlashFiler Server components, and SQL capabilities.

## Large table sizes

Many people told us 4 gigabytes wasn't enough to hold all their data. FlashFiler 2 breaks that barrier, supporting tables many gigabytes in size. The size of FlashFiler 2 tables is limited only by your operating system, file system, and available disk space. See "Chapter 2: Concepts and Glossary" on page 19 for more information about the size boundaries.

## Component-based, multi-threaded server

We removed the Orpheus components from the FlashFiler Server so you can recompile it without having to purchase Orpheus. The Server is built using new transport, command handler, and server engine components, to name a few. You can mix and match components as your needs dictate, to the point of having more than one server engine in a FlashFiler Server. Also, the server engine is multithreaded allowing it to process requests from multiple clients at the same time.

## Improved Explorer

We removed the Orpheus components from FlashFiler Explorer so that you can recompile it without having to purchase Orpheus. Explorer now allows you to browse multiple tables and SQL queries at the same time. You can use the built in filtering and search functionality to restrict the records shown in the Explorer's data browsers.

## Faster singleEXE applications

Using FlashFiler's component-based architecture, you can embed one or more server engines within your stand-alone applications. While FlashFiler 1 depended upon a communications layer to talk with an embedded server engine, FlashFiler 2 makes calls directly into the server engine for extremely fast performance.

## Improved handling of time-outs

Each client-side component now has a Timeout property and the Server calculates its time-outs based on the client's setting.

## Improved BLOB storage

FlashFiler now stores BLOBs in 64-byte segments, resulting in very little wasted space. Using a new indexing scheme, positioning to a certain byte within a BLOB is very fast. In FlashFiler 1, BLOBs are limited to a maximum size of 16MB. FlashFiler 2 supports BLOBs up to 2GB in size.

## New file formats

The support for large tables and improved BLOB storage required us to change the format of FlashFiler tables. As a result, any existing tables you wish to use with FlashFiler 2 must be converted to the new file format. FlashFiler comes with two utilities for converting your tables. See "Chapter 15: Data Conversion" on page 499 for more information about the data conversion utilities. If you need to upgrade existing FlashFiler 1 applications to FlashFiler 2, see the README.HLP for an upgrade checklist.

---

# System Requirements

1. A computer capable of running Microsoft Windows 9x, NT, or 2000. A minimum of 32MB of RAM is required.
2. Delphi version 3 or later, or C++Builder version 3 or later. FlashFiler no longer supports Borland Delphi 1, Delphi 2, or C++ Builder 1.
3. A hard disk with at least 25MB of free space. To install all FlashFiler files and compile the example programs requires at least 50MB.
4. To test multi-user capabilities, a local area network, Winsock 1, or Winsock 2 must be installed on your computer. In addition, TCP/IP and/or IPX/SPX drivers must be installed on your computer.

For information regarding deployment requirements, see “Chapter 3: FlashFiler Server” on page 31.

# Files Supplied

FlashFiler includes packages and programs in source, precompiled, and executable form. It also includes demonstrations, example programs, and a help system. Source files are listed in the next section.

FlashFiler includes the following files in its installation. Note that not all the files are installed. The actual files installed are based upon your choices during installation. If there is a path prefixed to the file name, it is found under the main FlashFiler installation directory (C:\TurboPower\FF by default).

## README.HLP

A standard help file that describes the changes to the documentation and new features added since this manual was printed. Please read this file before using FlashFiler.

## \HELP\CBUILDER\FFBCB.HLP

The FlashFiler C++Builder help file.

## \BIN

Executables and DLLs for applications, utilities, and dynamic-link libraries. Table 1.1 lists the executable files and DLLs supplied.

**Table 1.1:** *The Executables and DLLs supplied with FlashFiler.*

File	Description
BDE2FF.exe	The FlashFiler BDE conversion utility
BETA.exe	The FlashFiler BDE export utility
FFCnvrt.exe	The FlashFiler GUI conversion utility
FFComms.exe	The FlashFiler client communications utility
FFConvrct.exe	The FlashFiler command line conversion utility
FFE.exe	The FlashFiler Explorer executable
FFServer.exe	The FlashFiler Server executable
FFSrvce.exe	The FlashFiler Service executable

## \EXAMPLES

FlashFiler examples for C++Builder and Delphi.

## \EXPLORER

FlashFiler Explorer source and help files. For details about FlashFiler Explorer, see “Chapter 4: FlashFiler Explorer” on page 55.

## \BDE2FF

Source and help files for an example conversion program that converts BDE tables to FlashFiler tables. The conversion program even supports BLOB fields.

## \BETA

Source and help files for a simple conversion program to import data to FlashFiler tables.

## \CONVERT

Source files for a program that converts FlashFiler 1 tables to FlashFiler 2 tables. See “Chapter 15: Data Conversion” on page 499 for more details.

## Units supplied

Table 1.2 lists the units or headers used by applications to access the FlashFiler 2 VCL components.

**Table 1.2:** *The FlashFiler components declared in each unit.*

Unit	Classes and Components
FFClREng	TffRemoteServerEngine
FFDB	TffBaseClient, TffBaseDatabase, TffBaseTable, TffBLOBStream, TffClient, TffCommsEngine, TffDatabase, TffDataset, TffSession, TffTable, TffTableProxy
FFDbBase	TffDBListItem
FFLLBase	TffComponent, TffObject
FFLLComm	TffBaseCommandHandler, TffBaseEngineManager, TffBasePluginCommandHandler, TffBasePluginEngine, TffBaseTransport, TffThreadedTransport
FFLLCore	TffBaseEngineExtender, TffBaseEngineMonitor, TffBaseServerEngine
FFLLGcy	TffLegacyTransport
FFLLGrid	TffStringGrid
FFLLReq	TffRequest
FFLLThrd	TffPooledThread, TffThreadPool, TffTimerThread
FFSrCmd	TffServerCommandHandler

**Table 1.2:** *The FlashFiler components declared in each unit. (continued)*

Unit	Classes and Components
FFSrEng	TffServerEngine
FFSrIntm	TffIntermediateCommandHandler, TffIntermediateServerEngine
FFSrSec	TffSecurityExtender, TffSecurityMonitor
uFFEgMgr	TffEngineManager



# The FlashFiler Class Hierarchy

---

TComponent (VCL)

    TffComponent (FFLLBase)

        TffDBListItem (FFDbBase)

            TffBaseClient (FFDB)

                TffClient (FFDB)

                TffCommsEngine (FFDB)

            TffSession (FFDB)

        TffBaseDatabase (FFDB)

            TffDatabase (FFDB)

    TffTableProxy (FFDB)

    TffThreadPool (FFLLThrd)

    TffLoggableComponent (FFLLCOMP)

        TFFStateComponent (FFLLCOMP)

            TffBaseCommandHandler (FFLLComm)

                TffIntermediateCommandHandler (FFSrIntm)

                    TffServerCommandHandler (FFSrCmd)

        TffBaseEngineMonitor (FFLLCore)

            TffSecurityMonitor (FFSrSec)

        TffBasePluginCommandHandler (FFLLComm)

        TffBasePluginEngine (FFLLComm)

        TffBaseServerEngine (FFLLCore)

            TffIntermediateServerEngine (FFSrIntm)

                TffServerEngine (FFSrEng)

                TffRemoteServerEngine (FFClREng)

        TffBaseTransport (FFLLComm)

            TffThreadedTransport (FFLLComm)

                TffLegacyTransport (FFLLLgcy)

TDataModule (VCL)

    TffBaseEngineManager (FFLLComm)

        TffEngineManager (uFFEgMgr)

TDataset (VCL)

    TffDataset (FFDB)

        TffBaseTable (FFDB)

            TffTable (FFDB)

TObject (VCL)

    TffObject (FFLLBase)

        TffBaseEngineExtender (FFLLCore)

            TffSecurityExtender (FFSrSec)

            TffRequest (FFLLReq)

TThread (VCL)

    TffPooledThread (FFLLThrd)

    TffTimerThread (FFLLThrd)

TStream (VCL)

    TffBLOBStream (FFDB)

# Installation

Install FlashFiler directly from the TurboPower Product Suite CD. Simply insert the CD into your CD-ROM drive, select FlashFiler 2 from the list of products, click “Install”, and follow the instructions. If the TurboPower introductory splash screen does not appear automatically upon insertion of the CD, run `X:\CDROM.EXE` where *X* is the letter of your CD-ROM drive.

## IDE integration

The installation program should integrate FlashFiler properly within each of the IDEs you selected during installation. In some cases, you may have to install FlashFiler manually. The following steps assume FlashFiler is installed on your hard disk:

1. Ensure the run-time FlashFiler packages are in your Delphi BIN directory or the Windows System32 directory (system for Windows 95 and 98). The run-time packages are those packages whose name has the format `Fvw_Rdd.BPL`.
2. Start your IDE.
3. Add the FlashFiler design-time package to the component package list by going to Component | Install Packages on the menu. Click the Add button. When prompted to select a package, select the filename containing `_D` and your version of FlashFiler based on Table 1.3.

**Table 1.3:** *The packages required for IDE integration.*

IDE	Packages required (vvv is your version of FlashFiler)
Delphi 3	Fvvv_R30.BPL, FvvvJR30.BPL, Fvvv_D30.BPL, FvvvJD30.BPL
Delphi 4	Fvvv_R40.BPL, FvvvJR40.BPL, Fvvv_D40.BPL, FvvvJD40.BPL
Delphi 5	Fvvv_R50.BPL, FvvvJR50.BPL, Fvvv_D50.BPL, FvvvJD50.BPL
Delphi 6	Fvvv_R60.BPL, FvvvJR60.BPL, Fvvv_D60.BPL, FvvvJD60.BPL
C++ Builder 3	Fvvv_R35.BPL, FvvvJR35.BPL, Fvvv_D35.BPL, FvvvJD35.BPL
C++ Builder 4	Fvvv_R41.BPL, FvvvJR41.BPL, Fvvv_D41.BPL, FvvvJD41.BPL
C++ Builder 5	Fvvv_R51.BPL, FvvvJR51.BPL, Fvvv_D51.BPL, FvvvJD51.BPL

---

# Organization of this Manual

This manual is organized as follows:

- Chapter 1 introduces FlashFiler and discusses its installation.
- Chapter 2 provides the basic concepts and a glossary of terms.
- Chapter 3 describes FlashFiler Server.
- Chapter 4 describes FlashFiler Explorer.
- Chapter 5 describes the base classes used to construct the FlashFiler components.
- Chapter 6 discusses the FlashFiler Client architecture and provides a short tutorial on setting up a client application.
- Chapter 7 describes the FlashFiler Client classes.
- Chapter 8 discusses the FlashFiler Server architecture and describes how to customize the server.
- Chapter 9 describes the transport and thread classes.
- Chapter 10 describes the command handler classes.
- Chapter 11 describes the server engine classes.
- Chapter 12 describes the plug-in and engine extender classes.
- Chapter 13 describes FlashFiler's SQL implementation and SQL engine class.
- Chapter 14 provides troubleshooting and performance improvement information.
- Chapter 15 tells you how to convert your FlashFiler 1 tables to FlashFiler 2 tables.
- Chapter 16 is an Appendix that contains miscellaneous information about data types, user-defined indexes, source code modification, and other topics not fitting into the previous chapters.
- An identifier index and a conventional subject index are provided.

Class reference chapters start with an overview of the classes and components discussed in that chapter. The overview also includes a hierarchy for those classes and components. Each class and component is then documented individually, in the following format:

## Overview

Describes the class or unit.

## Hierarchy

Shows the ancestors of the class being described, generally stopping at a VCL class. The hierarchy also lists the unit in which each class is declared and the number of the first page of the documentation of each ancestor. Some classes in the hierarchy are identified with a number in a bullet: ❶. This indicates that some of the properties, methods, or events listed for the class being described are inherited from this ancestor and documented in the ancestor class.

## Properties

Lists all the properties in the class. Some properties may be identified with a number in a bullet: ❶. These properties are documented in the ancestor class from which they are inherited.

## Methods

Lists all the methods in the class. Some methods may be identified with a number in a bullet: ❶. These methods are documented in the ancestor class from which they are inherited.

## Events

Lists all the events in the unit. Some events may be identified with a number in a bullet: ❶. These events are documented in the ancestor class from which they are inherited.

## Reference Section

Details the properties, methods, and events of the class or component. These descriptions are in alphabetical order. They have the following format:

- Declaration of the property, method, or event.
- Default value for properties, if appropriate.
- A short, one-sentence purpose. A ↗ symbol is used to mark the purpose to make it easy to skim through these descriptions.
- Description of the property, method, or event. Parameters are also described here.
- Examples are provided in many cases.
- The “See also” section lists other properties, methods, or events that are pertinent to this item.

Throughout the manual, the ⚠ symbol is used to mark a warning or caution. Please pay special attention to these items.

## Naming conventions

To avoid class name conflicts with VCL components and classes, or those from other third party suppliers, all FlashFiler class names begin with 'Tff'. The 'ff' stands for FlashFiler.

## On-line help

Although this manual provides a complete discussion of each component, keep in mind there is an alternative source of information available. Once properly installed, help is available from within IDE. Pressing <F1> with the caret on a FlashFiler property or component or from the Delphi or C++ Builder Object Inspector display help for the item.

---

## Technical Support

The best way to get an answer to your technical support question is to post it in the FlashFiler newsgroup on our news server ([news.turbopower.com](http://news.turbopower.com)). Many of our customers find the newsgroups a valuable resource where they can learn from others' experiences and share ideas in addition to getting quick answers to questions. This is especially true of the FlashFiler newsgroups. Many people using FlashFiler have taken interest in extending and customizing the product. The result is that FlashFiler has many experts willing to help all.

To get the most from the newsgroups, we recommend you use dedicated newsreader software.

Newsgroups are public, so please do not post your product serial number, product unlocking code, or any other private numbers (such as credit card numbers) in your messages.

TurboPower's KnowledgeBase is another excellent support option. It has hundreds of articles about TurboPower products accessible through an easy-to-use search engine ([www.turbopower.com/search](http://www.turbopower.com/search)). The KnowledgeBase is open 24 hours a day, 7 days a week so you'll have another way to find answers to your questions even when we're not available.

Other support options are described in the support brochure included with FlashFiler. You can also read about support options at [www.turbopower.com/support](http://www.turbopower.com/support).

## Acknowledgements

The capabilities in this version of FlashFiler were made possible in large part by the efforts of the FlashFiler Advisory Board. Not only are they energetic in their opinions of what FlashFiler should do, they also have many ideas about how to integrate those features. They have been tireless in their efforts to make sure that FlashFiler is the best it can be.

---

## Suggested Reading

Although FlashFiler is designed to be easy to use for the novice and experienced programmer alike, some background reading might come in handy. These references were certainly handy during the development of FlashFiler Server, Client, and Explorer.

For information on BDE routines, see Borland's official BDE help files. *Borland Database Engine for Windows User's Guide* (version 2.0) was invaluable, but it is unfortunately no longer available in hard copy.

For a good grounding in writing database applications with Delphi we recommend *Database Developer's Guide with Delphi 2* by Ken Henderson (Sams).

For details on relational database theory, including reasons why you shouldn't use arrays (or null fields) in records, we recommend *An Introduction to Database Systems*, 6th Edition, by C.J. Date (Addison-Wesley).

To get familiar with SQL we suggest *The Practical SQL Handbook*, 3rd Edition, by Bowman, Emerson, and Darnovsky (Addison-Wesley). Advanced SQL users may be interested in *The SQL Standard*, 4th Edition, by CJ Date and Hugh Darwen (Addison-Wesley).

For details on the low level B-Tree indexing algorithms and data structures, we recommend the following:

- *File Structures*, 2nd Edition, by Folk and Zoellick (Addison-Wesley)
- *Introduction to Algorithms* by Cormen, Leiserson, and Rivest (McGraw-Hill)

For descriptions of peer-to-peer network programming, we recommend the following:

- *Windows NT Network Programming* by Ralph Davis (Addison-Wesley)
- *Windows Network Programming* by Ralph Davis (Addison-Wesley)
- *Developing for the Internet with Winsock* by Dave Roberts (Coriolis Group Books)

You'll find an introduction to transaction processing in *Principles of Transaction Process*, by Bernstein and Newcomer (Morgan-Kaufman). Detailed information is in *Transaction Processing: Concepts and Techniques* by Jim Gray and Andreas Reuter (Morgan-Kaufman).





---

## Chapter 2: Concepts and Glossary

FlashFiler may be used as a client/server database engine or as an embedded database engine. To help you understand the concepts behind FlashFiler's architecture, this chapter discusses terms and terminology used by the product. This discussion takes the form of several glossary or dictionary entries. Although this chapter appears early on in this manual, it is not meant to be read from start to finish, but rather to be used as a dictionary when an unfamiliar term appears elsewhere in the FlashFiler documentation. To aid as a cross-reference tool, if an entry in this glossary has an emphasized word, it refers to another entry in the glossary. The entries are in alphabetic order for easy reference.

### Alias

An alias is a name for a **database**. It defines the directory where the database resides. When a client application opens a database, it refers to the database using an alias instead of a directory path. Aliases are easier to remember and maintain than the directory path. They have the additional advantage that you can change the directory to which an alias refers without any effect on the client applications.

More than one alias may refer to the same directory. FlashFiler recognizes, allows, and supports this situation. Both aliases share the same **transaction manager**.

### Binary Large Object (BLOB)

A Binary Large Object (BLOB) is a special type of data field. A BLOB is an object of variable size such as a text memo, image, or sound file. Since it is not of fixed size, the BLOB data itself does not appear in the record (all FlashFiler records are fixed in size). Instead, FlashFiler maintains an 8-byte reference number within the record. The reference number points to the actual BLOB data, which can be part of the table's data file, or be part of another file altogether.

FlashFiler provides a special type of stream, `TffBLOBStream`, to access the data in a BLOB. Use it just like you would use `TBLOBStream`. This is the recommended way to process a BLOB rather than using the underlying low-level routines. See "TffBlobStream Class" on page 262 for more details.

When defining a table containing BLOBs, FlashFiler provides several BLOB field types such as `fftBLOBMemo` and `fftBLOBGraphic`. The BLOB field types are for your use, to describe the kind of data you are placing in the BLOBs. FlashFiler does not modify or adjust the BLOBs based upon their field type.

BLOBs may be stored in the table's data file or in a separate file. You may also store each BLOB as an individual file using the `AddFileBLOB` method of the `TffTable` component. In the latter case, the BLOB information stored in the FlashFiler table contains the path and

name of the external BLOB file. The client application sees no difference between BLOBs stored in the table and external BLOBs. File BLOBs are useful when files on the server already contain the data you want to reference but do not want to duplicate in the database.

BLOBs stored in a FlashFiler file are organized into segments. The minimum size of a segment is 64 bytes. The maximum size is the size of the file **block** minus the length of the block header (28 bytes in the case of a BLOB block). In FlashFiler 1, small BLOBs resulted in wasted space. This new BLOB storage scheme wastes very little space as a result of using dynamically sized segments.

Each BLOB contains a lookup segment serving as a table of contents into the BLOB. If you are interested in a portion of the BLOB then FlashFiler is able to quickly seek to that point within the BLOB.

## Blocks

When you define the files that make up a table, you must define a block size for each file. At a low-level, the server reads and writes data from table files in same-sized blocks. There are four types of blocks:

- data blocks that hold one or more records
- **index blocks** that hold a complete B-tree page
- **BLOB blocks** that hold one or more BLOB segments
- data dictionary blocks that hold part of the data dictionary

The valid block sizes are 4KB, 8KB, 16KB, 32KB, and 64KB.

FlashFiler supports files of very large size, based upon your operating system and file system. However, you cannot reliably guess the number of records in a data file by dividing the file size by the record length. A small amount of overhead data is associated with each block. This overhead data defines the type of data in the block and provides for a chain of deleted blocks available for reuse. Furthermore, the file as a whole has overhead data stored in yet another block.

Data blocks consist of the overhead data, a contiguous set of records, and unused space at the end of the block (less than the physical size of a record). Records are not split across blocks. The block size is the ultimate limit on the physical size of a record. For a 4KB block, the maximum physical record length is slightly less than 4KB (4062 bytes to be exact). For a 32KB block, the limit is 32732 bytes.

Each record is prefixed with a 1-byte flag indicating whether the record is deleted. Each record is suffixed with one or more bytes identifying null fields within the record. To calculate the number of suffix bytes, you need to know the number of fields:

```
SuffixBytes := (FieldCount + 7) div 8;
```

To calculate the maximum number of records that can be stored in a block, you need to know the number of suffix bytes and the sum of the sizes of the fields.

```
MaxRecordCount := (BlockSize - 32) div
                  (SumFieldSizes + 1 + SuffixBytes);
```

If the sum of a record's field sizes is less than 9 bytes, FlashFiler uses 9 bytes to store the record. This is because FlashFiler requires 9 bytes for each deleted record. The first byte indicates the record is deleted. The remaining 8 bytes point to the next deleted record. See **Deleted Record Chain** on page 23 topic for more information.

Index blocks consist of the overhead data, the B-Tree page, and unused space. FlashFiler Server uses an improved B-tree algorithm that does not depend on a fixed number of keys per page. Instead, it uses a fixed number of bytes per page and tries to fit as many keys as it can in that page. The algorithm requires a minimum of 3 keys per page. This leads to a maximum key length for an index of 1024 bytes (the minimal 3 keys, plus their overhead, plus the fixed overhead of the block as a whole, will then fit into a 4KB block).

For **BLOB** blocks, the situation is somewhat more complex. Each BLOB has a header segment, at least one lookup segment and one or more content segments. Header segments have a fixed length and contain basic information about the BLOB. Lookup segments are dynamically sized and contain a table of contents pointing to the location of the content segments.

Content segments tend to be the maximum size allowed in a block. The last content segment is usually less than the size of the block. Since the minimum segment size is 64 bytes, there is very little wasted space. The segments comprising a BLOB may be spread across multiple file blocks.

The **data dictionary** blocks taken together (although, in general, there is only one) form a stream. This stream is the persistent storage for the table's data dictionary object. These data dictionary blocks are only found in the data file part of the table.

## Client

The client represents an individual client connection to the server. When the client first contacts the server, the server returns a client ID and also sets up a **session** for that client. The client is represented by the TffClient component. If you do not explicitly place a TffClient component on your form, FlashFiler automatically creates a TffClient behind the scenes.

The client influences the **transport** used to talk with the FlashFiler server. If the client is connected to a **remote server**, all commands going through the client use the protocol dictated by the remote server engine's transport. If the client is connected to an **embedded server**, all calls are made directly into the server engine and no protocol is required. If the

client is not connected to any type of server engine, the client creates a server engine and transport based upon the value Protocol within registry key  
HKEY\_LOCAL\_MACHINE\Software\TurboPower\FlashFiler\<FF version>\Client Configuration.

## Comms engine

A comms engine is a client component that provides access to a network protocol. This component, TffCommsEngine, is deprecated and provided only for backwards-compatibility. When creating new applications, use TffClient instead. See the **Client** topic for more information.

The comms engine initializes the underlying network driver in readiness for contacting a server. All packets between client and server go through a comms engine. Note that activating a comms engine does not cause a connection to be made to a server, that is the job of the session.

## Cursor

When you open a **table** or execute a **query**, behind the scenes the FlashFiler server creates and returns a cursor. A cursor is a pointer into the collection of records from the table or query. As you instruct the table or query to navigate through the records (e.g., go to first record, move to next record), the FlashFiler server uses the cursor to carry out the navigation and track your current position. The locking, adding, updating, and deleting of records is also carried out through the cursor.

Any number of cursors may be opened on the same FlashFiler table. Cursors are associated with a **session**.

The VCL's TTable and TQuery classes are an encapsulation of a cursor, as is FlashFiler's TffTable and TffQuery.

## Database

Logically, a database is a collection of tables. This is how the client application views a database. Physically, a database is a directory that contains the files comprising the tables within the database. The directory must be accessible by the server. A database is known by its alias, and sometimes by the path on the server that points to the set of tables.

In FlashFiler Client, a database is represented by a TffDatabase component.

## Data dictionary

The definition of a table is in its data dictionary. A data dictionary holds lists of three types of objects: file descriptors, field descriptors, and index descriptors.

A file descriptor is a definition of one of the files that makes up a table. It includes the extension for the file and the type of the file (data file, index file, or BLOB file). There is always at least one file descriptor in a data dictionary, the file descriptor for the data file, and it cannot be altered or deleted.

A field descriptor is the definition of a single field that forms part of the layout of the record. It stores information about the field, such as its name, the type of data, its length in bytes, and its offset in the record buffer.

An index descriptor defines a single index for the table. It stores information such as the name of the index, the file in which it is stored, how the key is to be generated, and how the key is to be compared.

## Deleted record chain

When you delete a record from a table, the record is marked as deleted and is added to the table's deleted record chain. A pointer to the head of the deleted record chain is stored in file **block zero**. Each record in the chain points to the next record in the chain.

When you insert or append a record, FlashFiler re-uses the space occupied by the first record in the deleted chain. If the deleted chain is empty, FlashFiler uses the next available spot in the last block of the file or, if the last block is filled, allocates a new file block.

If you delete a large number of records from the table and wish to remove the space occupied by the deleted records, pack the table using the FlashFiler Explorer or via code. Packing creates a new copy of the table minus the deleted records.

## Embedded server

An embedded server means you have placed a TffServerEngine component within your application. Using a **client**, you can have **session**, **database**, and **table** components communicate directly with the TffServerEngine. This is a high-performance solution for stand-alone applications because calls are made directly into the server engine instead of through a communications layer.

Using a **transport** and command handler, it is also possible to have an embedded server engine handle requests from other client applications. See "Chapter 11: Server Engines" on page 399 for more information.

## Index

An index is a logical ordering of records, with each record represented by a key. FlashFiler Server generates a key for a record for each index and automatically updates those indexes, maintaining them in key sequence, when a record is inserted, modified, or deleted.

FlashFiler Server can maintain up to 255 indexes on a single **table**. The indexes can be composite indexes or user-defined indexes. In a composite index, the key is formed by a concatenation of one or more fields of the record. When you define the index, you define which fields will form the key. For example, if you have a customer table, you might want to index the records by last name and then first name. Up to 16 fields can be used in a composite index. If you need more fields in an index, you can create a user-defined index.

A user-defined index is formed from keys that are generated by a routine you supply. You must also provide a routine to compare two keys. These two routines are compiled into a DLL, which you declare to FlashFiler Server.

For example, you might want to index based on the Soundex value of a name. You could do that by storing the Soundex value in the record and then creating a composite key index on the Soundex field. However, this requires more storage for the extra field. But more importantly, it is difficult to ensure that the index field is updated whenever the name changes. The client application must do the updating. A better solution is to use a user-defined index. You can create a DLL function that reads the name field and computes a Soundex key value, which is used in the index. The extra storage for the Soundex value is no longer required, and the index field is automatically updated whenever the dependent fields are modified.

See “User-Defined Indexes” on page 543 for more information about user-defined indexes.

Every FlashFiler table has at least one index, the Sequential Access Index. This is a pseudo-index that allows you to retrieve records in their physical order within the table.

## Multi-threaded client

A multi-threaded client application contains a primary thread plus one or more secondary threads. FlashFiler Client supports multi-threaded applications. Most of the client-side components are not thread safe. Therefore, each thread must have its own `TffClient`, `TffSession`, `TffDatabase`, and `TffTable` components. Because server engines and transports are thread-safe, a single server engine and transport may be shared between the threads. In other words, you may connect each `TffClient` to the same server engine.

## Multi-threaded server

A multi-threaded server is a server that allows multiple client requests to be processed at the same time. The FlashFiler Server has a multi-threaded architecture. In order to receive requests from multiple communications channels, it creates one thread per **transport**. The FlashFiler Server also contains a **thread pool**. When a request is received through a transport, the transport obtains a thread from the thread pool and tells the thread to process the request.

## Query

A query describes a set of records matching a certain set of criteria, ordered and/or grouped in a certain manner. The records are comprised of one or more fields from records within one or more **tables**. The query is composed using Structured Query Language (SQL).

The FlashFiler TffQuery component represents a query. When a query is executed, the server builds a temporary table, opens a **cursor** on the table, and returns the cursor to the **client**.

## Remote server

A remote server is a FlashFiler Server running outside the client application's process space. The remote server may be on the same workstation or on a different workstation accessible via a local area network, wide area network, or the Internet. The FlashFiler client application represents a connection to a remote server using the TffRemoteServerEngine component.

## Security

Security is implemented in multiple levels in FlashFiler Server. FlashFiler Server can encrypt data stored in tables before it is written to disk (it is decrypted when it is read) using a proprietary method based on a standard algorithm. You cannot look at an encrypted FlashFiler table with a hex file viewer and see the records or the keys in an index. The source code for this encryption is provided in the FFTBCRYPT unit. Rather than using the standard FlashFiler encryption, you can implement your own and link it into the server application. See "Installing New Encryption Routines" on page 52.

FlashFiler Server can be run in two modes: secure and non-secure. Regardless of mode, messages transmitted between the server and its client are encrypted. During the connection process, the FlashFiler Server returns a random code to the client. The random code is generated using GetTickCount and a simple formula (see GenerateCode in the FFLLPROT unit). Messages are then encrypted and decrypted using this code. This is intended to deter casual hackers when FlashFiler is used over a public wide area network, the Internet, or even internally via packet sniffers on a LAN.



Secure mode requires that all users log on to the server before using any of its functions. This log on occurs both at client applications and within the FlashFiler Server application itself (to alter the configuration of the server requires the user to log on as an administrator).

In secure mode, messages undergo additional encryption by XORing the generated code with the client's hashed password. Since the user's password is never passed between client and server, the client application must have the correct password in order to decrypt the code and finalize the connection.

## Session

A session is used to organize **databases** and **tables** within a client application. Using TffSession components, the **client** can open extra sessions if required. Open databases, tables, and cursors are not shared between sessions. Each session must open and use its own objects.

## Single User Protocol

Single User Protocol (SUP) is a protocol used by the FlashFiler **transport** component to communicate between a FlashFiler Client and FlashFiler Server located on the same workstation. SUP is implemented using Windows messaging therefore it is faster than using TCP/IP or IPX/SPX in the same situation.

## Table

A table is a set of records, each of which have the same field layout, a set of **indexes** built over those records, and possibly a collection of **BLOBs** for those records. This is how the client application views a table. The server, on the other hand, views a table as a physical object; a set of one or more files defining and holding a set of records. The files comprising a table are the data file, optional index files, and an optional BLOB file. A table can consist of just one file, the data file, in which case all of the records, indexes, and BLOBs are in that one file. Opening a table causes a **cursor** to be created and returned to the **client**.

## TDataSet Descendant Model

FlashFiler Client uses the TDataSet Descendant Model for data access. The TDataSet Descendant Model works by defining TDataSet descendant classes called TffTable and TffQuery. TffTable mimics the standard VCL's TTable class and TffQuery mimics TQuery. Both call FlashFiler-specific code instead of BDE code. This class is a component that appears on the IDE's palette, together with components for the FlashFiler **client** (TffClient), a FlashFiler **session** (TffSession), and a FlashFiler **database** (TffDatabase).

## Thread pool

A thread pool maintains a set of threads for the use of the **multi-threaded server**. The thread pool is implemented by the `TffThreadPool` component. You may pre-define a maximum number of threads for the pool as well as an initial number of threads to be initialized when the pool is first created. When a server **transport** receives a request, it asks the pool for a thread. If a thread is available, the pool hands it off to the transport. If a thread is not available and the thread pool has not reached its limit, the pool creates a new thread and hands it off to the transport. If no threads are available and the pool has reached its limit, the transport's request is queued for the next available thread.

## Transaction

A transaction is a group of updates to one or more **tables** in a **database** that must be applied as a whole, or not at all. The integrity of the database would be compromised if only part of the individual updates were made. FlashFiler Server allows any number of concurrent transactions per database but only one transaction per `TffDatabase` component.

Because unforeseen problems such as a power outage, physical disk error, or a disk full condition can occur in the middle of an update, FlashFiler Server was designed around the concept of transactions. Every time you make an update, you must start a transaction. When the update is complete, you commit the transaction so that the updated data is written to the disk. FlashFiler only writes to the table files when you commit the transaction (it does not write to the table files during your update). If an error occurs in the middle of a transaction, all you have to do is rollback the transaction and all the changes from the start of the transaction are discarded.

Since numerous updates to a database only occur as a single record update, FlashFiler has a special facility called implicit transactions. If you try to insert a new record, or modify or delete an existing record, and the server notices that there is no transaction active for the **cursor**, it starts a new transaction for you, applies the update and then, providing no error occurred, commits the transaction. If, on the other hand, an error did occur (for example, a key violation in an index), the transaction rolls back automatically. Our recommendation, however, is to use the explicit transaction calls.

Wrapping multiple record inserts, updates, and deletes within an explicit transaction provides faster performance than relying upon the FlashFiler Server to use implicit transactions.

There are two categories of transactions: normal and fail-safe. The client application defines which is used. If the client uses fail-safe transactions, a transaction journal file logs all the individual updates that make up the transaction. The transaction journal file is fully updated and closed before any updates are made to the table files. After the updates are done and the table files are written to disk, the transaction journal file is deleted.

This methodology protects the transaction during a failure. If the server machine crashes while the transaction journal file is being written, then the transaction never took place (the table files were not updated). If the server machine crashes after the transaction journal file is deleted, the transaction was already fully committed and the database integrity is assured. The interesting case occurs between these two. If the server machine crashes while the tables are being updated (the table files have been partially changed), their integrity is highly suspect. The next time FlashFiler Server is run, it checks all of its defined aliases for a transaction journal file. If any are found, the administrator is prompted to commit the transactions or roll them back. Either way, FlashFiler Server has ensured the integrity of the database.

Normal transactions do not log updates to a transaction journal file. This makes them faster than fail-safe transactions, because there is less file I/O. However, the price for the faster speed is increased risk. If the server machine crashes during the commit phase, the only recovery available is to restore the table files from a recent backup.

## Transaction manager

FlashFiler Server uses a transaction manager to track the active **transactions** within a database. Each **database** has its own transaction manager. The transaction manager assigns each transaction a Log Sequence Number (LSN) which, when a file **block** is modified within the context of the transaction, is written to the file block. This allows the FlashFiler Server to know how recently the block has been modified and what transaction modified the block. When the FlashFiler Server closes a database (i.e., no clients have the database open), it writes the next LSN to a file named FFSRTRAN.CFG within the database directory. The transaction manager reads the next LSN from this file the next time the database is opened.

## Transport

When using FlashFiler as a client/server database, FlashFiler consists of two distinct parts: the client and the server. Packets of information are sent between the client and the server in a well-defined manner. The client sends a request packet to the server to ask it to perform some database activity. The server handles the requested activity and sends a reply back to the client with the answer. This answer can be either data or a simple error code.

These packets of communication are controlled via a transport. The TffLegacyTransport component supports three protocols: TCP/IP, IPX/SPX, and Windows messaging (the latter is known as **Single User Protocol**). The operating system, the network, and its drivers are responsible for performing the actual transmission of the packets. FlashFiler uses the standard APIs to send the packets.

TCP/IP is provided by Microsoft's Winsock layer. It is the standard protocol for Internet communications, and is widely used in local area networks. It is generally more complex to set up than other network protocols. This is the preferred protocol for FlashFiler.

On Windows machines, IPX/SPX can be provided by two different drivers: Microsoft's Winsock and Novell's Client32 drivers. FlashFiler only supports Microsoft's version. IPX/SPX is easier to set up than TCP/IP and is a logical choice for smaller LANs.

Single user (Windows messaging) is not really a network protocol, since it only applies between processes running on the same machine, however FlashFiler creates a protocol look-alike so that single user applications can be supported (i.e., those that don't require a network to run).

If you tell FlashFiler to use TCP/IP or IPX/SPX, it conducts a prioritized search for a Winsock DLL. It attempts to use Winsock 2 but will use Winsock 1 if that is the only Winsock DLL available. If no Winsock DLL is available then FlashFiler client raises an exception and FlashFiler Server lists the transport as "Driver not installed".



---

## Chapter 3: FlashFiler Server

FlashFiler is a client/server database engine. Its client applications send requests for data and operations to the server application, and it is the server that performs the actual request (opening a table or getting a record) on behalf of the client. The server sends the results of the request back to the client that requested it.

The main alternative to a client/server methodology is the file manager type database engine where all the clients have direct access to the files that make up the tables in the database. The benefits of the client/server architecture over the traditional file manager type are many; the most important being the increased data integrity offered by such a scheme. A FlashFiler Server is the only application that opens and updates tables. The client applications have no direct access to the files that make up the tables. This ensures that clients cannot, through programming bugs or power outages, corrupt the data in the tables. FlashFiler allows you to increase data integrity further by giving you the option of using fail-safe transactions to maintain a journal of table updates.

For single-user applications, FlashFiler also provides the opportunity to embed a server engine directly into the client application. This functionality does not take away the client/server aspect of FlashFiler—it is still there under the hood. However, a separate server application is no longer needed since the server code is included in the client application.

For maximal data integrity, FlashFiler Server opens all its files in exclusive (non-shareable) mode. These files not only include the server's own tables, but also the table files opened on behalf of client applications. This does not mean that clients cannot share tables and data. It means that the server application makes sure that no other process can open the files. Once a table has been opened on behalf of one of more clients, another application cannot open the physical file. This becomes a problem only during backups. When a client has a table open, the server has the physical files open for that table, and so the backup program cannot open the files to copy them to the backup device. If it were possible for a backup program to open the table's files, it would have no way of guaranteeing that the data it backs up is coherent—the server may be in the middle of committing or rolling back a transaction.

This chapter describes the FlashFiler Server application, its configuration and operation. It also explains the FlashFiler Service, journal recovery, how to encrypt tables, and deploying FlashFiler Server. The FlashFiler Server executable is called `FFServer.EXE` and is located in FlashFiler's `Bin` subdirectory. The FlashFiler service executable is called `FFSrvce.EXE` and is also located in the `Bin` subdirectory.

---

# Configuration

The FlashFiler Server application maintains its global configuration in tables. The type of information stored is fourfold:

- General, network, and start-up information
- The set of user IDs for accessing FlashFiler in secure mode
- The set of database aliases and their paths
- The set of definitions for user-defined indexes

The Config menu on the main FlashFiler Server screen provides access to all four of these configuration types.

FlashFiler Server maintains three lists of information: alias definitions, user-defined index descriptions, and user ID definitions. An alias definition is a shortcut to a FlashFiler database which points to the complete path of the database. A user-defined index description is a complete identification of a table index (by the path and name of the table and the index number), and the identification of the DLL that performs key creation and comparison for the index (by complete path and name, and the routine names). A user ID definition contains a user's name and a list of their rights to access and modify FlashFiler tables. The user rights include administrative, read, update, insert, and delete rights. All of the configuration options are fully explained later in this chapter.

## FlashFiler Server Tables

FlashFiler Server stores its configuration in standard tables, known as server-specific tables. The tables are called FFSINFO.FF2, FFSUSER.FF2, FFSALIAS.FF2, and FFSINDEX.FF2. FlashFiler Server can be run in secure or non-secure mode. The default, secure server encrypts these server tables with a special method to ensure that client applications cannot read and alter them—all configuration changes must go through the server. The non-secure server does not encrypt the server-specific tables; hence they can be read and altered with standard FlashFiler client applications, like the FlashFiler Explorer program. Note, however, that FlashFiler Server will always recreate these server-specific tables using information stored internally in memory when their data changes. This ensures that the data integrity remains high, but does mean that client applications cannot alter the server-specific tables and expect the data changes to be permanent.

The security mode setting is located on the General Configuration screen and is labeled "Creation of encrypted tables enabled." The General Configuration screen and the security setting are discussed later in this chapter.

# Operation

To start FlashFile Server, run FFServer.EXE. The FlashFile Server main screen is displayed. If no server-specific tables are found, the communications layer is not started (i.e., it is not listening for client applications). If the server-specific tables are found, then the server will start according to the configuration information contained in the FFSINFO.FF2 table. If the communication layer is started, it will be indicated in the state of the transports shown on the lower portion of the main screen.

Figure 3.1 shows FlashFile Server's default main screen, except that the server name is "FF2" and the debug log is on so you can see how and where these things appear.

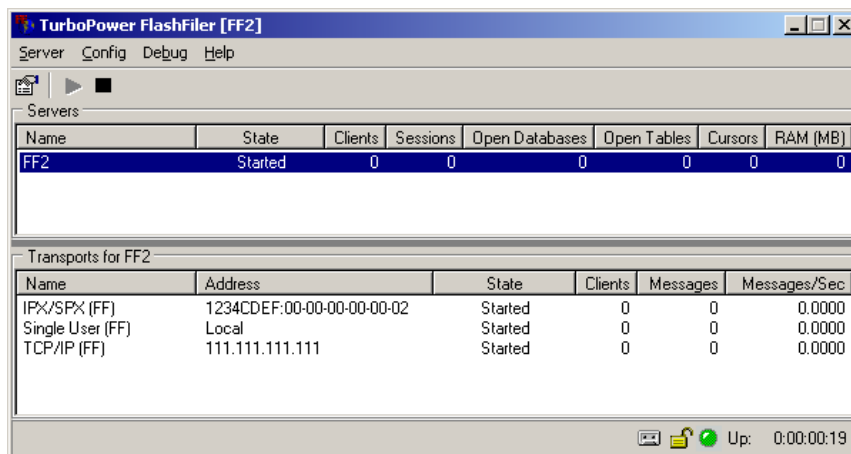


Figure 3.1: FlashFile Server's main screen with server name and debug log visible.

The menu allows access to the various functions of the server (bringing the server up, setting the debug logging option, etc.), as well as accessing the configuration options of the server. The menu is discussed in detail in the following section.

Directly below the menu is a toolbar. The toolbar's speed buttons give easy access to frequently used menu options. The first speed button allows you to define the properties of the server. The next two speed buttons bring the server up and take it down (they are equivalent to Server | Start and Server | Stop in the menu). Menu options are described in detail in the following section.

The body of the main screen is divided into two sections: servers and transports. The upper section is the Servers section. It lists all the servers in your FlashFile Server application. For each server, you can see its name, state, the number of clients logged on, the number of active sessions, the number of open databases, the number of open tables, number of



cursors, and how much RAM the server is currently using to cache data from the tables it has open. We ship FlashFiler Server with a single server. “Chapter 8: Advanced Architecture” on page 269 includes information about adding additional servers.

The lower section of the screen is the Transports section. This section lists all the transports attached to the server selected in the Server section. For each server, you can see its name, address, state, the number of clients attached via this transport, the number of messages it’s processed since it was started, and the average number of messages it is processing per second (this number constantly changes as time passes). You can reset the message count and message processing statistics by using the Debug | Reset Counters menu option. Resetting these statistics allows you to get a snapshot of the network traffic. We include three transports with each server: TCP/IP, IPX/SPX, and Single User. Chapter 8, “Advanced Architecture,” (page 279) explains how to customize the included transports and how to create your own custom transports.

The circle icon in the bottom right corner of the status bar is green when the selected server is listening for clients (in other words, the server(s) are started) and red when it is not. If you see a cassette tape icon to the left of the red or green lights, it means debug logging is enabled. Otherwise, debug logging is disabled. Figure 3.1 shows that debug logging is turned on. The padlock lets you know if security is enabled for the selected server in the Server section. The padlock is closed if security is enabled and open if security is disabled. The counter shows you how long the server(s) have been up and listening for client requests. The counter, server statistics, and transport statistics are refreshed once per second.

## The Menu

The menu provides access to server-related options, configuration options, and debugging options. If FlashFiler Server is in secure mode, you must log in before performing actions such as starting the server application, bringing the server up or down, or restoring the server window from a minimized state. When you attempt to perform one of these actions, the Server Log On dialog box is displayed, as shown in Figure 3.2.

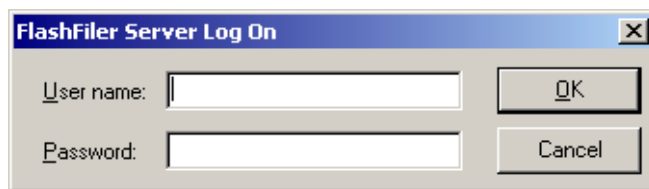


Figure 3.2: The Server Log On dialog box.

Enter your user ID and password. If they are correct, the action you requested is performed. See “Security” on page 25, for more information on the security levels in FlashFiler Server.

## Server | Name

Defines the name of the selected server. The name can be up to 15 characters. You can use the letters A-Z (both upper and lower case), numbers 0-9, and embedded spaces. This option is disabled if the server is up.

## Server | Start

Starts the servers. After initializing the network drivers, the servers start listening for clients. This option is disabled after the servers are up and listening.

## Server | Stop

Stops the servers. This option is disabled if the servers are already down.

## Server | Exit

Terminates the FlashFile Server application. If the servers are up and listening for clients, they are automatically stopped before the server application terminates.

## Config | General configuration...

Allows you to set the start-up configuration for the selected server. This menu item is available whether the server is up or not. However, certain fields of the general configuration dialog are disabled if the server is up. Figure 3.3 displays the FlashFiler Server General Configuration dialog box.

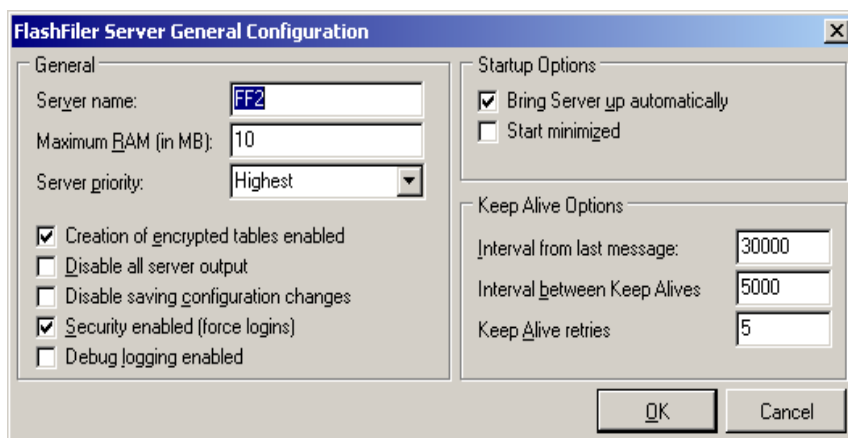


Figure 3.3: The FlashFiler Server General Configuration dialog box.

You can name the server by entering a name in the “Server name” edit control. “Server name” is disabled if the server is up.

“Maximum RAM (in MB)” is used for sizing the server’s internal cache. The server caches pages from the tables it opens in a special cache. This makes the server more efficient in providing records to clients because the page on which the record exists may already be in

the cache and doesn't have to be reread from disk. It also helps with transactions because the pages altered by a transaction are marked "dirty" and kept in the cache until the transaction is committed or rolled back. At that point the dirty pages are either written to disk (commit) or are marked empty (rollback). The server reuses the pages in the cache as it runs. To make sure that the server doesn't keep on allocating memory infinitely, the server tries to keep the amount of pages in the cache below the Maximum RAM value. For example, if the value were set to 10 megabytes, the server would start reusing the least recently used pages once it had filled its 10 megabytes of cache. Dirty pages are never reused, and indeed, if a transaction is large enough, the server will willingly exceed the maximum limit to ensure that the whole transaction is kept in memory. As such, the Maximum RAM value is an advisory value. The default is 10.

The "Server Priority" combo box defines the priority given to the server application. In 32-bit Windows, the foreground process runs at a higher priority than any background process, so if the server is minimized, it is possible for clients to appear sluggish, because the server does not have enough priority to fully process client requests quickly. The available options are Lowest, Below Normal, Normal, Above Normal, and Highest. You need to experiment to find the optimum setting for your system. The default is Highest.

The "Creation of encrypted tables enabled" check box defines whether the server creates its own tables in an encrypted form or not. If the box is checked, the server tables created by FlashFile Server are encrypted and can only be read by the server application itself. FlashFile client applications will not be able to open the tables—the encryption used is different than that used for the client tables. The default is checked.

The "Disable all server output" check box defines whether the server is allowed to output any data (table updates, debug log, and configuration tables). If the box is checked, the server does not produce any output. You may want to enable this setting to ensure your data is not modified. The default is not checked.

The "Disable saving configuration changes" check box defines whether the server updates its configuration tables. The check box will be disabled if "Disable all server output" is enabled. If the box is checked, you can modify server options but they will not be retained after the FlashFile Server application is closed. You may want to enable this setting while testing different settings. The default is not checked.

The "Security enabled" check box allows you to specify secure mode for the server. See "Security" in Chapter 2, "Concepts and Glossary" for more information on secure mode. Default: Not checked.

The "Debug logging enabled" check box, if checked, will cause every message received and replied to by the server to be written to a text file, FFSERVER.LOG (it will be created in the same directory as the server application). The actual information written to the text file depends on the messages being received by the server. The log is a debugging tool to see which messages are being received by the server from a client when the client performs a

particular function. Enabling the debug log causes a dramatic drain on performance. We recommend that you only enable this setting while debugging. This option performs the same operation as the Debug | Log option on the main menu. The default is not checked.

There are two options in the “Startup Options” group. If “Bring Server up automatically” is checked, the server is brought up when the application starts. If “Start minimized” is checked, the server application minimizes to a tray icon when it is run. The default for both options is not checked.

The set of fields in the “Keep-Alive Options” group define what happens when there is a period of inactivity between the client and server. Because of the way that most network protocols are written, you are only notified if the connection between client and server is broken when either partner attempts to send a message across the connection. Since the server never sends a message to a client unless it is a reply to a request from that client, it is very likely that the server would not notice that a connection was broken (for example, the client had crashed, the client machine has been powered off, and so on). If the client were to crash half way through a transaction, there is no way for the server to know, and hence all other clients would be locked out from updating portions of the database. It is in this situation that the Keep-Alive options come into play.

Periodically both the client and the server will send each other a message just to say that they are alive and running. If either side does not receive a message within a certain amount of time, the connection is deemed to be broken. This connection time-out processing proceeds like this: after a period of “Interval from last message” milliseconds from when the last standard message was sent or received, the first keep-alive message is sent. Keep-Alive messages will then be sent every “Interval between Keep-Alives” milliseconds, until a total of “Keep-Alive retries” messages have been sent. At this point, the application, be it client or server, will assume that the partner is no longer reachable, and close the connection. If a message (standard or Keep-Alive) is received anywhere within this process, the entire process is reset to the beginning again. Defaults values for these options are shown in Figure 3.3.

## Config | Network configuration...

Allows you to configure the default transports for the selected server. The Network configuration page has three sections: Winsock TCP/IP Transport, Winsock IPX/SPX Transport, and Single User Transport. All three transports have an “Enabled” check box that allows you to enable or disable the transport. Only Single User is enabled by default. Figure 3.4 shows the Network Configuration dialog box.

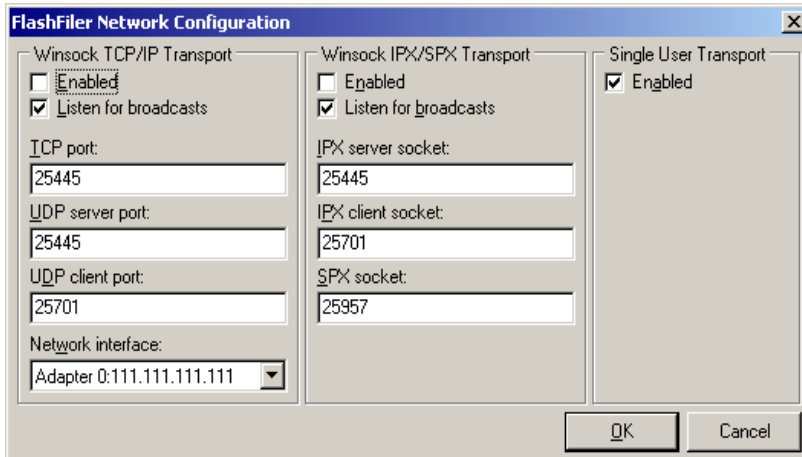


Figure 3.4: The FlashFiler Network Configuration dialog box.

The two Winsock transports have a “Listen for Broadcasts” option. If the “Listen for Broadcasts” option is checked, the server listens for (and responds to) client applications asking for a list of server names. The FlashFiler Client communications layer has a routine that retrieves all server names on a network. It performs this task by broadcasting a request across the network for all listening servers to send back a message containing the server name and net name. Under certain circumstances, you won’t want your server to respond to such a request (e.g., it is supposed to be invisible or unavailable to all network users), so you can prevent the response by clearing this box.

Using the TCP/IP transport, the client and server, once connected, will communicate over the port specified by “TCP Port.” The client broadcasts a “Who is there?” message to all servers on the port specified by “UDP Server Port”. The server responds to such messages on the port specified by “UDP Client Port”. The “Network Interface” list box allows you to choose which installed network card the TCP/IP transport should use.

Using the IPX/SPX transport, the client and server, once connected, will communicate over the “SPX Socket”. The client broadcasts a “Who is there?” message to all servers on the “IPX Server Socket”. The server responds to such messages on the socket specified by “IPX Client Socket”.

The values shown in Figure 3.4 are the defaults. If you change the ports or sockets, you are responsible for altering the client values to match.

The OK button saves all data in a server table, providing the “Disable all server output” and “Disable saving configuration changes” check boxes (Config | General Configuration...) are not checked. Whether or not the check boxes are checked, the changes are available immediately during the current run of the server application. The Cancel button discards all changes.

### Config | User Permissions...

Allows you to add, delete, and modify the users known to FlashFile Server. These user attributes are only used if the server is being run in secure (i.e. users must log on) mode. This menu item is available whether the server is up or not. When you select this item, the FlashFile Server User Permissions dialog box is displayed, as shown in Figure 3.5.

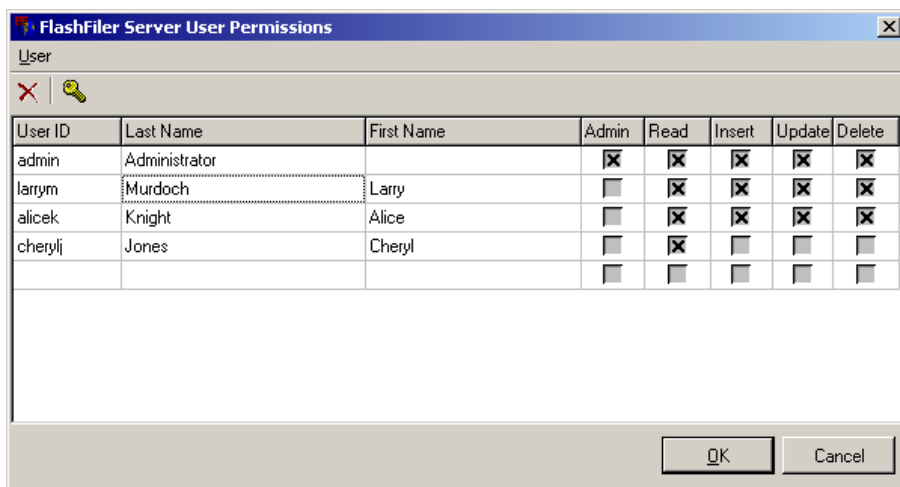


Figure 3.5: The FlashFile Server User Permissions dialog box.

The admin user is defined by default. This user has full rights. The user's default password is “flashfiler”.

You can define additional users by entering a User ID, Last Name, First Name, and a set of rights for the user. A user can have administration, read, insert, update, and delete rights. A user with administration rights can log on directly to FlashFile Server and change any of the configuration options. A user with read rights can open and access databases and tables in read-only mode. A user with insert rights can create new tables and insert new records in tables. A user with update rights can rename, restructure, pack, and re-index tables and modify records in tables. A user with delete rights can delete existing tables and delete records from tables. You can sort the list of users by any column by simply clicking on its header.

The “Delete User” button deletes the selected user. You cannot delete the admin user. The “Password” button displays the Password dialog so you can enter a password for the selected user. The password must be entered twice, the second time for verification purposes. If no password is supplied for a user, the user is marked as unavailable (the user ID is displayed as a red cell).

The OK button saves all data in a server table, providing the “Disable all server output” and “Disable saving configuration changes” check boxes (Config | General Configuration...) are not checked. Whether or not the check boxes are checked, the changes are available immediately during the current run of the server application. The Cancel button discards all changes.

### Config | Aliases...

Allows you to add, delete, and modify the database aliases known to FlashFiler. This menu item is available whether the server is up or not. Deleting or modifying an alias while it is already open does not cause any difficulties. Clients that have the database alias open will continue to use that alias in its original form; eventually when all the clients that have the database alias open close it, the alias will no longer be available in any form.

When a client opens a database by its alias, the alias is converted to a path and the path is used from then on internally to the server.

When you select this item, the FlashFiler Server Aliases dialog is displayed, as shown in Figure 3.6.

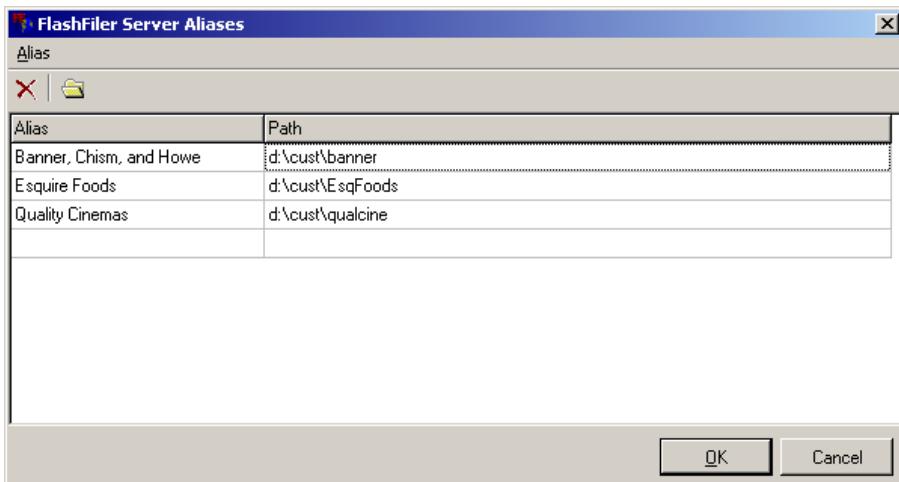


Figure 3.6: The FlashFiler Server Aliases dialog box.

“Alias” is a simple identifier for a database and “Path” is the full directory path to the database. FlashFiler Server converts the path you enter to a full universal naming convention (UNC) path. A UNC path is one of the form \\MACHINE\SHARE\FOLDER. Unshared local directories are stored in the standard DRIVE:\FOLDER format. Whether the path is converted or not, it must be available from the PC on which the server application is running. The server must be able to open and update files in that folder.

The columns in this dialog can be resized. Move the mouse over the dividing line between the column header cells until it changes shape, click, and drag. You can sort by a particular column by clicking on its header.

The “Delete Alias” button deletes the selected alias. The “Browse...” button makes it easier to select a path for an alias. It is enabled only if the grid highlight (the active cell) is in the “Path” column.

The OK button saves all data in a server table, providing the “Disable all server output” and “Disable saving configuration changes” check boxes (Config | General Configuration...) are not checked. Whether the check boxes are checked, the changes are available immediately (as described in the first paragraph of this section) during the current run of the server application. The Cancel button discards all changes.

#### Config | User-Defined Indexes...

Allows you to add, delete, and modify user-defined indexes. See “Chapter 16: Appendices” on page 523 for more information on user-defined indexes. This menu item is available whether the server is up or not. However, if you alter a user-defined index definition while its table is open, the behavior is unpredictable.



When you select this item, the FlashFiler Server User-defined Indexes dialog box is displayed, as shown in Figure 3.7.

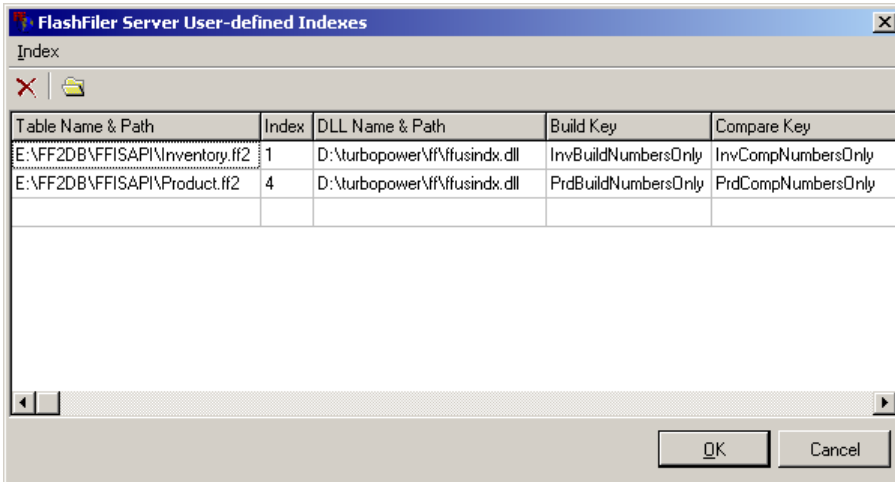


Figure 3.7: The FlashFiler Server User-defined Indexes dialog box.

The “Table Name & Path” and “Index” columns define the indexes for which you will supply build key and compare key routines. You must enter a complete file name and path in “Table Name & Path”. FlashFiler converts the path you enter to a full universal naming convention (UNC) path. A UNC path is one of the form \\MACHINE\SHARE\FOLDER\FILE. Unshared local files are stored in the standard DRIVE:\FOLDER\FILE format.

“DLL Name & Path” is the name of the DLL that contains the build key and compare key routines. You must enter a complete file name and path in “DLL Name & Path”. Your entry is converted to the full UNC path. The “Build Key” and “Compare Key” columns define the names of the routines.

The columns in this dialog can be resized. Move the mouse over the dividing line between the column header cells until it changes shape, click, and drag.

The “Delete” button deletes the selected index definition. The table to which it refers is not altered. The “Browse...” button makes it easier to select a path for a “Table Name & Path” or “DLL Name & Path” cell. It is enabled only if the grid highlight (the active cell) is in one of those two columns.

The OK button saves all data in a server table, providing the “Disable all server output” and “Disable saving configuration changes” check boxes (Config | General Configuration...) are not checked. Whether the check boxes are checked, the changes are available immediately during the current run of the server application (with the warning from the first paragraph of this section). The Cancel button discards all changes.

## Debug | Log

The debugging log shows the messages FlashFiler Server received from clients and the responses it made. It also logs all exceptions generated internally by FlashFiler Server. Exceptions raised internally by the server in response to client requests are sometimes converted into BDE error codes. In the conversion, information can be lost and the log is the only way to see that information. A check mark in the menu by the Log option shows that logging is active. The log is written to the file, FFSERVER.LOG, in the directory from which the server application was started. Note that internal exceptions generated by the server application are always written to the log file, whether debug logging is active or not. The only time internal exceptions aren't written to the log file are when the "Disable all server output" setting is checked on the General Configuration page.

Logging is an extremely disk-intensive process. To ensure that the log is always valid, the server opens the log file, writes a line, and closes the file for every line in the log. Consequently, the server is significantly slower if logging is on. We recommend that you only use the debug log while testing or debugging. The default is not checked.

## Debug | Reset Counters

This option resets the counter and transport message-related statistics.

## Help | About...

Displays a screen of information about FlashFiler Server.

## Help | Visit TurboPower's Web Site...

Starts your web browser, passing the URL address of the TurboPower Software Web site.

---

## FlashFiler Service

One limitation of FlashFiler Server is that the hosting workstation must always be logged in. For customers using Windows NT or Windows 2000, the FlashFiler Service can be used in place of the FlashFiler Server. The FlashFiler Service executable is named FFService.EXE and is located in the Bin subdirectory of your FlashFiler installation.

FlashFiler Service is compatible with Windows NT and Windows 2000. It supports TCP/IP, IPX/SPX, and Single User Protocol (SUP). SUP sees requests only if the service's Allow Service To Interact With Desktop property is enabled.

To install FlashFiler Service, invoke the following from your installation program or command prompt:

```
FFService /i
```

To uninstall the service, invoke the following from your installation program or command prompt:

```
FFService /u
```

You may run FlashFiler Service as a normal program (i.e., not as a service) by typing the following at the command prompt:

```
FFService /c
```

Prior to starting FlashFiler Service, you must make sure that it has the appropriate configuration information. See “Deploying FlashFiler Server” on page 54 for more information. If you must configure the service after it has been installed, do the following:

1. Stop the FlashFiler Service.
2. Make sure the Windows Event Viewer is closed.
3. Make sure a FlashFiler Server executable is located in the same directory as the FlashFiler Service executable.
4. Start FlashFiler Server and make the required configuration changes.
5. Stop FlashFiler Server.
6. Start FlashFiler Service.

FlashFiler Service registers itself as an event message source and, if an error occurs, writes error messages to the Windows event log. This means that Windows NT/2000 will open the FlashFiler Service executable if you open the Event Viewer and the Event Viewer contains messages posted by FlashFiler Service. You will not be able to replace the FlashFiler Service executable until the Event Viewer is closed.

---

## Transaction Journal Recovery

All updates to FlashFiler tables are performed within transactions. If a client wishes to update one or more records, a transaction is started, the updates are made, and then the transaction is either committed or rolled back. The transaction can be explicit—the client makes an explicit request to the server to start a transaction—or implicit—the server starts a transaction if it detects an update is occurring and commits afterwards. The client does have the choice of making the transaction fail-safe or leaving it as a normal transaction: fail-safe transactions are selected by setting the FailSafe property of the TffDatabase component to True. The TffDatabase component and its properties are described in Chapter 7. The fail-safe transaction mode must be set for each database used in the application, and once set will remain so until the FailSafe property is set to False. The default is to use normal (non fail-safe) transactions.

If a client is using fail-safe transactions, updates are initially written to a transaction journal file rather than straight to the database itself. It is only when the transaction is committed, and the transaction journal file is closed and flushed to disk, that the updates are written to the database. If a crash occurs during the database update (and the tables are thereby not updated correctly), the database can be made good the next time the FlashFiler Server is started, using the information saved in the transaction journal file.

When a fail-safe transaction is started on a database, a transaction journal file is created. An “incomplete journal header” is written to the transaction journal file at this time. Keep in mind that FlashFiler 2 supports multiple transactions on a database. Each transaction will have a separate journal file.

Just before a page of memory is changed in the server cache, a before image of the page is written to the transaction journal file. Then the page is marked dirty. The dirty page is known as an after image. Changes are just made to the page in memory; no disk files are updated at this time.

If the transaction is rolled back, the transaction journal file is deleted and the dirty pages in the cache are discarded. Changes were not written to the actual database files, so no further action is necessary.

If the transaction is committed, the following process occurs. First, the after images are written to the transaction journal file. Second, the transaction journal file is marked complete (the “incomplete journal header” is overwritten with a “complete journal header”) and the file is closed, flushing all data to disk. Third, all the database pages marked as dirty in the server cache are written to the disk. Finally, the transaction journal file is deleted.

Fail-safe transactions become important if the server application crashes during the third step. In this case, the user believes that the (committed) fail-safe transactions have been fully written to the database, when, in actuality, they have only been saved to the transaction journal file.

Each time FlashFiler Server is started, it performs transaction journal recovery before it becomes available as a database server. For each database defined as an alias, it searches for all transaction journal files. If it finds a transaction journal file that is marked incomplete, it cannot be processed (indeed the database is valid since nothing was written to it), so the journal file is simply deleted. This is akin to rolling back an incomplete transaction.

If, however, it finds a transaction journal file marked as complete, a dialog is displayed for the FlashFiler Server administrator. The dialog displays the database alias and file names, notes that a complete transaction journal was found, and asks whether to commit or rollback the changes. If the administrator chooses to rollback the changes, the before images in the journal are written to the database files and the transaction journal file is deleted. If the administrator chooses to commit the changes, the after images are written to the database files and the transaction journal file is deleted. After all of the transaction journal files are processed, the server starts its network protocol, and becomes available to act as a database server.

Note that should a FlashFiler client application crash in the middle of a transaction, the server will eventually notice that the client has disappeared (see the discussion on “Keep-Alives” earlier in this chapter) and will roll back all transactions for that client.

# Server Scripting

The server has the ability to read its initial configuration from a script file. The script file is read and the commands processed when the server first starts. Since the script is acted upon after the server has read configuration data from its own tables, you can:

- Use a script instead of server tables to store the server's configuration.
- Use a script to set up the server part of a SingleEXE application.
- Reduce the number of support files you must ship with FlashFiler Server (indeed, your installation program can create the script file when your product is installed).

The configuration items supported by the server-scripting engine are most of the general configuration items (excluding the port/socket numbers and the debug logging option) and alias definitions. User-defined indexes and user IDs are not supported with this scripting facility. The script is a simple ASCII file, structured as a series of lines of the form KEY=VALUE. Both KEY and VALUE strings are case-insensitive.

If you distribute a FlashFiler Server or singleEXE application containing multiple server engines, you may include two or more sets of configuration items within the same script. Each set of configuration items must be prefixed with a line having the following format:

```
[<server component name>]
```

where <server component name> is the value of the server engine's Name property. If the first items encountered in a script are not tied to a specific server engine then the items are treated as though they apply to all server engines. For example, the following script sets the MAXRAM and AUTOUPSERVER items for all servers. It then sets the SERVERNAME for two different servers, one named ProdEngine and the other named TestEngine.

```
AUTOUPSERVER=TRUE
MAXRAM=100

[ PROENGINE ]
SERVERNAME=Production

[ TESTENGINE ]
SERVERNAME=Test
```

Table 3.1 shows the scripting keys corresponding to the FlashFiler Server General Configuration screen, including a brief description and possible values. See “Config | General configuration...” on page 35 for details about setting a particular key using FlashFiler Server.

**Table 3.1:** *The keys corresponding to the FlashFiler Server General Configuration screen.*

Key	Description	Possible Values
SERVERNAME	The server name	String up to 15 characters with the letters A-Z (both upper and lower case), numbers 0-9, or embedded spaces
MAXRAM	The recommended maximum amount of RAM (in megabytes) the server will use to cache FlashFiler tables	Number from 0 up to available RAM
USELOGIN	Whether the server requires users to log on	TRUE, FALSE, YES, or NO
AUTOUPSERVER	Whether the server starts automatically	TRUE, FALSE, YES, or NO
AUTOMINIMIZE	Whether the server minimizes automatically	TRUE, FALSE, YES, or NO
ALLOWENCRYPT	Whether the server will create encrypted tables	TRUE, FALSE, YES, or NO
READONLY	Whether to prevent output from the server	TRUE, FALSE, YES, or NO
LASTMSGINTVAL	How many milliseconds to wait before sending the first Keep Alive message	Any number from 1,000 to 86,400,000
ALIVEINTERVAL	How many milliseconds to wait between Keep Alive messages	Any number from 1,000 to 86,400,000
ALIVERETRIES	How many total Keep Alive messages to send	Any number from 1 to 100

**Table 3.1:** *The keys corresponding to the FlashFiler Server General Configuration screen. (continued)*

Key	Description	Possible Values
PRIORITY	Priority given to the server application	LOWEST, BELOWNORMAL, NORMAL, ABOVENORMAL, or HIGHEST
DELETESCRIPT	Whether the script file is deleted after processing has occurred	TRUE, FALSE, YES, OR NO
NOAUTOSAVECFG	Whether the configuration tables can be updated from the server	TRUE, FALSE, YES, OR NO

Table 3.2 shows the keys corresponding to the FlashFiler Server Network Configuration screen. See “Config | Network configuration...” on page 38 for more information concerning these keys.

**Table 3.2:** *The keys corresponding to the FlashFiler Server Network Configuration screen.*

Key	Description	Possible Values
IPXSPXLFB	Whether the Winsock IPS/SPX Transport should listen for broadcasts	TRUE, FALSE, YES, or NO
TCTPIPLFB	Whether the Winsock TCP/IP Transport should listen for broadcasts	TRUE, FALSE, YES, or NO
TCPINTERFACE	Which network interface(s) to bind the FlashFiler Server to	-1 to (total interfaces minus 1)
USEIPXSPX	Whether to enable the Winsock IPX/SPX Transport	TRUE, FALSE, YES, or NO
USESINGLEUSER	Enable the Single User Transport	TRUE, FALSE, YES, or NO
USETCPIP	Whether to enable the Winsock TCP/IP Transport	TRUE, FALSE, YES, or NO

Any other KEY value is assumed to be an alias name and the VALUE part is assumed to be the path. That means that you cannot create an alias named as one of the above identifiers.



Here is an example script:

```

ServerName=MyServer
MaxRAM=15
UseLogin=False
AutoUpServer=yes
AutoMinimize=False
PRIORITY=HIGHEST
USESINGLEUSER=True
USETCPIP=True
USEIPXSPX=False
TCPIPLFB=True
TCPINTERFACE=-1
AllowEncrypt=no
ReadOnly=no
NoAutoSaveCfg=True
DELETESCRIP=no
TestDB=d:\flashfiler\TestDB

```

Dissecting this script we see the following:

- The server will start a server with the name “MyServer”.
- A maximum of 15 megabytes of RAM will be used for its cache.
- The server will not require users to log on.
- The server is to start automatically but won’t minimize.
- The server will execute at the highest priority.
- Single User and TCP/IP transports are enabled.
- IPX/SPX transport is disabled.
- The server will advertise itself via TCP/IP.
- The TCP/IP transport will be bound to all network interfaces.
- It will not create any new tables that are encrypted.
- The server is allowed to write data.
- The server won’t save its configuration.
- The server will not delete the script after processing it.

One alias is to be created: the TestDB alias pointing to the D:\FLASHFILER\TESTDB directory.

Once a script has been created, you get the server to read it by passing the script file name as the first parameter on the server command line. For example, you could set up an icon on your desktop that executed the following command:

```
D:\FF2\SERVER\FFServer C:\DATABASE\FFSERVER.SCR
```

When you double click on this icon, the FlashFiler Server would start and read and process the C:\DATABASE\FFSERVER.SCR script. The same thing applies to singleEXE applications: the first parameter to your application's command line would be the script file.

It must be emphasized that a script file is read after the current configuration of the server is read. This means that items in a script file override the values in the normal server configuration tables.

---

## Installing New Encryption Routines

In your `\FF\EXAMPLES\DELPHI` folder is an example `FFTBCRYPT.PAS` file that replaces the one we ship pre-compiled into the FlashFiler Server. You can change the encryption algorithm implemented in this unit and then recompile the server application. This will enable you to have encrypted tables that only you can read.

The `FFTBCRYPT` unit provides four routines that you will have to replace: `FFCodeBlock`, `FFDecodeBlock`, `FFCodeBlockServer`, and `FFDecodeBlockServer`

The first two routines encrypt and decrypt a block of table data. Recall that a FlashFiler table is a file (or set of files) of many blocks, each of which is the same size: 4, 8, 16, 32, or 64KB. `FFCodeBlock` is called just prior to writing one of these blocks to disk and so is responsible for encrypting the data. `FFDecodeBlock` is called just after reading a block from disk and so is responsible for decrypting the encoded data.

The second two routines do exactly the same as the first two, except the table is one of the server-specific tables, and must be encrypted in a different fashion. The reason for this is to avoid a situation where a user maps an alias to the server directory and reads the server-specific tables directly. If the server tables are encrypted in a different fashion they will be unintelligible and only usable by the internal code in the server.

The parameters to the routines are the same in all four cases. `aBlock` is a pointer to the block to be encrypted/decrypted. `aBlockLen` is the length of this block. `aRandomizer` is a special value that is different for different blocks (actually, it's the block number). This will introduce a "random" element into the encryption/decryption process, so that each block can be encrypted in a slightly different way if you so wish (if you don't, just ignore the parameter).

Tables are encrypted in two situations: first, when they are created with the encryption option set, and second, if they are server-specific tables. (Note that, for either of these two cases to apply, the `SecureServer` compiler define must be active in `FFDEFINE.INC.`) The header record for FlashFiler tables (the first block in each table file) is not encrypted. This allows non-secure servers to read the header and determine that the table is encrypted. The routines you write in this unit will not be called for header blocks.

FlashFiler, by default, does not use passwords to encrypt and decrypt table data. Consequently, you could try to fashion a “password” as key for your encryption routine from the aRandomizer parameter.

You can replace the encryption algorithm with any encryption routine you have at hand, a good source for such routines (with printed code in C) is *Advanced Cryptography* by Bruce Schneier (Wiley). The encryption routine used in the example unit is the LockBox Stream Cipher from TurboPower's LockBox library. Simpler alternatives could be XORing a key (possibly a random key) over and over into a block: the longer the key the better. Be aware, however, that simpler encryption algorithms are usually more easily broken.

---

## Deploying FlashFiler Server

Deploying FlashFiler Server is quite simple since you are free to distribute the FlashFiler Server application and service along with your own applications. The only precaution is ensuring your application has the configuration data it needs, especially FFSINFO.FF2 and the required database aliases.

Since you may not know where a user will install your product, you probably don't want to ship your application with pre-defined aliases. You have two choices:

- During installation, create a script file containing the required aliases. Run the FlashFiler Server, specify the script file as the first parameter on its command line.
- Modify your program to create aliases on the fly. You can create aliases using the `TffSession.AddAlias` method, as shown in the following example:

```
MySession.AddAlias( 'MyApp', 'D:\MyFF2App' );
```

If you are deploying a FlashFiler service, you must install a pre-defined FFSINFO.FF2 file or manually define the information through FlashFiler Server. You must create the aliases manually through the FlashFiler Server or programmatically in code. The FlashFiler service does not accept or process a script file. You must place the server tables in the directory containing the service executable (i.e., `FFSrvce.EXE`).

If you are upgrading an existing FlashFiler site with a new version of FlashFiler Service, your installation program will not be able to replace the service executable until a) the service is stopped and b) the Event Viewer is closed. If the Event Viewer is open and it contains messages from FlashFiler Service, the Event Viewer keeps the FlashFiler Service executable open.

If your application is a SingleEXE application, you don't need to deploy the FlashFiler Server application but you still need to ensure the embedded server has the configuration information it needs. As with the FlashFiler Server application, it's probably best to create your aliases using information gathered during the installation process.

---

# Chapter 4: FlashFiler Explorer

FlashFiler Explorer is a database management client utility that works in tandem with one or more FlashFiler Servers. FlashFiler Explorer allows you to:

- Create, edit, and delete database aliases for a FlashFiler Server
- Create and view table structures and indexes
- Restructure existing tables
- View and edit table data
- Select records using SQL queries
- Filter and search records within tables
- Import data from external files

# Operating FlashFiler Explorer

When started, FlashFiler Explorer lists all active FlashFiler Servers that are available on the network. As shown in Figure 4.1, the outline displays the hierarchical structure of the FlashFiler network. FlashFiler Servers are the topmost level of the hierarchy. Only servers that are currently running are shown. (See “Server | Start” on page 37). You can expand the view into a FlashFiler Server by clicking the plus sign next to the server entry in the outline. This also attaches you to the selected server.

4

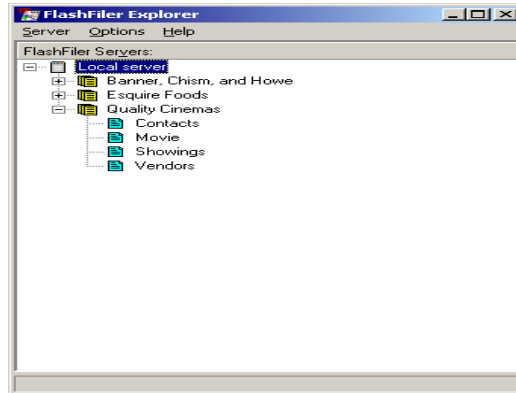


Figure 4.1: The FlashFiler Explorer main window.

The next level of the hierarchy lists all of the databases defined for the server. You can expand the view into a database by clicking the plus sign next to the database entry.

The lowest level of the hierarchy lists all of the tables defined within the database. Double clicking a table displays its contents in a separate Table Browser window. You can even view multiple tables simultaneously. See “The Table Browser” on page 73 for more information.

Clicking the minus sign next to an expanded level collapses that level of the hierarchical view. Right click on any object (server, database, or table) to display its context menu. Most of the work you do with these objects will be through context menus, so they are discussed first. The menu at the top of the dialog mostly provides configuration options and a few simple functions. It is discussed at the end of this section.

---

# The Server Context Menu

In the FlashFiler Explorer main dialog, select a server and then right-click to display the server context menu.

## Attach

Attaches to the selected FlashFiler Server. Note that you can also attach to a server by just double-clicking on it in the outline hierarchy. If you are attempting to attach to a secure server, you are asked to provide a username and password.

## Detach

Detaches FlashFiler Explorer from the selected FlashFiler Server (if attached). All open tables are closed and all database connections are dropped.

## Register...

Allows you to enter a server name in the list of registered servers. FlashFiler Explorer uses a broadcast request to find available FlashFiler Servers. It lists all of the FlashFiler Servers that respond to the broadcast in the hierarchical list on the main screen. However, some FlashFiler Servers do not respond to broadcast requests. See “Config | Network configuration...” on page 38 for further details. To display such FlashFiler Servers in the hierarchical list, FlashFiler Explorer maintains a list of registered servers. The registered servers are always shown in the hierarchical list.

## Unregister

Removes the currently selected server from FlashFiler Explorer's list of registered servers.

## Refresh

Refreshes the list of active FlashFiler Servers. All existing connections are dropped and FlashFiler Explorer searches the network for all active servers.



## New database...

Creates a new database for a FlashFiler Server. The Add Database dialog box is displayed as shown in Figure 4.2:

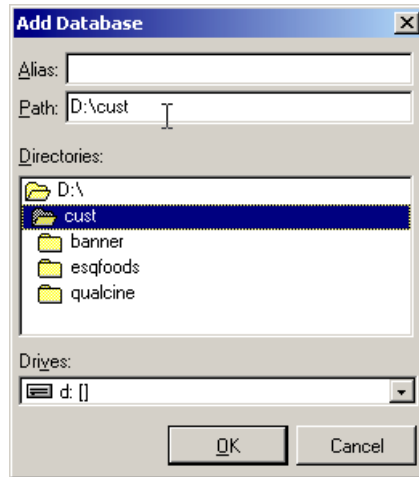


Figure 4.2: The Add Database dialog box.

A database is a collection of files in a directory on the network. Client applications refer to the database by an alias because that is more convenient than using the path. In a Delphi or C++Builder client application, you supply the alias name in the `AliasName` property of the `TffDatabase` component or the `DatabaseName` property of the `TffTable` component.

Enter a name in the “Alias” edit box. The name is not case-sensitive and must be unique for that server. Next, provide the directory path on the server PC that contains the files for the tables in the database. You can also select the drive and directory from the list boxes. If you enter a directory that does not exist, FlashFiler Explorer gives you the option to create the directory. FlashFiler converts the path you enter to a full Universal Naming Convention (UNC) path. A UNC path is one of the form `\\MACHINE\SHARE\FOLDER`.

**Note:** Ensure the path is relative to the server. If your server is not on the same machine as FlashFiler Explorer, the server will not recognize a path that is relative to the client machine.

---

# The Database Context Menu

In the FlashFile Explorer main dialog, select a database and then right-click to display the database context menu.

## New database...

Creates a new database for a FlashFile Server. The Add Database dialog is displayed (see Figure 4.2 on page 58).

## Delete

Removes the selected database alias from the FlashFile Server. The physical files associated with the database are not deleted.

## Rename...

Allows you to rename the selected database alias.

## SQL...

Allows you to select records using SQL statements. Figure 4.3 illustrates the SQL dialog box.

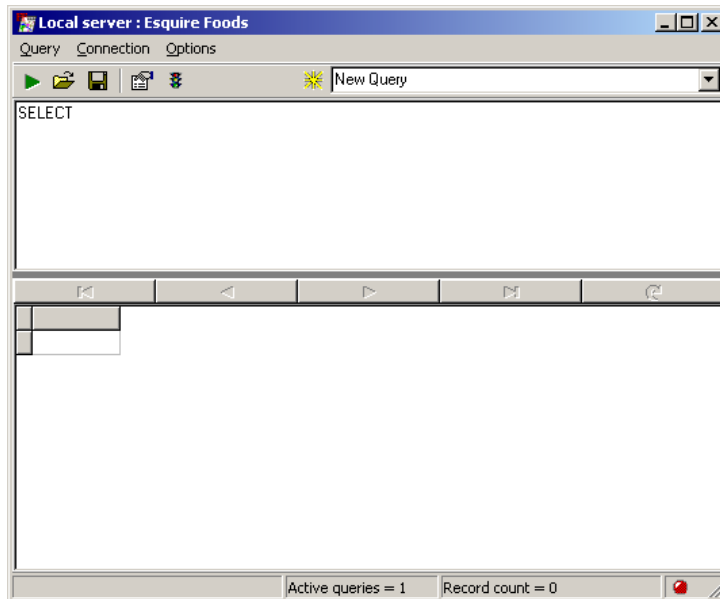


Figure 4.3: The SQL window for testing queries.

At the top of the SQL dialog box is a toolbar. The “Execute” button executes the SQL statement you type in the SQL Input window. The “Stop” button cancels the SQL statement that is currently executing. The “Save” button saves your current query to a file for later retrieval. The “Load” button retrieves a saved query file.

The SQL dialog box allows you to manage and execute multiple queries. Each query is executed within a separate connection to the server. The “New” button is used to start a new query within the dialog box. The combo box to the right is used to switch between open queries. The Properties button allows you to configure options for the SQL window and the current connection. The Options dialog box is shown in Figure 4.4.

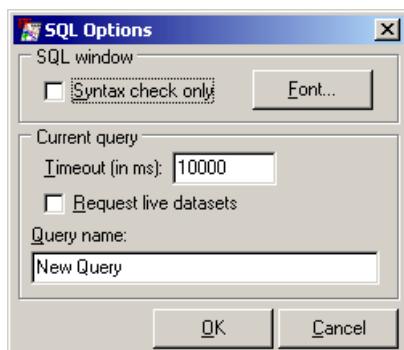


Figure 4.4: The SQL Options dialog box.

Below the Toolbar is the SQL Input window. This is where you type the SQL statement to be executed when you press the “Execute” button.

The grid at the bottom of the page displays the SQL result set. You can navigate the result set using the buttons at the bottom of the Result Set window. The navigational buttons are described in Table 4.3: “The Table Browser navigator buttons.” on page 73.

The Status Bar along the bottom of the dialog box displays the status of the current query, number of active connections, record count of the resulting set, and whether the current result set is editable.

## Refresh

Refreshes the list of databases for this server. This is important only if other users on the network have added new databases (or deleted them) since you started FlashFiler Explorer.

## New table...

Creates a new FlashFile table. Figure 4.5 shows the Define New Table dialog box.

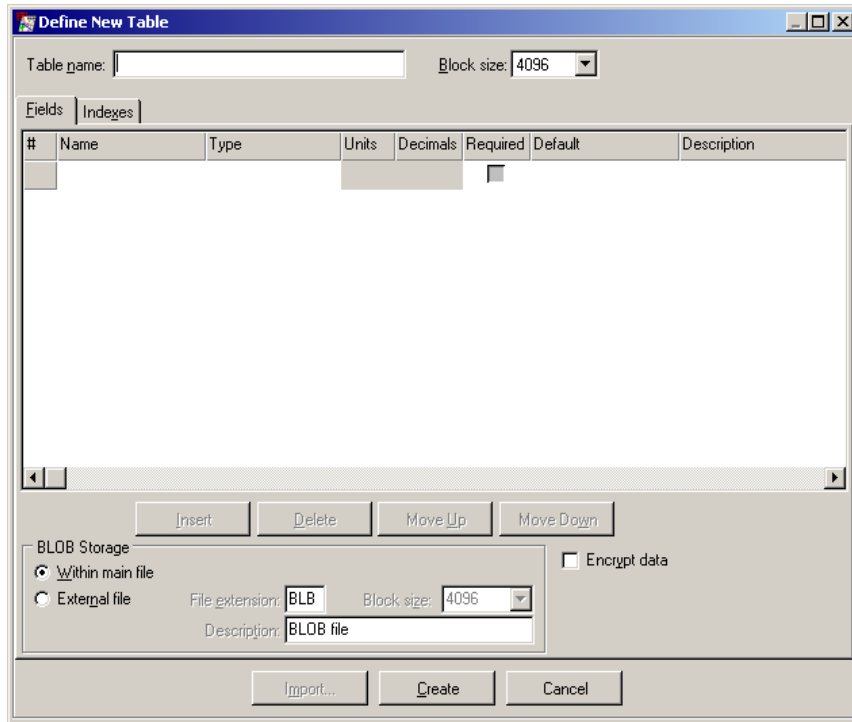


Figure 4.5: The Define New Table dialog box.

To create a new table, enter the name of the table in the “Table name” edit box. The table name must be unique within a database. Table names can contain the letters A - Z (upper- or lower-case), the digits 0 - 9, or underscores (\_). Table names can be no more than 31 characters.

After naming the table, select a block size for the main data file from the “Block size” combo box. See “Blocks” on page 20 for more information on block sizes.

### Defining fields

Click on the “Fields” tab to define the fields in the table.

“Name” is the name by which you reference the field in client applications. A field name can be up to 31 characters long and can contain any characters.

Use “Data Type” to select the FlashFiler data type of the field. Data types are described in the “FlashFiler Data Types” section of the Appendix.


The meaning of “Units” differs among data types. For string types, “Units” is the maximum number of base string characters the field can contain. For example, for a ShortString field of up to 20 characters, enter 20 for “Units”. The field actually occupies 21 bytes in the record because 1 byte is needed to store the actual length of the string. Similarly, for a WideString field of up to 20 wide characters, enter 20 for “Units”. In this case, the field actually occupies 42 bytes in the record, two bytes for each wide character and two bytes for the null terminator.

For numeric data types, “Units” indicates the precision of the field. It is the total number of digits to display (including left and right of the decimal point, if any). It does not affect the storage space of the field. Units for numeric data types are not currently used in the VCL or BDE interfaces, so this field can be ignored.

For real number data types (e.g., single, double), “Dec Pl” is the number of digits that should be displayed to the right of the decimal point. Again, this is not used and can be ignored.

Check the “Req” box if the field is required to contain a value (i.e., it cannot contain a null value). Note that all fields with a default value (see next paragraph) assigned will contain a value.

“Default” is used to apply a default value to this field as each record is inserted. For instance, a real estate agent may want to use a standard commission rate of 6 percent. Enter .06 as the default for the “CommissionRate” field and anytime they insert a record without assigning a value to the “CommissionRate” field “.06” will automatically be entered into this field. If they need to use a different commission rate, they enter it and the default value is ignored.

 **Caution:** Using a default value for fields in a unique index may cause key violations. A key violation is raised when an attempt is made to insert a record with non-unique fields for a unique index. Default values are best used with non-unique fields.

“Description” is used to store a description of the field, but only FlashFiler Explorer will display the field. The description can be up to 63 characters.

To insert a new field, select a field in the field list and click on the “Insert” button. The new field is inserted before the selected field.

To delete a field, select it and click on the “Delete” button.

Use the “Move Up” and “Move Down” buttons to move the selected field up or down in the list.

## Storing BLOB fields

If you define BLOB fields in the table, use the “BLOB Storage” group to specify how the BLOB data is stored. By default, all BLOB data is stored in the main data file, along with all of the other data for the record. However, you can specify an external file to contain the data for all BLOB fields by selecting “External File”. If you store BLOBs externally, you must supply a file extension and block size for the BLOB file. You can also enter a description for the file.

See “Binary Large Object (BLOB)” on page 19 for more information on BLOBs.

## Defining indexes

Click on the “Indexes” tab (shown in Figure 4.6) to define the indexes for the table.

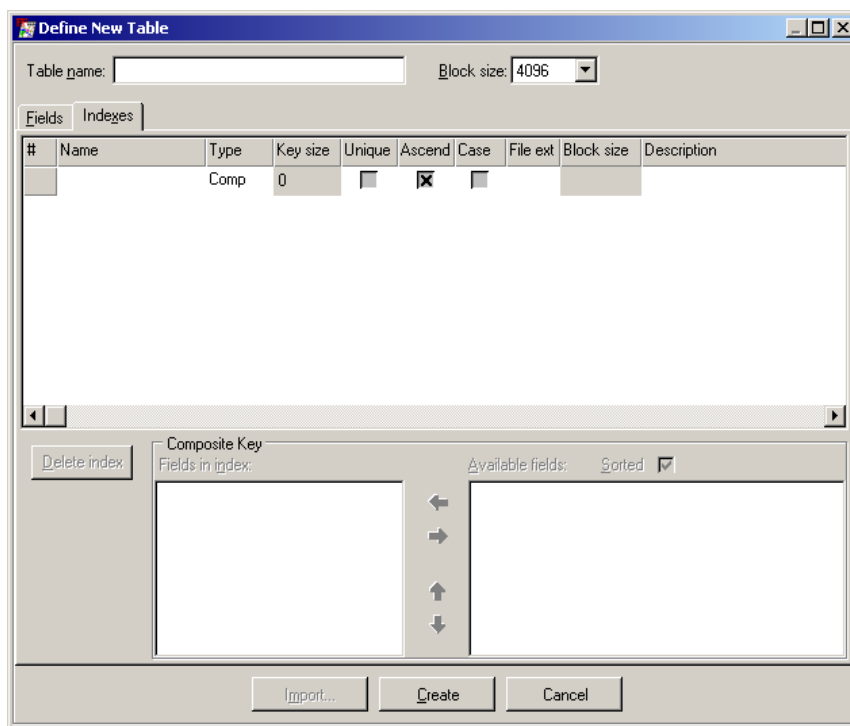


Figure 4.6: The Indexes tab of the Define New Table dialog box.

You must supply a “Name” to uniquely identify each index in the table. Index names can be up to 31 characters long and can contain any characters.


“Type” is used to indicate the type of index: composite key index or user-defined index.

Composite indexes simply define a sequence for the record based on the sort order of the values in one or more fields in the record. The actual fields that comprise the composite index are defined in the “Composite Key” group, which is described in “Composite indexes” on page 65. To delete an index, select it and click on the “Delete Index” button.

User-defined indexes allow you to programmatically decide how the index is computed. You write DLL functions that compute the key value for a given record and compare two key values. The DLL is linked into the FlashFiler Server and is not defined through FlashFiler Explorer. See “Config | User-Defined Indexes...” on page 43 for more information about user-defined indexes.

For composite key indexes, the key length is automatically computed and displayed in “Key Len”. For user-defined indexes, you must provide the number of bytes, because it is dependent on how you define the index.

Check the “Unique” check box if all key values for this index are unique (no duplicate values allowed in the index fields).

 **Caution:** Using a default value for fields in a unique index may cause key violations. A key violation is raised when an attempt is made to insert a record with non-unique fields for a unique index. Default values are best used with non-unique fields.

Check the “Ascend” check box if the sort order for the index is ascending (key values ordered from smallest to largest). Leave it unchecked for a descending index. Note that when you include multiple fields in your index, all fields must be either ascending or descending. It is not possible to have some fields ascending and other fields descending in the same index using FlashFiler’s standard indexing. You will have to implement a user-defined index to implement this requirement. See “User-Defined Indexes” on page 543 for more information.

By default, indexes are stored in the main data file. If you want the index data stored in an external file, you must enter a file extension in the “Ext” field. The filename is the same as the base table name. The file extension must be unique among all files comprising the table (e.g., main data file, other indexes, BLOB file).

If indexes are stored in an external file, specify the block size of the file in the “Block size” field. This is the smallest block of the index file that the server reads and stores in memory at one time.

You can optionally enter up to 63 characters of text in the “Description” field to describe the index. This information is displayed only by FlashFiler Explorer.

## Composite indexes

The “Composite Key” group defines the fields that comprise a composite index. These controls affect the currently selected index. This group is disabled if the index name has not been filled in or it is not a composite key index.

Move the desired fields from the “Available Fields” list to the “Fields in Index” list. You can do this by selecting the fields and clicking “Add”, or by dragging and dropping the fields between the two lists. To change the relative position of the fields within the index, select a field in the “Fields in Index” list and move it up or down using the arrow buttons.

## Import...

Imports data from an external file into a FlashFiler table. The Import Data dialog box is displayed as shown in Figure 4.7.

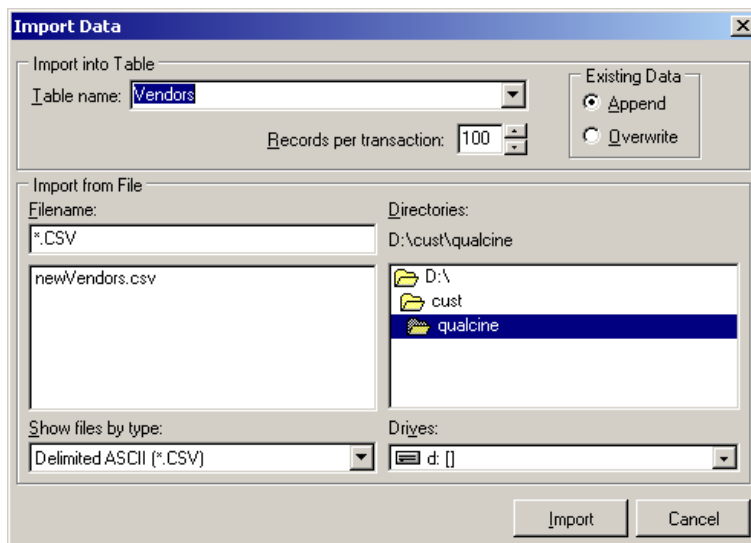


Figure 4.7: The Import Data dialog box.



The requirements for data files to be imported into FlashFiler are outlined in “Importing Data into FlashFiler Tables” on page 534.

In the “Table Name” combo box, select the FlashFiler table to import into, or enter a new table name to create a table. Select “Append” if you want any existing data to be preserved. Select “Overwrite” if you want to destroy existing data. Use the “Import from File” group to supply the name and path of the file to import. Set “Records Per Transaction” to determine how often transactions are committed. See “Transaction” on page 27 for more information about transactions.

Click the “Import” button when you’re ready to load the data into the FlashFiler table.

# The Table Context Menu

In the FlashFile Explorer main dialog, select a table and then right click to display the table context menu.

## View definition...

Displays the Table Definition dialog box, which shows the field layout and indexes for the selected table, as illustrated in Figure 4.8.

4

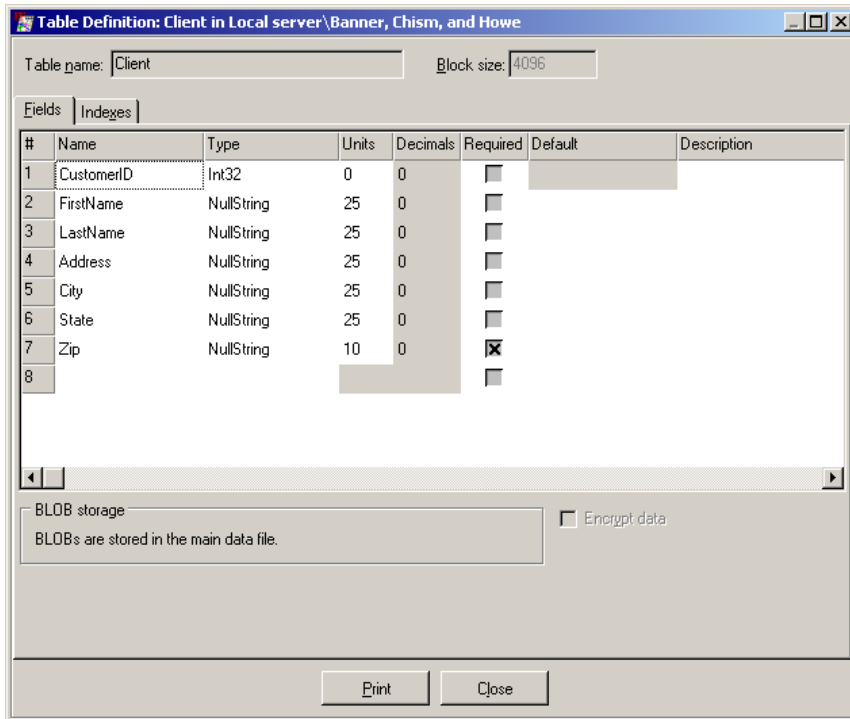


Figure 4.8: The Table Definition dialog box.

The information shown here is similar to the “The Define New Table dialog box.” on page 61. The difference is that you cannot modify the table’s structure from this dialog.

The “Print” button prints the table definition (showing the definitions for all fields and indexes).

## View indexes...

Displays the Table Definition dialog box, as illustrated by Figure 4.9, with the table's indexes shown first.

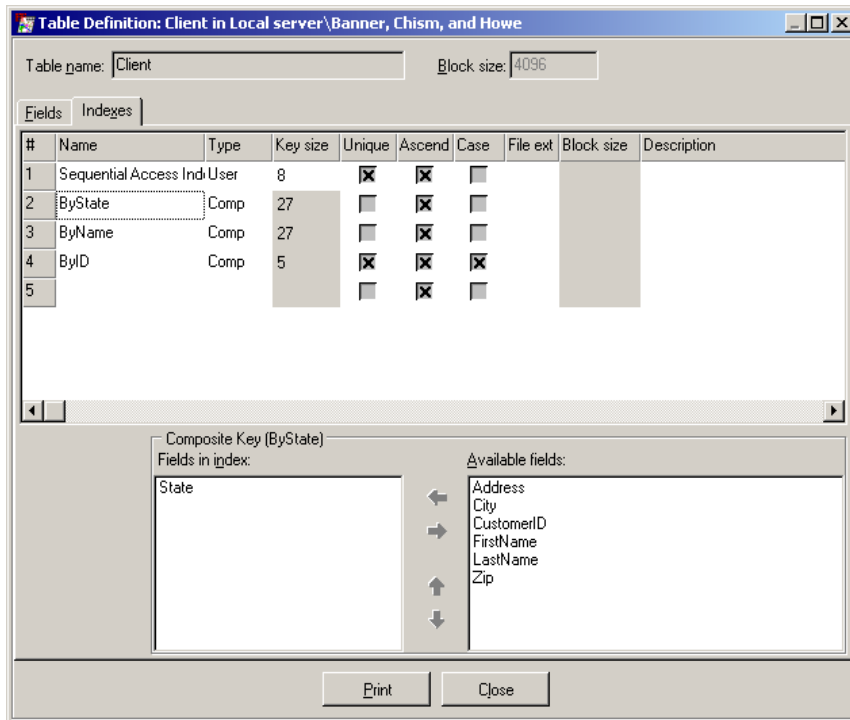


Figure 4.9: The Indexes tab of the Table Definition dialog box.

The information shown here is similar to the “The Define New Table dialog box.” on page 61. The difference is that you cannot modify the table's structure from this dialog.

The “Print” button prints the table definition (showing the definitions for all fields and indexes).

## New table...

Allows you to define the field layout and indexes for a new FlashFiler table. The “Define New Table dialog box” (see page 61) is displayed.

## Delete

Removes the selected table from the database. All data files associated with the table are physically deleted.

## Rename

Allows you to rename the selected table.

## Pack

Starts a table pack operation, which removes all deleted records from the table. If the table is large, this can be a lengthy process.

## Reindex...

Starts a table reindex operation, which rebuilds the keys for a selected index in the table. If the table is large, this can be a lengthy process.

## Redefine...

Allows you to change the field layout and indexes for an existing table. The Redefine Table dialog box, shown in Figure 4.10, is displayed. You can change any aspect of an existing table's structure, but typical uses are to add a new field, remove an existing field, or change the name or data type of a field.

The "Fields" and "Indexes" tabs work just like the tabs in the "Define New Table dialog box" shown on page 61. The only difference is that you have an existing structure to work with. You can add, delete, or change any of the information.

The "Existing Data" tab allows you to determine how existing data is translated into the new structure. The simplest approach is to abandon the existing data. If the "Preserve Existing Data" check box is not checked, the table is redefined and all existing data is irretrievably lost.

To preserve part or all of the existing data, you must provide a field map. The field map links a field in the new structure with a field in the original structure. The data in the old field is copied into the new field when you click on the "Restructure" button. The field map lists all of the new fields and their data types on the left side of the grid. For each of these fields, you must select the corresponding field from the original structure in the "Old Field" column by selecting a field from the combo box. Only old fields that are assignment compatible with

the new field and that have not already been mapped to a new field are available in the “Old Field” combo box. For example, you cannot map an old datetime field to a new Boolean field. See “Converting Data Types” on page 531 for a list of the legal data type conversions.

4

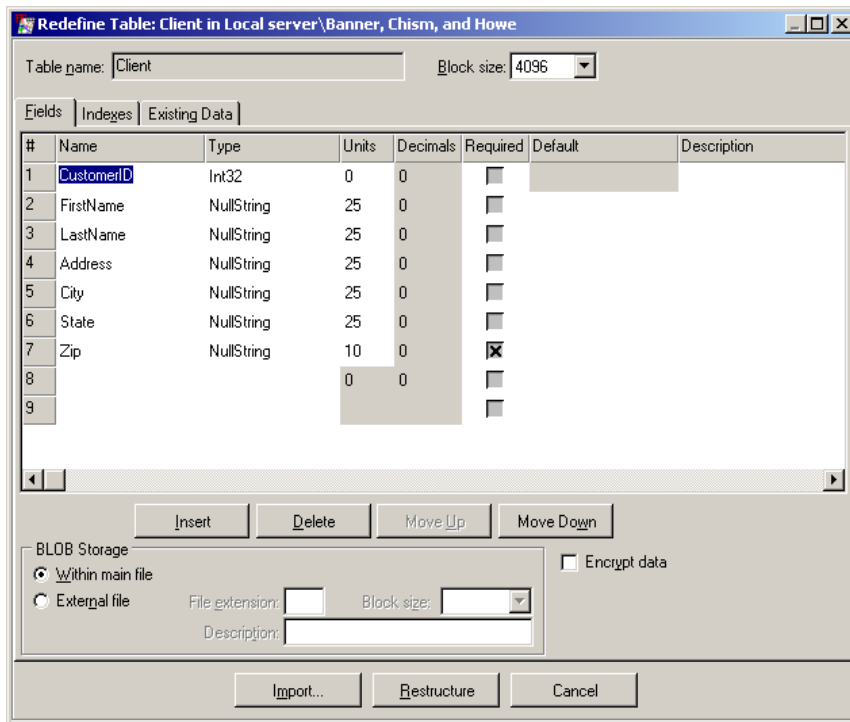


Figure 4.10: The Redefine Table dialog box.

The “Orphaned Fields” tab displays a list of all the original fields that are not mapped to a new field. If any orphaned fields remain when you click the “Restructure” button, the data in those fields is lost. Similarly, if any new fields are not mapped to an old field, the new fields are filled with a null value (or default value if assigned) in the restructured table.

Use the “Match By Name” button to automatically map old fields to new fields with matching field names (assuming they are of compatible data types).

Use the “Match By Position” button to automatically map old fields to new fields occupying the same ordinal position in the record (assuming they are of compatible data types). This means that the first old field is mapped to the first new field, etc.

The “Clear Assignments” button erases all old field assignments in the field map.

## Import...

Imports data from an external file into a FlashFiler table. The “Import Data” dialog (described on page 65) is displayed.

## Set AutoInc...

Assigns a specific seed to the AutoInc value. Normally, AutoInc values begin at 1 and increment by 1 as they are used. This option allows you to set the AutoInc to a value of your choosing. The AutoInc field of the next record inserted will have the value you set plus 1. Existing AutoInc values are not altered.

## Empty

Deletes all records from the table, but retains the table's structure.

# The Table Browser

When a table object is double clicked on the Explorer's main screen, a Table Browser displays the table's data as shown in Figure 4.11.

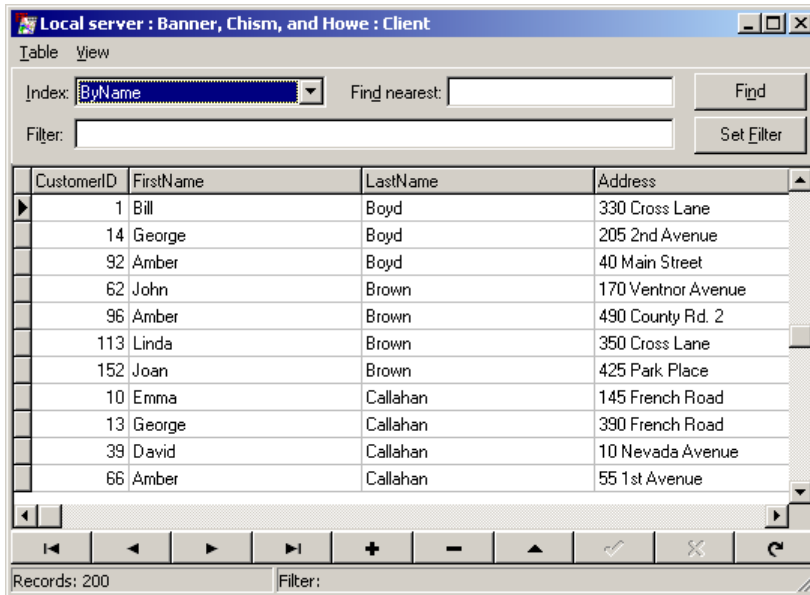


Figure 4.11: The Table Browser window.

## The Table Browser's menu

The Table Browser menu offers a few options for displaying data in the window.

### Table | Close

Closes the Table Browser window.

### View | Show Filter

If this option is checked, you are presented with an edit box that allows you to enter a filter and a "Set Filter" button to apply the filter. The edit box and button are not displayed if this option is not checked.

### View | Show Record Count

If this option is checked, the record count for the current record set is displayed in the status bar across the bottom of the Table Browser's window. Disabling (unchecking) this option may speed up browsing of large tables.

## View | Refresh

Updates the Table Browser's data. This is only important if other FlashFiler clients are modifying the selected table.

## The Table Browser window

Use the “Index” list box to sort the data by a particular field or fields as defined by the chosen index. If you choose an index other than the sequential index, you can search the fields of the index using the “Find Nearest” edit box and the button to its right. See “FindNearest method” on page 239 for more information about using “Find Nearest”.





If “Show Filter” is checked (see View | Show Filter in the previous section), you can also limit the visible records by entering a filter and pressing the “Set Filter” button. While the filter is active, the “Set Filter” button becomes a “Clear Filter” button. Press “Clear Filter” to remove the filter and view all records in the table. See “TffBaseTable Class” on page 232 for more information on using filters.

Use the scroll bars to move left, right, up, and down through all of the visible rows and columns of the table. Note that the horizontal scroll bars are only available if the table's columns are wider than the grid.

If you want to change the table data, ensure Options | Live Datasets (see page 75) is checked on the main menu. Live Datasets allow you to edit, insert, and delete data in the table.






The navigator buttons at the bottom of the table browser help you navigate through the table. The first four buttons are always available. The last five buttons are only available if Live Datasets is enabled on the main screen options. They perform the functions described in Table 4.3.

**Table 4.3:** *The Table Browser navigator buttons.*

	Move to the first record in the table.
	Move to the previous record.
	Move to the next record.
	Move to the last record in the table.



**Table 4.3:** *The Table Browser navigator buttons. (continued)*

	Insert a new record. You can enter data in each column and then either click the button with the check mark on it or select another row to save the changes. Click the button with the 'x' on it or press <Esc> to cancel the changes and discard the new record.
	Delete the current record.
	Edit the current record.
	Save the changes made to the current row.
	Cancel the changes made to the current row (the original values are restored).

The Status Bar at the bottom of the Table Browser's window shows you the record count for the current record set and the current filter.

---

# The Main Menu

Nearly all the operations in FlashFiler Explorer are accessed via the context menus discussed previously. However, the main menu in the FlashFiler Explorer main dialog box provides simple functions and configuration options.

## Server | Refresh

Refreshes the list of active FlashFiler Servers. All existing connections are dropped and FlashFiler Explorer searches the network for all active servers.

## Server | Register...

Allows you to enter a server name in the list of registered servers. See “Register...” in the Server Context Menu on page 57 for a description of the list of registered servers.

## Server | Exit

Terminates FlashFiler Explorer.

## Options | Live Datasets

When this option is checked, the data shown in the table browser can be modified.

## Options | Print Setup...

Displays the “Print Setup” dialog so you can choose a printer and print options.

## Options | Transport

FlashFiler can operate in single-user mode and across a network for multi-user access. In single-user mode, your client applications and FlashFiler Server operate on the same machine. In multi-user mode, FlashFiler Server can be accessed from the same machine and remotely across a network on a different machine than the client applications.

You specify how FlashFiler Explorer communicates with FlashFiler Server by selecting the transport from the submenu. Select Single-User, Winsock IPX/SPX (network access), or Winsock TCP/IP (network access). FlashFiler Explorer finds and communicates only with FlashFiler Servers that have enabled the selected transport.

We recommend using Single User transport when FlashFiler Explorer is on the same machine as the server. If they are not on the same machine, we suggest using TCP/IP.

## Help | Help Topics . . .

Displays the FlashFiler Explorer on-line help.

## Help | About . . .

Displays a screen of information about FlashFiler Explorer.

## Help | Visit TurboPower's Web Site . . .

Starts your web browser with the address of the TurboPower Software Web site.

## Help | Send Mail to TurboPower

Initializes a new email message addressed to our mail support using your default email reader.

---

## Chapter 5: Base Classes

This chapter describes the base classes and components used to create the FlashFiler client and server components and classes. `TffObject` and `TffComponent` are simple classes whose sole purpose is to replace the use of the VCL's memory manager with FlashFiler's custom memory manager. `TffLoggableComponent` allows a component to log informational messages, warnings, and errors to an instance of `TffBaseLog`. `TffStateComponent` implements a state engine used by transports and server-side components. `TffBaseLog` defines the interface for a logging mechanism. `TffEventLog` implements the logging of messages to an operating system file. `TffStringGrid` is a customized `TStringGrid` used by FlashFiler Explorer.

---

# TffComponent Class

The TffComponent class overrides the memory allocation performed by TComponent and serves as an ancestor to all FlashFiler components. Due to issues with VCL's heap manager, FlashFiler Server uses its own memory management scheme to allocate memory for components. This provides greater reliability for database applications requiring uninterrupted availability.

## Hierarchy

TComponent (VCL)

    TffComponent (FLLBASE)

## Properties

    Version

## Methods

    FreeInstance

    NewInstance

## Reference Section

### FreeInstance

method

```
procedure FreeInstance; override;
```

- ↪ Deallocates memory allocated by a previous call to the NewInstance method.

This method deallocates memory obtained for an instance of this type. Memory deallocated by this method is returned to the FlashFiler memory pool from which it was obtained. You should never need to call FreeInstance directly.

### NewInstance

method

```
class function NewInstance : TObject; override;
```

- ↪ Allocates memory for each instance of TffComponent.

This method obtains memory from a FlashFiler memory pool for a new instance of this type. The memory is freed by the FreeInstance method. You should never need to call NewInstance directly.

### Version

property

```
property Version : string
```

- ↪ Returns the version number of FlashFiler.

Version is provided so you can identify your FlashFiler version if you need technical support. You can display the FlashFiler about box by double-clicking this property or selecting the dialog button to the right of the property value.

# TffLoggableComponent Class

The TffLoggableComponent class aids in the logging of informational messages, warnings, and errors to a centralized logging facility. To centralize logging, connect each component inheriting from TffLoggableComponent to an instance of TffBaseLog. All log messages sent via the LcLog method are routed to the same TffBaseLog. To enable or disable logging on individual components, use the EventLogEnabled property. To enable or disable logging for all components, use the Enabled property of the TffBaseLog.

## Hierarchy

TComponent (VCL)	
❶ TffComponent (FFLLBASE) .....	78
TffLoggableComponent (FFLLCOMP)	

## Properties

EventLog	EventLogEnabled
----------	-----------------

## Methods

❶ FreeInstance	LcLog	❶ NewInstance
----------------	-------	---------------

## Reference Section

### **EventLog** **property**

---

`property EventLog : TffBaseLog`

↳ Identifies the event log to which error messages are written.

You may leave this property set to nil, in which case no messages are written.

See also: EventLogEnabled, LcLog

### **EventLogEnabled** **property**

---

`property EventLogEnabled : Boolean`

Default: False

↳ Identifies whether the component will write to its EventLog.

Messages are written to the EventLog if the EventLog property is set to an instance of TffBaseLog and if EventLogEnabled is set to True.

To disable logging, set EventLogEnabled to False, the default value.

See also: EventLog, LcLog

### **LcLog** **virtual method**

---

`procedure LcLog(const aString : string); virtual;`

↳ Writes the specified string to the component's event log.

If a TffBaseLog component is attached to the component via the component's EventLog property and the EventLogEnabled property is set to True, this method writes the specified string to the attached event log. Use this method to record errors, informational messages, and warnings.

See also: EventLog, EventLogEnabled



# TffStateComponent Class

The TffStateComponent class implements a state engine, as shown in Figure 5.1.

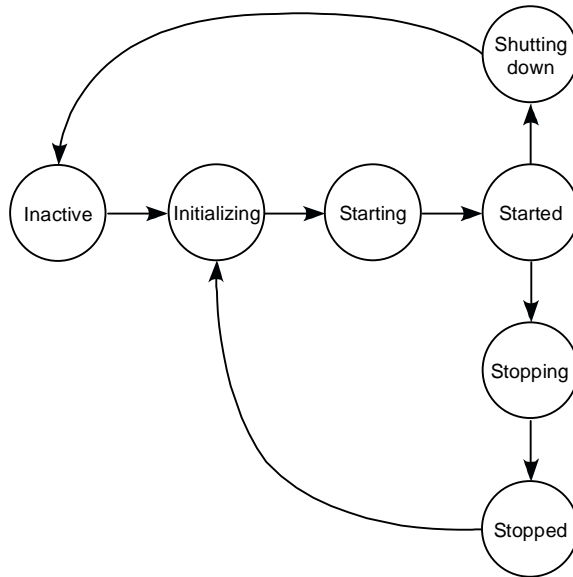


Figure 5.1: The state engine diagram.

The circles represent states. Lines between the circles represent the path taken by the component's internal state engine as it goes from a start state to a destination state. For example, when a component is first created its State is set to `ffesInactive`. Setting the State to `ffesStarted` causes the state engine to change the state from `ffesInactive` to `ffesInitializing`, then to `ffesStarting`, and finally to `ffesStarted`.

As a `TffStateComponent` progresses from one state to the next, the state engine calls protected methods within the component. By overriding these methods, you can perform initialization, startup, and shutdown actions required by your component. Table 5.1 describes each state.

**Table 5.1:** *The `TffStateComponent` status values.*

State	Meaning
<code>ffesInactive</code>	The component is not doing any work. If the component reaches this state from state <code>ffesShuttingDown</code> then the state engine calls the <code>ScShutdown</code> method.
<code>ffesInitializing</code>	The component is being prepared for work. The state engine calls the <code>ScInitialize</code> method.
<code>ffesStarting</code>	The component is starting. The state engine calls the <code>ScStartup</code> method.
<code>ffesStarted</code>	The component has successfully started and is ready for work.
<code>ffesShuttingDown</code>	The component has been told to shutdown. The state engine calls the <code>ScPrepareForShutdown</code> method.
<code>ffesStopping</code>	The component is stopping. This is similar in meaning to <code>ffesShuttingDown</code> . As described in Chapter 11, this state is used to stop a <code>TffBaseServerEngine</code> without shutting down its associated command handlers and transports.
<code>ffesStopped</code>	The component has stopped. As described in Chapter 11, a <code>TffBaseServerEngine</code> reaches this state when the engine manager desires to shut down the engine without shutting down its associated command handlers and transports.
<code>ffesUnsupported</code>	The component is not supported on this workstation. This may be due to incorrect or missing drivers, packages, etc.
<code>ffesFailed</code>	An error occurred during startup of the component. The component may not be used until the application is restarted.

# Hierarchy

TComponent (VCL)

- ❶ TffComponent (FFLLBASE) ..... 78
  - ❷ TffLoggableComponent (FFLLCOMP) ..... 80
    - TffStateComponent (FFLLCOMP)

## Properties

❷	EventLog	❷	EventLogEnabled	State
---	----------	---	-----------------	-------

## Methods

❶	FreeInstance	ScPrepareForShutdown	Startup
❷	LcLog	ScShutdown	Stop
❶	NewInstance	ScStartup	
	ScInitialize	Shutdown	

## Events

OnStateChange

## Reference Section

### OnStateChange

event

```
property OnStateChange : TNotifyEvent
```

↳ Defines an event handler called when the state of the component changes.

Use this event to receive notification when the component's State property has changed. At the time this event handler is called, the component's State has already been set to the new value.

See also: State

### ScInitialize

virtual abstract method

```
procedure ScInitialize; virtual; abstract;
```

↳ Initialize internal variables or perform actions when the component's State is set to `ffesInitializing`.

This method is called when the component's State is changed to `ffesInitializing`. Use this method to initialize variables or perform any other actions necessary to prepare the component for operation.

See also: `ScPrepareForShutdown`, `ScStartup`, `ScShutdown`, State

### ScPrepareForShutdown

virtual abstract method

```
procedure ScPrepareForShutdown; virtual; abstract;
```

↳ Prepare the component for shutdown.

This method is called when the component's State is set to `ffesShuttingDown`. Use this method to perform any actions required to prepare the component for shutdown.

See also: `ScInitialize`, `ScShutdown`, `ScStartup`

### ScShutdown

virtual abstract method

```
procedure ScShutdown; virtual; abstract;
```

↳ Shut down the component.

This method is called when the component's State is set to `ffesInactive`. Use this method to perform any final actions required to shut down the component. For example, use this method to free any memory allocated in `ScInitialize`.

See also: `ScInitialize`, `ScPrepareForShutdown`, `ScStartup`

## ScStartup

virtual abstract method

```
procedure ScStartup; virtual; abstract;
```

↳ Start the component.

This method is called when the component's state is set to `ffesStarting`. It is assumed that `ScInitialize` has been called by the time this method is called. Use this method to perform any final actions required for the component to start operation.

See also: `ScInitialize`, `ScPrepareForShutdown`, `ScShutdown`

5

## Shutdown

virtual method

```
procedure Shutdown; virtual;
```

↳ Sets the component's state to `ffesInactive`.

See also: `Startup`, `State`, `Stop`

## Startup

virtual method

```
procedure Startup; virtual;
```

↳ Sets the component's state to `ffesStarted`.

See also: `Shutdown`, `State`, `Stop`

## Stop

virtual method

```
procedure Stop; virtual;
```

↳ Sets the component's state to `ffesStopped`.

See also: `Shutdown`, `Startup`, `State`

```
property State : TffState
```

```
TffState = (ffesInactive, ffesInitializing, ffesStarting,  
    ffesStarted, ffesShuttingDown, ffesStopping, ffesStopped,  
    ffesUnsupported, ffesFailed, ffesUnsupported, ffesFailed);
```

Default: ffesInactive

↳ Identifies the current state of the component.

A component's State indicates its readiness for work. The meaning of each state is described in the introduction to this class on page 82.

See also: ScInitialize, ScPrepareForShutdown, ScShutdown, ScStartup

# TffBaseLog Class

The TffBaseLog class defines the basic interface for a component that writes informational messages, warnings, and errors to some kind of event log. Components inheriting from TffBaseLog may implement the event log using a file, the Windows event log, or any other mechanism. Components inheriting from TffBaseLog must be thread-safe.

To enable or disable logging, use the Enabled property. To specify a file name to which the messages are written, use the Filename property.

Classes inheriting from TffBaseLog must implement the WriteBlock, WriteString, WriteStringFmt, and WriteStrings methods.

## Hierarchy

TComponent (VCL)	
❶ TffComponent (FFLLBASE) .....	78
TffBaseLog (FFLLLOG)	

## Properties

Enabled	Filename
---------	----------

## Methods

❶ FreeInstance	WriteBlock	WriteStringFmt
❶ NewInstance	WriteString	WriteStrings

## Reference Section

### Enabled

property

```
property Enabled : Boolean
```

Default: False

↳ Indicates whether messages are logged.

Set Enabled to True to enable logging or False to disable logging.

### Filename

property

```
property FileName : TFileName
```

Default: Empty string

↳ Identifies the file to which messages are written.

This property is applicable to descendant classes that write messages to the file system. In other situations, this property may be ignored or its meaning re-interpreted.

### WriteBlock

virtual abstract method

```
procedure WriteBlock(const S : string; Buf : pointer;  
  BufLen : TffMemSize); virtual; abstract;
```

↳ Writes a message and block of data to the log.

This method is suited for logging binary data. It writes the message S and the data referenced by Buf to the log. BufLen identifies the length of the data referenced by Buf. In a multithreaded situation, S and Buf should be written consecutively, without interruption, to the log.

See also: WriteString, WriteStringFmt, WriteStrings

### WriteString

virtual abstract method

```
procedure WriteString(const aMsg : string); virtual; abstract;
```

↳ Writes aMsg to the log.

This method is suited for logging individual messages, writing aMsg to the log.

See also: WriteBlock, WriteStringFmt, WriteStrings



## WriteStringFmt

virtual abstract method

```
procedure WriteStringFmt(  
    const aMsg : string; args : array of const); virtual; abstract;
```

↳ Writes a formatted message to the log.

This method uses the VCL Format function to combine aMsg and args, writing the resultant string to the log.

See also: WriteBlock, WriteString, WriteStrings

## WriteStrings

virtual abstract method

```
procedure WriteStrings(  
    const Msgs : array of string); virtual; abstract;
```

↳ Writes an array of strings to the log.

In a multithreaded situation, all strings should be written consecutively, without interruption, to the log.

See also: WriteBlock, WriteString, WriteStringFmt



# TffEventLog Component

The TffEventLog component writes log messages to an operating system file. This component is thread-safe. Each write method uses a critical section to control access to the log. This allows the WriteBlock and WriteStrings methods to write their strings to the log without interruption. When you read the log, you will not see any intermixing of messages from different threads.

## Hierarchy

TComponent (VCL)

❶ TffComponent (FFLLBASE) .....	78
❷ TffBaseLog (FFLLLOG) .....	88
TffEventLog	

## Properties

❷ Enabled	❷ Filename
-----------	------------

## Methods

❶ FreeInstance	WriteString
❶ NewInstance	WriteStringFmt
WriteBlock	WriteStrings

## Reference Section

### WriteBlock

method

```
procedure WriteBlock(  
    const S : string; Buf : pointer; BufLen : TffMemSize); override;
```

↳ Writes a message and block of data to the log.

This method is suited for logging binary data. It writes the message *S* followed by *BufLen* to the log. It then writes the first 1,024 bytes of *Buf* to the log. Up to 16 bytes are written per line, in hex format followed by ASCII format. As an example, the following source code writes the contents of a data dictionary stream to the log:

```
FEventLog.WriteBlockv(  
    'Dictionary', Stream.Memory, Stream.Size);
```

The resulting log entries look like the following:

```
Dictionary (Size: 438)  
08 02 01 06 12 44 61 74 61 2f 44 61 74 61 44 69 [.....Data/DataDi]  
63 74 20 46 69 6c 65 06 03 46 46 44 04 00 00 01 [ct File.FF2....]  
00 02 00 02 06 06 02 49 44 06 00 02 0a 02 00 02 [.....ID.....]  
00 02 04 02 09 08 08 06 09 46 69 72 73 74 4d 61 [.....FirstNa]  
6e 65 06 00 02 19 02 00 02 04 02 1a 02 2d 09 08 [me.....]  
06 08 4c 61 73 74 4d 61 6e 65 06 00 02 19 02 00 [...LastName.....]
```

In a multithreaded situation, *S* and *Buf* are written consecutively, without interruption, to the log.

See also: `WriteString`, `WriteStringFmt`, `WriteStrings`

### WriteString

method

```
procedure WriteString(const aMsg : string); override;
```

↳ Writes *aMsg* to the log.

This method is suited for logging individual messages. *aMsg* is the string that is to be written to the log. The following source code illustrates the use of this method:

```
FEventLog.WriteString('Starting server');
```

See also: `WriteBlock`, `WriteStringFmt`, `WriteStrings`

```
procedure WriteStringFmt(  
    const aMsg : string; args : array of const); override;
```

↳ Writes a formatted message to the log.

This method uses the VCL Format function to combine aMsg and args, writing the resultant string to the log. The following source code illustrates the use of this method:

```
on E:EffDatabaseError do begin  
    FEventLog.WriteStringFmt(  
        'Exception raised (%d): %s', [E.ErrorCode, E.Message]);  
end;
```

See also: WriteBlock, WriteString, WriteStrings

```
procedure WriteStrings(const Msgs : array of string); override;
```

↳ Writes an array of strings to the log.

This method writes all strings to the log consecutively and without interruption, even in a multithreaded situation. The following source code illustrates the use of this method:

```
if FEventLog.Enabled then  
    FEventLog.WriteStrings  
        ([ 'OpenTable',  
          format(' ClientID %d', [dmClientID]),  
          format(' DBase ID %d', [DatabaseID]),  
          format(' TblName [%s]', [TableName]),  
          format(' InxName [%s]', [IndexName]),  
          format(' InxNum %d', [IndexNumber]),  
          format(' OpenMode %d', [byte(OpenMode)]),  
          format(' ShrMode %d', [byte(ShareMode)]) ]);
```

See also: WriteBlock, WriteString, WriteStringFmt

---

# TffObject Class

The TffObject class overrides the memory allocation performed by TObject and serves as an ancestor to FlashFiler classes that are not components. Due to issues with VCL's heap manager, FlashFiler Server uses its own memory management scheme to allocate memory for components. This provides greater reliability for database applications requiring uninterrupted availability.

## Hierarchy

TObject (VCL)

    TffObject (FFLLBASE)

## Methods

FreeInstance

NewInstance

## Reference Section

### FreeInstance

method

```
procedure FreeInstance; override;
```

↪ Deallocates memory allocated by a previous call to the NewInstance method.

This method deallocates memory obtained for an instance of this type. Memory deallocated by this method is returned to the FlashFiler memory pool from which it was obtained. You should never need to call FreeInstance directly.

### NewInstance

method

```
class function NewInstance : TObject; override;
```

↪ Allocates memory for each instance of TffObject.

This method obtains memory from a FlashFiler memory pool for a new instance of this type. The memory is freed by the FreeInstance method. You should never need to call NewInstance directly.



## TffStringGrid Component

The TffStringGrid component is used within several forms of the FlashFiler Server. It inherits from TStringGrid, containing methods and events simplifying the use of the string grid.

### Hierarchy

TStringGrid (VCL)

TffStringGrid (FFLLGRID)

### Methods

AnyCellsIsEmpty

BeginUpdate

BlankRow

CopyRow

EndUpdate

LastRowIsEmpty

RestoreToRow

RowIsEmpty

RowIsFilled

SaveRow

### Events

OnEnterCell

OnExitCell

OnSortColumn

## Reference Section

### AnyCellsEmpty

method

```
function AnyCellsEmpty(const aRow : integer) : Boolean;
```

↳ Returns True if any cell in the specified row is empty.

Use this method to determine if there are any empty cells within a specific row of the grid. An empty cell is defined as having no content (i.e., the Cells method returns an empty string). aRow is the row to check with row 0 being the first row of the grid. This function returns True if there is at least one empty cell within the row otherwise it returns False.

See also: LastRowIsEmpty, RowIsEmpty, RowIsFilled

### BeginUpdate

method

```
procedure BeginUpdate;
```

↳ Disables redrawing of the grid.

When you modify the grid, the grid redraws itself. If you need to make a large number of modifications, the resulting flicker may waste time and have an unappealing look. Use this method to prevent the grid from redrawing itself while the code modifies the contents of the grid. When the code has finished modifying the grid, call the EndUpdate method. The following example illustrates this process:

```
aGrid.BeginUpdate;
try
  {Clear out the first cell of each row.}
  for anIndex := 0 to pred(aGrid.RowCount) do
    aGrid.Cells[0, anIndex] := '';
finally
  aGrid.EndUpdate;
end;
```

See also: EndUpdate

### BlankRow

method

```
procedure BlankRow(const aRow : integer);
```

↳ Empties each cell in the specified row.

Use this method to delete the contents of each cell in the row specified by parameter aRow. aRow is base zero. The contents of each cell are replaced with a blank string.



```
procedure CopyRow(const srcRow, destRow : integer);
```

↳ Copies all cells in one row to another row.

Use this method to copy the contents of each cell in the row specified by `srcRow` to the corresponding cell in the row specified by `destRow`. The cells being copied are untouched. The following example shows how to shift each row up in the grid by one line, effectively deleting the first row.

```
for anIndex := 1 to pred(aGrid.RowCount) do
  aGrid.CopyRow(anIndex, pred(anIndex));
{Remove the last row.}
aGrid.RowCount := aGrid.RowCount - 1;
```

**EndUpdate****method**

```
procedure EndUpdate;
```

↳ Enables redrawing of the grid.

When you modify the grid, the grid redraws itself. If you need to make a large number of modifications, the resulting flicker may waste time and have an unappealing look. Use the `BeginUpdate` method to disable redrawing of the grid. Once the code has finished making changes, use this method to enable redrawing of the grid. The following example illustrates this process:

```
aGrid.BeginUpdate;
try
  {Clear out the first cell of each row.}
  for anIndex := 0 to pred(aGrid.RowCount) do
    aGrid.Cells[0, anIndex] := '';
finally
  aGrid.EndUpdate;
end;
```

See also: `BeginUpdate`

**LastRowIsEmpty****method**

```
function LastRowIsEmpty : Boolean;
```

↳ Indicates whether each cell of the last row in the grid is empty.

This method returns `True` if each cell of the last row contains a blank string (e.g., `''`). If any cell contains something other than a blank string this method returns `False`.

See also: `AnyCellIsEmpty`, `RowIsEmpty`, `RowIsFilled`

**OnEnterCell****event**

```
property OnEnterCell : TffCellFocusEvent

TffCellFocusEvent = procedure(
    Sender : TffStringGrid; aCol, aRow : integer;
    const text : string) of object;
```

↳ Defines an event handler that is called when a cell gains focus.

Use this event to perform special actions when a cell gains focus. Sender is the TffStringGrid containing the cell. aCol is the cell's column and aRow is the cell's row. text is the contents of the cell at the time the cell gains focus.

See also: OnExitCell

**OnExitCell****event**

```
property OnExitCell : TffCellFocusEvent

TffCellFocusEvent = procedure(
    Sender : TffStringGrid; aCol, aRow : integer;
    const text : string) of object;
```

↳ Defines an event handler that is called when a cell loses focus.

Use this event to perform special actions when a cell loses focus. Sender is the TffStringGrid containing the cell. aCol is the cell's column and aRow is the cell's row. text is the contents of the cell at the time the cell loses focus.

See also: OnEnterCell

**OnSortColumn****event**

```
property OnSortColumn : TffColumnSortEvent

TffColumnSortEvent = procedure(
    Sender : TffStringGrid; aCol : integer) of object;
```

↳ Defines an event handler that is called when the mouse is clicked on the first fixed row.

Use this event to carry out the sorting of a specific column. This event is raised only if the grid has one or more fixed rows and the user clicks the right or left mouse button while positioned over the first row. Sender is the TffStringGrid in focus and aCol is the column over which the mouse is positioned.

## RestoreToRow

method

```
procedure RestoreToRow(const aRow : integer);
```

↳ Restores a previously saved row to the specified row of the grid.

If you saved a row of the grid using the `SaveRow` method, use this method to restore the saved row to a specific row within the same grid. `aRow` is base zero and is the row to which all cells of the saved row are written.

Note that only one row may be saved at any given time.

See also: `SaveRow`

5

## RowIsEmpty

method

```
function RowIsEmpty(const aRow : integer) : Boolean;
```

↳ Returns `True` if every cell within the specified row of the grid is empty.

Use this method to verify each cell within the specified cell contains a blank string. If any one cell contains at least one character this method returns `False`. Otherwise this method returns `True`.

See also: `AnyCellIsEmpty`, `LastRowIsEmpty`, `RowIsFilled`

## RowIsFilled

method

```
function RowIsFilled(const aRow : integer) : Boolean;
```

↳ Returns `True` if every cell within the specified row contains a value.

Use this method to verify each cell in a row contains at least one character. If any cell is empty this method returns `False`. Otherwise it returns `True`. `aRow` is base zero and is the row to be checked by this method.

See also: `AnyCellIsEmpty`, `LastRowIsEmpty`, `RowIsEmpty`

---

```
procedure SaveRow(const aRow : integer);
```

↳ Makes a copy in memory of the cells in the specified row.

Use this method to preserve the contents of a row. Use the `RestoreToRow` method to restore the contents to the same row or a different row within the same grid. `aRow` is base zero and is the row whose cell contents are saved.

**Note:** Only one row may be saved at any given time. If you call `SaveRow` twice, the first set of saved cells is overwritten by the second set of saved cells.

See also: `RestoreToRow`



---

## Chapter 6: Client Architecture

This chapter is devoted to demonstrating the use of the FlashFiler components by examining the Mythic Proportions order entry example application included with FlashFiler. This chapter will not attempt to recreate the example from scratch. Rather, it focuses on specific issues involved with using FlashFiler components in a database application.

The following topics are covered in this chapter:

- Setting up a data module to connect to a FlashFiler server
- Creating FlashFiler Tables that use Calculated and Lookup fields
- Using FlashFiler Tables to create a master/detail relationship
- Setting up data-entry forms
- Using Filters
- Ranges
- Finding Records
- Blob Streams

If you wish to follow along using Delphi, make sure that:

- FlashFiler is installed into a compatible IDE
- The FlashFiler components are available on the component palette
- A FlashFiler server is running
- The FlashFiler Explorer is able to connect successfully to the FlashFiler server
- An alias named “mythic” points to the Mythic Proportions demo data that is installed with FlashFiler. This alias must be created manually by using the FlashFiler Server | Aliases dialog, or by creating a new database using the FlashFiler Explorer. The Mythic Proportions demo data is located in the Examples\Delphi\Mythic directory, or the Examples\Cbuildr\Mythic directory, depending on the options selected when FlashFiler was installed.

## Setting Up a Data Module to Connect to a FlashFile Server

Data modules provide a centralized store for any non-visual controls, but they are particularly useful for data access controls. The data module for the order entry system (dMythic.pas) contains all the necessary components to connect to a remote FlashFile server. Most data modules also contain the basic tables and code required by many of the forms in a project.

Figure 6.1 illustrates the data module used in the Mythic Proportions order entry system. The primary purpose of this data module is to provide a connection to a FlashFile server. Of consequence to this purpose are the components in the upper left-hand corner. These components represent the connection to a FlashFile server, and are linked together in such a way that one component is dependent upon another.

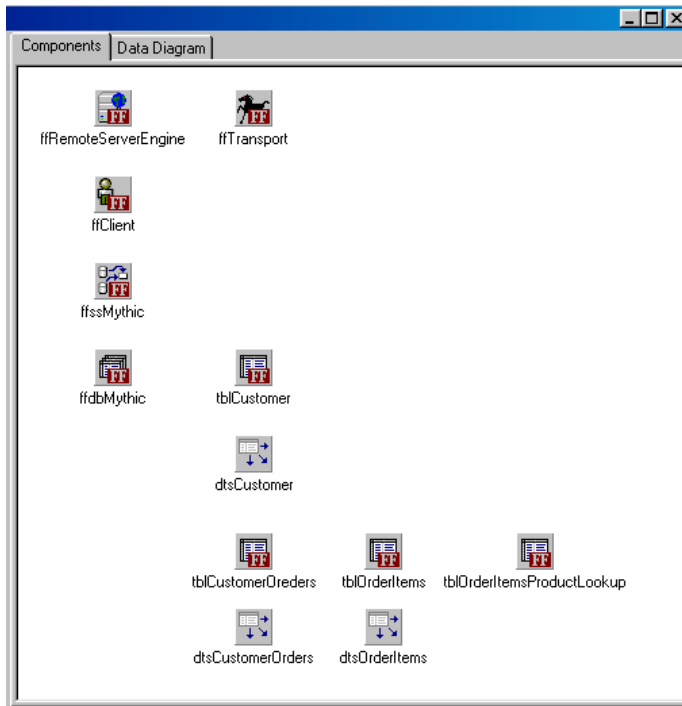


Figure 6.1: The data module for Mythic Proportions order entry system.

The TffRemoteServerEngine is used whenever a connection to a remote server is required. A remote server is a FlashFile server that resides in another application, or process. The TffRemoteServerEngine is connected to a transport via the Transport property. The transport provides a connection to the FlashFile server that can be used by one or more TffRemoteServerEngine components. The TffRemoteServerEngine and

TffLegacyTransport in the data module represented in Figure 6.1 provide the physical connection to a FlashFiler server. The connection between the TffRemoteServerEngine and the TffLegacyTransport is shown as Item 1 in Figure 6.2. Figure 6.2 represents the connections between the FlashFiler components used in the Mythic Proportions order entry data module.

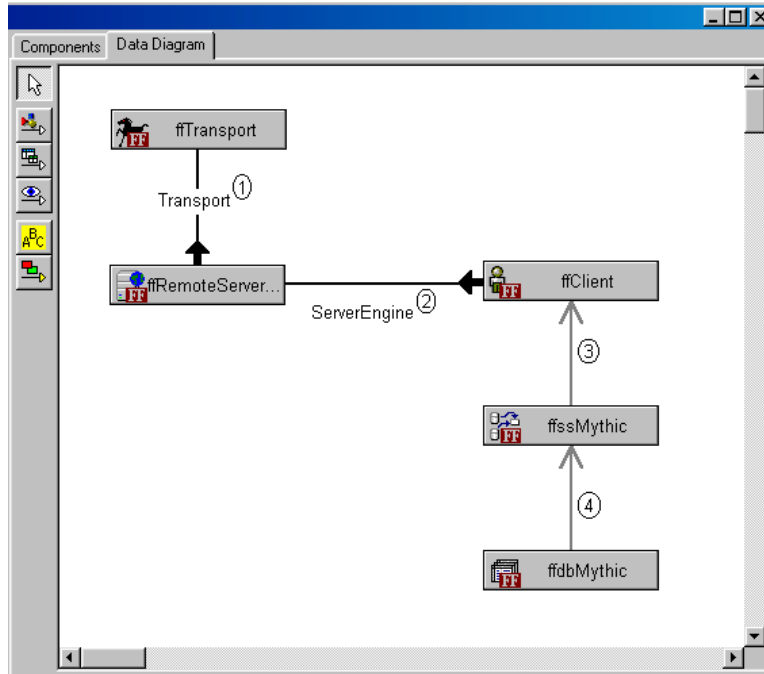


Figure 6.2: The data diagram for Mythic Proportions order entry system.

While a physical connection is priority, the server must also be made aware of the database or databases that will be used by a client application before any tables can be used. The TffClient, TffSession, and TffDatabase components work together with the TffRemoteServerEngine to request access to a database controlled by the remote FlashFiler Server.

The TffClient component is connected to a server engine via the ServerEngine property. The property may reference a local server engine (TffServerEngine), a remote server engine (TffRemoteServerEngine), or any 3<sup>rd</sup> party FlashFiler server engine that descends from TffBaseServerEngine. The TffClient component requires a unique name assigned to the ClientName property before a connection to a FlashFiler server can be established. This can be done manually, or by setting the AutoClientName property to True. The connection between the TffClient and the TffRemoteServerEngine is shown as Item 2 in Figure 6.2.



The TffSession component is connected to a TffClient component via the ClientName property. The property must match the ClientName property of an existing TffClient component. Similar to TffClient, the TffSession component requires a unique name assigned to the SessionName property before a connection to a TffClient component can be established. This can be done manually, by setting the AutoSessionName property of the session component to True. The connection between the TffSession and the TffClient is shown as Item 3 in Figure 6.2.

The TffDatabase component is connected to a TffSession component via the SessionName property. The property must match the SessionName property of an existing TffSession component. The TffDatabase component requires that two other properties be set before a connection to the session is made. The required AliasName property references an alias that resides on a server that points to a directory containing FlashFiler tables. The required DatabaseName property is a name unique to a TffSession that will be later referenced by the TffTable's DatabaseName property. The connection between the TffDatabase and the TffSession is shown as Item 4 in Figure 6.2.

# Creating FlashFiler Tables That Use Calculated and Lookup Fields

TffTable components may be added to a data module after a connection to a FlashFiler server is established. The TffTable components in the Mythic Proportions order entry application provide centralized access to data and services used by multiple forms. The tables provided in the data module represent customers, orders, and the items specific to each order. Calculated fields are used in the Orders table to provide tax, and amount due values. Lookup fields are used to provide additional columns to the Order Items table. These columns represent OEM number and item name data.

There are four steps to successfully creating a TffTable component and connecting it to the physical table containing data on a server:

1. Drop a new table on a data module.
2. Set the SessionName property to “MythicSession”.
3. Set the DatabaseName property to “dbMythic”.
4. Set the TableName property to a valid table.

6

## Calculated fields

Any special fields, like calculated or lookup fields, may be added once a table is connected. To create a calculated or lookup field, right-click on the TffTable component and select “New Field”. This opens a dialog box that contain configuration options for calculated and lookup fields. The dialog box in Figure 6.3 shows the configuration dialog box for one of the Mythic Proportions order entry application’s calculated fields.

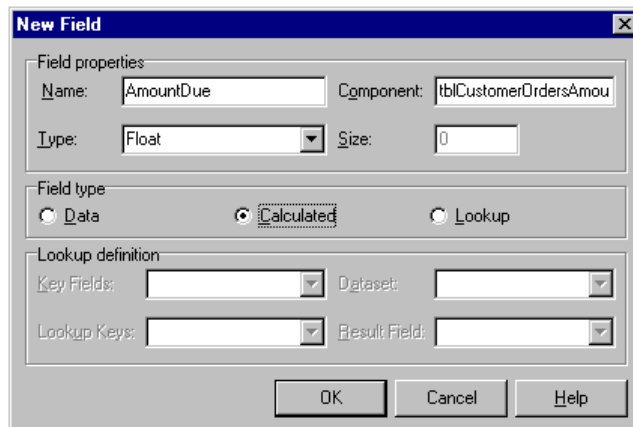


Figure 6.3: The New Field dialog box.

Of special note in the dialog are the Type and Size options. When the value of a calculated field is set, it is validated based on the value of these options. In the case where a calculated field represents a string, strings longer than the specified size will be truncated. Choose these options carefully.

When using calculated fields, take care to make the calculation code quick and efficient. When the table is in AutoCalc mode, calculated fields will be recalculated whenever a field value or record position changes.

The code example that follows is used to calculate the tax and amount due values for the CustomerOrders table in the example data module:

```
procedure TdtmMythic.tblCustomerOrdersCalcFields(
    DataSet : TDataSet);
begin
    {These calculated fields are used to generate information that
    is displayed on the OrderEntry dialog.}
    tblCustomerOrdersTaxTotal.Value :=
        (tblCustomerOrdersTaxRate.Value / 100) *
        tblCustomerOrdersItemsTotal.Value;

    tblCustomerOrdersAmountDue.Value :=
        tblCustomerOrdersItemsTotal.Value +
        tblCustomerOrdersFreight.Value -
        tblCustomerOrdersAmountPaid.Value +
        tblCustomerOrdersTaxTotal.Value;
end;
```

Of note in the code is the fact that there is no need to call Edit before making changes to the field values, nor a need to call Post after making changes.

## Lookup fields

Lookup fields have their own considerations and issues, especially when using a client-server database engine like FlashFiler. To create a lookup field, follow the same initial steps as for a calculated field. The dialog box in Figure 6.4 shows the configuration box for one of the Mythic Proportions order entry application's lookup fields.

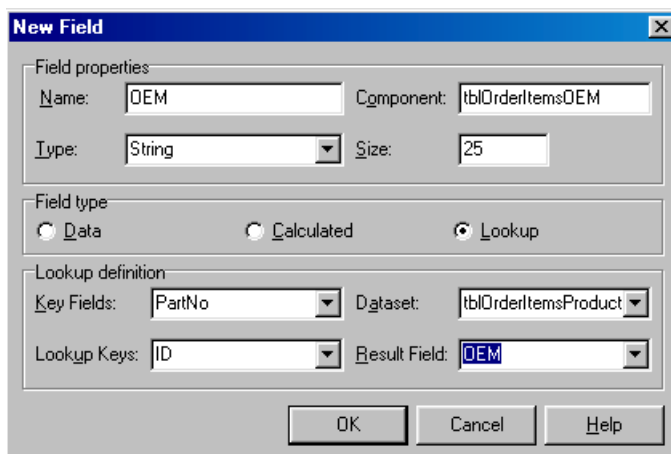


Figure 6.4: Configuration options for a Mythic Proportions order entry system lookup field.

Of special note in the dialog box are the Type and Size properties. The Type and Size properties of a lookup field should match the result field from the lookup table. If the values don't match, unwanted results like truncated strings may occur. Of primary concern when creating a lookup field is the setting of the Lookup Definition properties. These properties are important since they provide the necessary information a table needs to retrieve result information from a lookup table. The KeyFields property represents one or more fields in the primary table that contain key values that map to a lookup table. The Dataset property contains a reference by component name to the TffTable that is acting as a lookup table. The LookupKeys property references the fields in the lookup table that will be searched for the matching record. Finally, the ResultField property contains the actual data that will be made available in the primary table.

When lookup fields are used in FlashFiler, special care must be taken to make sure they remain a productive feature. When a lookup key value or record position changes, the lookup table will be queried for a fresh set of data. This can cause serious speed issues, especially when large lookup datasets and/or slow network connections are involved. A lookup field's LookupCache property can be set to True because most lookup datasets are relatively static. This keeps the lookup data in memory, which results in faster lookup operations.

## Using FlashFile Tables to Create a Master/Detail Relationship

TDataSource components can be added and linked to TffTable components once the connection components and tables have been added to a data module. TDataSource components provide the conduit for table data from TffTable to data-aware components. They are also necessary when configuring master/detail relationships. Once the TDataSource components have been added, you can configure any master/detail relationship necessary for an application. The Mythic Proportions order entry application configures such a relationship in its data module between the orders and order items table.

Master/detail relationships are not difficult to configure, but the configuration must be correct for the relationship to work. After setting the MasterSource and Index Name properties on your detail table, click on the ellipse on the MasterFields property. The dialog box shown in Figure 6.5 joins the fields between your tables.

6



Figure 6.5: The Field Link Designer dialog box links a detail table to the master table.

Note the Available Indexes combo box, and the two list boxes containing fields. The Available Indexes combo box contains a list of indexes for the detail table. Master/detail relationships are maintained internally using ranges. This requires that the detail table (tblOrderItems) have one or more indexes based on the key value stored in the orders table. An index containing the key or keys referenced in the master table should be selected. The second step is matching the key field in the detail field to a field in the master table that contains the same key. Once both fields are selected clicking the Add button will mark the fields as “Joined”. After one or more sets of fields are joined, the relationship will be completely configured. Also note, that since master/detail relationships are based on the index of a detail table, changing the index of that detail table will result in the relationship being invalidated.

## Setting Up Data Entry Forms

Once the data module is configured, setting up data entry forms is simple. After the layout of a screen is done, all that remains is to configure the properties for any data-aware controls. Almost all data-aware components have a DataSource property. This property should be set to reflect the data source that points to the TffTable containing the FlashFiler table data the control is to display. The DataSource component must be local to the form, or be available in another used form or data-module. Another common setting is the DataField property. This property reflects the field name for a specific column of data, and can reference a standard, calculated, or lookup field.

## Using Filters

One of the most powerful tools in a FlashFiler table is its filtering capabilities. In FlashFiler most filters can be run at the client or the server. Filters that run on the server are the quickest, and should be used whenever possible. When speed is not an issue, or when the general filter property does not provide enough functionality, filters can be run on the client. Filters can be created using two methods. First, a filter string can be specified using the TffTable's Filter property. Filter strings are in the following format:

```
FieldName = Value
```

FieldNames that contain spaces must be surrounded by brackets. When a field is a string field, the value must be surrounded by single quotes. The QuotedStr call from SysUtils provides this quoting functionality. A partial match filter can also be created by appending an asterisk to the end of a value.

A second filtering method uses the TffTable's OnFilterRecord event. When filters are active, this event will be called to evaluate each record in a dataset. This is the most flexible type of filter, as compiled source code can be used to determine exactly what is an acceptable record. The filter types can be combined if necessary, and are only active when the Filtered property is set to True.

In the Mythic Proportions order entry application, a filter is used to narrow the products that can be selected for purchase. This is done by passing the OEM number from the New Order dialog box into a filter expression. If only one record is returned, then it is added immediately to the order items table. If more than one record matches the filter, a selection dialog is displayed. The following code creates the available product filter:

```
procedure TfrmOrderEntry.btnAddItemClick(Sender : TObject);
var
    OEM : string;
    CancelItem : Boolean;
    RecCount : Integer;
begin
    {...}
    {Grab the OEM from the edit box for easy access}
    OEM := edtOEM.Text;

    {perform a filter against the product table based on item id.}
    tblProducts.Filter := '[OEM] = ' + QuotedStr(OEM + '*');
    tblProducts.Filtered := True;
```

Note the use of an asterisk within the string assigned to the Filter property. Without the asterisk, only records matching OEM exactly would be displayed. Also, the table's Filtered property is set to True. Without this setting, the string assigned to the Filter would not be evaluated.

---

# Ranges

The most efficient way of restricting a set of records based on a set of criteria is through the use of a range. Ranges are index based, so they are extremely fast. Ranges are set by specifying start and end values for the field or set of fields that are part of the TffTable's current index. In the Mythic Proportions order entry example, a range is used to display a set of orders for the specified customer that is within a certain number of days. The main form of the example contains a combo box named cboOrderRange that contains the following items:

- All orders
- Last 7 days
- Last 30 Days
- 1 Year

When the combo box is modified, it sets a property in the data module that in turn sets the range based on the selection. The range is reset whenever a new customer is selected. The following code sets the range according to the selection in the cboOrderRange combo box:

```
procedure TdtmMythic.tblCustomerAfterScroll(DataSet : TDataSet);
var
    {variable to store customer ID for easy access}
    ID : Longint;
begin
    {The customer table has completed a scroll operation, and we
    must set a range on the order table to emulate a master-
    detail relationship. The range is based on the ID of the
    customer, and the date-range of orders to display based on
    the internal CustomerOrderRange property. The property is set
    by the GUI's main form whenever the show combo box is
    changed.}
    ID := tblCustomerID.Value;
```



```

{Set range on Orders table}
with tblCustomerOrders do
    case CustomerOrderRange of
        rtLast7 : SetRange([ID, Date - 7], [ID, Date]);
        {include orders from the last 7 days that match ID}
        rtLast30 : SetRange([ID, Date - 30], [ID, Date]);
        {include orders from the last 30 days that match ID}
        rt1Year : SetRange([ID, Date - 365], [ID, Date]);
        {include orders for 1 year}
    else
        SetRange([ID], [ID]);
        {Include all orders}
    end;
end;
end;

```

6

Note that the first item in SetRange method's aStartValues property is the customer ID. Because the orders table contains order data for every customer, the order data must be narrowed down to a specific customer. The second item in the aStartValues property represents the current date minus the number of days to include option specified by the combo box. When using ranges, dates should be specified using their native TDateTime format instead of a localized string format. We are able to specify the item as "Date - x" because internally the TDateTime is a double type with the whole portion of the number representing the number of days, and the fractional portion representing the number of milliseconds.

## Finding Records

There are many methods to find records in FlashFiler. The Mythic Proportions order entry example uses the FindNearest method to locate a customer based on an exact or partial match of a supplied company name string. The FindNearest method suits these requirements because it can find based on partial and exact matches, and it finds records using an index resulting in quick and efficient search operations. The following code is an example of the use of the FindNearest method:

```
TffTable(CustDataset).FindNearest([edtCompanySearch.Text]);
```

Note that FindNearest depends on the current index to find a record.

---

## BLOB Streams

Using BLOB fields in FlashFile can sometimes be tricky. When used with normal data-aware controls, there are no problems, but people often run into a bit of trouble when trying to access BLOB data directly using streams. FlashFile includes a `TffBlobStream` class that makes these types of operations relatively simple. The Mythic Proportions order entry example uses a `TffBlobStream` to load an employee's picture into a FlashFile table from a disk file. The example code is as follows:

```
procedure TfrmEmployees.btnSelectPhotoClick(Sender : TObject);
var
    BS : TffBlobStream;
    FS : TFileStream;
begin
    if dlgOpenPicture.Execute then begin
        if not (Dataset.State in [dsEdit, dsInsert]) then
            Dataset.Edit;
        FS := TFileStream.Create(dlgOpenPicture.FileName, fmOpenRead
            or fmShareDenyWrite);
        try
            BS := TffBlobStream(Dataset.CreateBlobStream(
                Dataset.FieldByName('Picture'), bmReadWrite));
            try
                BS.CopyFrom(FS, FS.Size);
                Dataset.Post;
            finally
                BS.Free;
            end;
        finally
            FS.Free;
        end;
    end;
end;
```

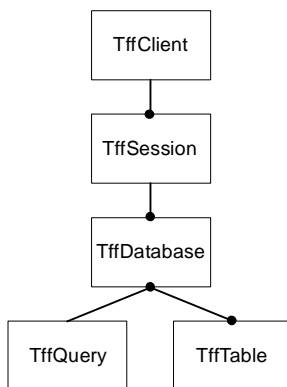
There are a few things of special note in the example. First, a file stream is created based on the file selected in the `TOpenPictureDlg`. This file stream is the source of data that will be loaded in the BLOB field. The `TffBlobStream` is created by calling the generic `CreateBlobStream` method in the `TDataSet` class and typecasting the resulting stream to a

TffBlobStream. This is possible because FlashFiler table internally creates a TffBlobStream when that method is called. Also notice that bmReadWrite mode is specified for the TffBlobStream. If just bmRead mode was specified, it would be impossible to modify the BLOB data stored in the table. Of final note is how that data is copied from the file stream to the blob stream. The TffBlobStream object, and all streams for that matter, contain a CopyFrom method. This method is used to copy a set of data from one stream to another. Finally if either stream had been used previously then it would have been necessary to reset the stream's position to 0.

## Chapter 7: FlashFiler Client

FlashFiler provides a rich client interface for connecting applications to a FlashFiler Server. The client side components in FlashFiler reflect FlashFiler's heritage as a BDE replacement engine. Developers who have built applications using the BDE will be on familiar ground. Most of the FlashFiler client classes have a BDE counterpart. The BDE has a TSession, where FlashFiler includes a TffSession, and likewise for the TDatabase and TTable components. FlashFiler strives to remain functionally compatible with the BDE wherever possible. This is of significant benefit because of the vast amount of technical information included with Delphi, third-party books, and tutorials that can be applied to database programming with FlashFiler.

There are four major client classes that must be included at some level within every FlashFiler applications. The most commonly used components are TffTable and TffQuery. They provide access to a FlashFiler data file, and can be used in conjunction with Borland's TDataSource and data-aware controls to build database applications with very little code. Depending on the application, a TffDatabase, TffSession, and/or TffClient component may also be required. These components form an object hierarchy as shown in Figure 7.1.



*Figure 7.1: The FlashFiler Client Class hierarchy.*

The TffClient component provides the actual connection to the FlashFiler server. There may be one or more TffSession components per TffClient, with the TffSession component representing a client session on the FlashFiler Server. There may be one or more TffDatabase components per TffSession with the TffDatabase component representing a FlashFiler database (i.e., alias). There may be one or more TffTable and TffQuery components per TffDatabase.

## Automatic components

When writing an application using the standard database components, in many instances the TDatabase and TSession components are not dropped on the form or data module. Behind the scenes, the VCL creates hidden TDatabase and TSession components for the TTable components. FlashFiler also works in a similar way. When writing a FlashFiler application, many developers don't want to be bothered with having to place database, session, and client components onto the form or data module. Instead, they would rather let the FlashFiler Client take care of the details and use some predefined defaults to connect the application to the server, open a session with the server, and open the correct database. In FlashFiler, these behind-the-scenes components are called automatic components (their Name property has the value "[automatic]"). FlashFiler creates these components when the client application is initialized, and the components provide a connection to a server. There are two automatic components: the client and the session. The session creates a TffDatabase object if one is required. These components are activated and used only when a table or query is opened (i.e., the Active property of a TffTable or TffQuery is set True) and the table or query is not already connected to explicit session and client objects.

For more information about the automatic components, see "TffBaseClient Class" on page 134 and "TffSession Component" on page 146.

## Multithreaded applications

There is one simple and inviolable rule when using FlashFiler in a multithreaded client application: you cannot share any FlashFiler database component between two or more threads. The database components are not thread-safe. The corollary from this rule is that every thread must have its own TffClient, TffSession, TffDatabase, TffQuery, and TffTable components. Once this is enforced, it follows that threads cannot share any database object between them.

In order to create a multithreaded application that accesses FlashFiler Server, either place the FlashFiler database components in their own data module or create them dynamically within a routine. Create an instance of the data module for each thread or call the dynamic creation routine for each thread. Each database component must have a unique Name property. In the case of a TffClient, it must have a unique value for its ClientName property. Similarly, TffSession must have a unique SessionName and TffDatabase must have a unique DatabaseName.

The following example shows how to dynamically create a unique set of database components that open a table on a FlashFiler Server:

```
FClient := TffClient.Create(nil);
FClient.AutoClientName := True;
FClient.Active := True;

FSession := TffSession.Create(nil);
FSession.ClientName := FClient.ClientName;
FSession.AutoSessionName := True;
FSession.Active := True; {Assumption: csAliasDir is always a
    path.}

FDB := TffDatabase.Create(nil);
FDB.SessionName := FSession.SessionName;
FDB.AutoDatabaseName := True;
FDB.AliasName := csAliasDir;

FTable := TffTable.Create(nil);
FTable.DatabaseName := FDB.DatabaseName;
FTable.SessionName := FDB.SessionName;
FTable.TableName := csContacts;
```

The Mythic Proportions ISAPI example, located in the Examples\CBuilDr\Mythic\ISAPI or Examples\Delphi\Mythic\ISAPI subdirectories of your FlashFiler installation, shows how to create a multithreaded ISAPI DLL.

## Database ancestor classes

The FFDBBASE unit provides two simple ancestral classes for the FlashFiler database class hierarchy. These base classes provide a simple paired relationship: TffDBListItem is a class that maintains an internal array of other list items using the class TffDBList. Descending the FlashFiler database classes (i.e., the TffTableProxy, TffBaseDatabase, TffSession, and TffBaseClient classes) from TffDBListItem means that the database classes inherit the ability maintain a list of owned objects. A base client object can maintain a list of sessions attached to it, a session object can maintain a list of databases open under it, and a database object can maintain a list of open tables.

To fit in with this design TffTable must descend from TffDBListItem. However, to participate in the VCL's data-aware capabilities, TffTable must descend from TDataSet. Delphi does not support multiple-inheritance. We resolved the problem using another method. TffTable is descended from the VCL's TDataSet, but it also has a hidden, tightly linked helper class called TffTableProxy. When it is created, a TffTable object will create its own TffTableProxy instance. The TffTable will participate in the VCL's data-aware functionality, whereas the table proxy object will participate in FlashFiler's database hierarchy.

Having provided this explanation, it must be noted that, in general use, you do not need to be concerned with the two separate classes defining a table. The FlashFiler client code takes care of the interaction between a table and its proxy. If an exception is raised with text relating to a table proxy class, it is sage to assume that the exception really refers to the TffTable class instead.

**Note:** The TffDBList class is not documented in this manual. For further details, you are encouraged to review the source code.

## Client classes hierarchy

Before continuing, it would be worthwhile to review the structure of the client classes. The classes and their relationships will be discussed at great length later in this chapter. However, the listing that follows will prepare you for the next two sections on Global Variables, and Helper Routines.

### TComponent (VCL)

```

    TffComponent (FFLLBase)
        TffDBListItem (FFDBBase)
            TffBaseClient
                TffClient
                TffCommsEngine
            TffSession
            TffBaseDatabase
                TffDatabase
            TffTableProxy

```

### TDataSet (VCL)

```

    TffDataSet
        TffBaseTable
            TffTable
        TffQuery

```

---

# Global Variables and Constants

The FFDB unit contains a few global variables that should be of value. These variables are exposed to provide information to the developer. However they should be used with caution.

## Variables

Clients

## Constants

AutoObjName

DefaultTimeOut

FFMaxBlobChunk

The descriptions for these variables assume knowledge of information available later on in the chapter. However, it makes sense to introduce these variables now.



## Reference Section

### AutoObjName

constant

```
const AutoObjName = '[automatic]';
```

↳ Stores the name used for automatic objects.

This constant is used internally by FlashFiler to label the automatic TffClient, and TffSession objects created upon the initialization of a FlashFiler client application. There should not be a need to use this constant. However, to use the FindFFClientsName routine to find the automatic Client, this constant should be used rather than a literal string.

See also: FindFFClientName

### Clients

variable

```
var Clients : TffClientList;
```

↳ Stores a reference to the client list container object.

The Client variable is a descendent of the TffDBList class, and contains an item for every TffBaseClient object for an application. Although this list makes it possible to add, change, or delete client objects, use of the published methods documented in this manual is recommended. FlashFiler relies on this global list to provide functionality to an application (for example, freeing all FlashFiler components at the end of the program). Any changes to the list might result in erroneous behavior from the application.

The following example closes all active TffBaseClient objects:

```
procedure CloseClients;
var
  Idx : Integer;
begin
  for Idx := Pred(Clients.Count) downto 0 do
    Clients[Idx].Close;
  end;
```

## DefaultTimeout

constant

```
const DefaultTimeout = 10 * 1000; {10 seconds}
```

↳ Stores the timeout setting used by a TffBaseClient object.

This constant initializes the Timeout property during the creation of the TffBaseClient component. To change the default timeout used for all Client components, modify this constant directly instead of setting the component properties individually or in code.

The value of this constant must be expressed in milliseconds. The default being 10,000 ms. (10 seconds).

See also: TffBaseClient.Timeout

## FFMaxBlobChunk

constant

```
const FFMaxBlobChunk : Integer = 64000;
```

↳ Stores the number of bytes of data that pass between the client and server in a single operation.

The TffBlobStream uses this constant during creation to initialize the ChunkSize property. The default setting of 64,000 means that up to 64,000 bytes of BLOB data are sent to or retrieved from the server at one time. This allows the FlashFiler Server to process requests from other clients while it is handling BLOB requests. Setting this constant to zero means that the entire BLOB is sent to or retrieved from the server at one time. When dealing with large BLOBs, do not set this constant to zero as it may cause a bottleneck on the FlashFiler Server.

**Note:** You may temporarily change the chunk size for a specific BLOB read or write using the ChunkSize property of the TffBlobStream class.

See also: TffBlobStream.ChunkSize

---

# Component Helpers

There are several helper routines in the FFDB unit that work to create and retrieve various FlashFiler components. In general, these routines are never accessed directly. However, they are used extensively within the implementation of the various client classes so they are documented here.

## Compatibility notes

The component helper routines have changed with version 2.0 of FlashFiler. The primary difference is only in the routine names. The CommsEngine routines of the past now have their names changed to Client. The change was made due to the deprecation of the TffCommsEngine component, and the introduction of the TffBaseClient class.

## Routines

7

FFSession	FindFFDatabaseName	GetFFDatabaseNames
FindAutoFFClient	FindFFSessionName	GetFFSessionNames
FindDefaultFFClient	GetDefaultFFClient	Session
FindDefaultFFSession	GetDefaultFFSession	
FindFFClientName	GetFFClientNames	

The descriptions for these routines assume knowledge of information available later on in the chapter. However, it makes sense to introduce these routines now.

## Reference Section

### FFSession

function

```
function FFSession : TffSession;
```

↳ Returns the default session object.

This routine returns the default session object for a FlashFiler application.

### FindAutoFFClient

function

```
function FindAutoFFClient : TffBaseClient;
```

↳ Returns the automatic client.

The return value is the automatic client that FlashFiler creates during the initialization of an application. If the automatic client has been freed, this routine returns nil. The automatic client created during initialization is always a TffClient object.

Since the FFDB unit frees the automatic client at the end of the program, it is not necessary to free the automatic client programmatically.

The following code opens the automatic client's connection to the server, if the automatic client exists:

```
procedure OpenAutomaticClient;
var
  Client : TffBaseClient
begin
  Client := FindAutoFFClient;
  if Assigned(Client) then
    Client.Open;
end;
```

```
function FindDefaultFFClient : TffBaseClient;
```

↳ Returns the default client.

This function returns the TffClient component whose IsDefault property is set to True. If there are no client components, this routine returns nil.

The following code returns the name of the default client, if a default client exists:

```
function GetDefaultClientName : string;
var
  Client : TffBaseClient
begin
  Client := FindDefaultFFClient;
  if Assigned(Client) then
    Result := Client.ClientName
  else
    Result := '';
end;
```

See also: GetDefaultFFClient

```
function FindDefaultFFSession : TffSession;
```

↳ Returns the default session.

This function returns the default session for the FlashFiler Client. The default session is the main session for the default client. If there are no clients, or if the default client has no sessions, this routine returns nil.

The following code opens the default session, if the default session exists:

```
procedure OpenDefaultSession;
var
  Client : TffBaseClient
begin
  Client := FindDefaultFFClient;
  if Assigned(Client) then
    Client.Open;
end;
```

See also: GetDefaultFFSession

```
function FindFFClientName(  
    const aName : string) : TffBaseClient;
```

↪ Returns the client with a specific ClientName.

Use this function to find the TffBaseClient component whose ClientName property matches the aName parameter. If no such client exists, this function returns nil.

aName is the name of the client component and must contain a value.

The following example opens the automatic client, if it exists:

```
procedure OpenAutomaticClient;  
var  
    Client : TffBaseClient;  
begin  
    Client := FindFFClientName(AutoObjName);  
    if Assigned(Client) then  
        Client.Open;  
end;
```

See also: AutoObjName, TffBaseClient.ClientName

```
function FindFFDatabaseName(  
    aSession : TffSession; const aName : string;  
    const aCreate : Boolean) : TffBaseDatabase;
```

↪ Returns the database object owned by the specified session and having a DatabaseName property equal to the specified name.

This function searches for a database whose DatabaseName property has the value specified by parameter aName and is connected to the session component aSession. You must specify a value for aSession and aName. aCreate indicates whether the database is to be created if a matching database is not found. Set aCreate to True if the database is to be created otherwise set aCreate to False.

If aCreate is set to True then aName must specify the name of a database alias visible to the session. If aSession is not already open, this function forces aSession to open so that it may interrogate the FlashFile Server for a list of aliases.

If aCreate is False and no matching database is found, this function returns nil.

The following example returns a database object in the default session for a specified alias:

```
function GetDatabase(  
    const aAliasName : string) : TffDatabase;  
var  
    DB : TffBaseDatabase;  
begin  
    DB := FindFFDatabaseName(FFSession, aAliasName, True);  
    Result := TffDatabase(DB);  
end;
```

See also: `TffBaseDatabase.DatabaseName`

## **FindFFSessionName**

**function**

```
function FindFFSessionName(  
    const aName : string) : TffSession;
```

7

↳ Returns the session object with a specific SessionName.

This function returns the session component whose SessionName property contains the value specified by the aName parameter. You must specify a value for aName. The session does not have to be associated with the default or automatic client; all TffBaseClient components are searched for the specified session. If a matching session is not found, this function returns nil.

The following example looks for the automatic session and opens it if it exists:

```
procedure OpenAutomaticSession;  
var  
    Session : TffSession;  
begin  
    Session := FindFFSessionName(AutoObjName);  
    if Assigned(Session) then  
        Session.Open;  
end;
```

See also: `TffSession.SessionName`

## GetDefaultFFClient

function

```
function GetDefaultFFClient : TffBaseClient;
```

↳ Returns the default TffBaseClient object.

This function returns the default TffBaseClient object for a FlashFiler application. If there are no client objects at all, this routine raises an EffDatabaseError exception having error code ffdse\_NoDefaultCE.

See also: FindDefaultFFClient

## GetDefaultFFSession

function

```
function GetDefaultFFSession : TffSession;
```

↳ Returns the default TffSession object.

This function returns the default TffSession object for a FlashFiler application. The default session is the main session for the default TffBaseClient object. If there is no default TffBaseClient object, this routine raises an EffDatabaseError exception having error code ffdse\_NoSessions.

See also: FindDefaultFFSession

## GetFFClientNames

function

```
procedure GetFFClientNames(aList : TStrings);
```

↳ Returns a list of client names.

Use this function to obtain the names of all existing TffBaseClient objects. aList is an instance of a class inheriting from TStrings. aList is populated with the names of the TffBaseClient objects. This function clears aList before adding the client names.

The following example populates a memo control with a list of client names:

```
GetFFClientNames(Memo1.Lines);
```



```
procedure GetFFDatabaseNames(  
    aSession : TffSession; aList : TStrings);
```

↳ Returns a list of database names and alias names visible to the specified session.

Use this function to obtain the value of the `DatabaseName` property of all active database objects managed by a specific session as well as the alias names visible to the session. `aSession` is the `TffSession` component for which the values are to be retrieved. `aList` is an instance of a class inheriting from `TStrings`. This function clears `aList` before adding populating it with values.

Values are added in the following order: first, the `DatabaseName` of all database components having their `Active` property set to `True` and their `SessionName` property set equal to the `SessionName` property of `aSession`; second, all aliases known to the session specified by `aSession`.

The following example populates a combo box with a list of database and alias names for the default session:

```
GetFFDatabaseNames(FFSession, ComboBox1.Items);
```

**Note:** Calling this routine results in the session being opened if it is not already open.

---

**GetFFSessionNames****function**

---

```
procedure GetFFSessionNames(aList : TStrings);
```

↳ Returns a list of sessions names.

This function retrieves the name of each session object associated with each `TffBaseClient` object. `aList` is an instance of a class inheriting from `TStrings`. `aList` is populated with the session names in the FlashFiler environment. This function clears `aList` before adding all known session names. Session names are retrieved for all `TffBaseClient` objects.

---

**Session****function**

---

```
function Session : TffSession;
```

↳ Returns the default session object.

This function returns the default session object for a FlashFiler application. It exists solely to provide compatibility with the standard VCL.

See also: `FindDefaultFFSession`, `GetDefaultFFSession`

---

# TffDBListItem Class

As described in “Database ancestor classes” on page 119, the purpose of the TffDBListItem class is to manage a list of TffDBListItem instances. This allows classes inheriting from TffDBListItem, such as the TffBaseClient class, to own one or more other FlashFiler database objects. TffDBListItem is used as the base class for a number of other FlashFiler classes. As such, the specific tasks that its methods perform vary depending on the derived class. For example, the Open method will establish a connection on the server when the derived class is TffBaseClient. For other descendents it will simply allocate resources on the server. For this reason, the text in the reference section that follows is generic.

## Hierarchy

TComponent (VCL)	
❶ TffComponent (FFLLBASE) .....	78
TffDBListItem (FFDBBase)	

## Properties

IsOwned

## Methods

CheckActive	ForceClosed	Open
CheckInactive	❶ FreeInstance	
Close	❶ NewInstance	

## Reference Section

### CheckActive

method

```
procedure CheckActive;
```

↳ Tests the active state of the object on the server.

This routine verifies the object is active on the server. If it is not, then an `EffDatabaseError` exception having error code `ffdse_MustBeOpen` is raised.

### CheckInactive

method

```
procedure CheckInactive(aCanClose : Boolean);
```

↳ Tests the active state of the object on the server and potentially makes it inactive.

This routine verifies the object is not active on the server. If the object is active and the `aCanClose` parameter is set to `True`, this routine calls the `Close` method. If the `aCanClose` parameter is set to `False`, and the object is active, an `EffDatabaseError` exception having error code `ffdes_MustBeClosed` is raised.

### Close

method

```
procedure Close;
```

↳ Frees resources on the server previously allocated for this object.

Instances of `TffDBListItem` maintain an internal reference count. The reference count is incremented when the instance is opened. The reference count is decremented by this method. If the reference count decrements to zero, the object is deactivated (i.e., closed) on the server. In descendant components such as `TffBaseClient`, this may cause the connection with the FlashFiler Server to be terminated or specific resources on the FlashFiler Server to be released.

See also: `ForceClosed`

## ForceClosed

method

```
procedure ForceClosed;
```

↪ Forces the object to close regardless of internal reference count.

This routine forces the object to close, resetting its internal reference count to zero. In descendent components such as `TffBaseClient`, this may cause the connection with the FlashFile Server to be terminated or specific resources on the FlashFile Server to be released.

See also: `Close`

## IsOwned

read-only property

```
property IsOwned : Boolean
```

↪ Indicates whether the object is owned by another object.

This read-only property returns `True` if the object is owned by another object. In `FlashFile` this property is `True` for all data-access components that descend from `TffDBListItem` except for components that inherit from `TffBaseClient`.

## Open

method

```
procedure Open;
```

↪ Opens the object on the server.

Use this routine to open a data access component. If the component does not have a value for its `Name` property, an `EffDatabaseError` exception having error code `ffdse_NeedsName` is raised. If the component has an owner and the owner name is not set (e.g., the `ClientName` property for a `TffSession` component, the `SessionName` property for a `TffDatabase` component), an `EffDatabaseError` exception having error code `ffdse_NeedsOwnerName` is raised.

This method sets the object's active state to `True`. Internally, this causes the object to validate required properties and increment the internal reference count. On descendent components, a connection to, or a resource on the FlashFile Server may be opened to complete the operation.

# TffBaseClient Class

The TffBaseClient class provides a communication link to a FlashFiler Server. This class is responsible for creating the necessary components required to connect to a server when a server engine is not specified.

The TffBaseClient class is used as a base class for the TffClient, and TffCommsEngine components. You must never create a TffBaseClient instance directly.

## Hierarchy

TComponent (VCL)

❶ TffComponent (FFLLBASE).....	78
❷ TffDBListItem (FFDBBase).....	122
TffBaseClient (FFDB)	

## Properties

AutoClientName	IsDefault	Sessions
ClientID	❷ IsOwned	Timeout
ClientName	ServerEngine	UserName
CommsEngineName	SessionCount	

## Methods

CheckActive	❶ FreeInstance	❶ NewInstance
❷ CheckInactive	GetServerName	❷ Open
❷ Close	GetServerNamesEx	
❷ ForceClosed	IsConnected	

## Reference Section

### **AutoClientName**

**property**

```
property AutoClientName : Boolean
```

Default: False

↪ Indicates whether the component is to generate it's own unique name.

Use this property to have a client component automatically generate a unique value for its Name property. Set this property to True to obtain a unique name. The value is guaranteed to be unique within a FlashFiler application, making it useful for multithreaded applications where a unique client component is required for each thread.

FlashFiler generates the unique name by appending the integer representation of the object's pointer to the string "FFClient\_".

When set to False, you are required to assign a proper value to the component's ClientName property.

If you set this property to True while the component is open (i.e., active) then an EfficDatabaseError exception having error code ffdse\_CEMustBeClosed is raised.

See also: ClientName

### **ClientID**

**read-only property**

```
property ClientID : TffClientID
```

```
TffClientID = type TffWord32;
```

```
TffWord32 = type DWORD;
```

↪ Specifies the unique identifier used by a server engine to represent the client.

The client uses this property when communicating with a server engine. Before a connection to the server is established, this property has a value of zero. Use this property if you need to make direct calls into the server engine.

See also: ServerEngine

**ClientName****property**

```
property ClientName : string
```

Default: Empty string

↪ Specifies a unique name for the client component.

This property provides a unique identifier for the component. The ClientName may be any valid identifier, as long as it is unique among all components inheriting from TffBaseClient within the application. When the value of this property changes, the same value is assigned to the CommsEngineName property. This property may be set only while the component is closed (i.e., inactive). If this property is set while the component is open, an EffDatabaseError exception having error code ffdse\_CEMustBeClosed is raised.

If another client component already has the same value for ClientName, an EffDatabaseError exception having error code ffdse\_CENameExists is raised.

See also: AutoClientName, CommsEngineName

**CommsEngineName****property**

```
property CommsEngineName : string
```

Default: Empty string

↪ Specifies a unique name for the client component.

This property is provided for backwards compatibility with TffCommsEngine. This property provides the same functionality and has the same behavior as the ClientName property. When the value of this property changes, the same value is assigned to the ClientName property.

See also: AutoClientName, ClientName

**GetServerNames****method**

```
procedure GetServerNames(aServerNames : TStrings);
```

↪ Retrieves a list of all servers accessible by this client.

This method causes the associated server engine to retrieve a list of all the FlashFiler Servers visible to the client. For local server engines, this results in a string list containing a single entry labeled “Local”. For remote server engines, results depend on the transport associated with the remote server engine. Please see the GetServerNames method of the TffBaseTransport class on page 331 in Chapter 9 for more information.

aServerNames is a instance of a class inheriting from TStringList. aServerNames is cleared before the server names are retrieved. The list of server names is placed in the TStringList instance specified by aServerNames. The client must be open before this method is called. Otherwise, an EFlashDatabaseError exception having error code ffidse\_CEMustBeOpen is raised.

The following example populates a memo control with a list of available servers:

```
Client1.GetServerNames(Memo1.Lines);
```

See also: ServerEngine

## **IsConnected** method

```
function IsConnected : Boolean
```

↪ Indicates whether the client is connected to a FlashFiler Server.

A client component may be active before an actual connection to the server is made. Use this function to determine if the connection exists. This function returns True if the client has established a connection with a FlashFiler Server otherwise it returns False.

## **IsDefault** property

```
property IsDefault : Boolean
```

Default: False

↪ Indicates whether this component is the default client component.

When a TffSession component is placed on a form, it sets its ClientName property to the ClientName property of the default client component. The default client component is the client component having its IsDefault property set to True. Only one client component may have its IsDefault property set to True.

When the value of this property is changed, the following rules apply:

- If you set IsDefault to True, the other component with its IsDefault property set to True now has its IsDefault property set to False.
- If you set IsDefault to False, the automatic client's IsDefault property is set to True.
- If you set the automatic client's IsDefault property to False, the first user-created TffBaseClient descendent is made the default.
- If the automatic client is the only client available, an attempt to set IsDefault to False will fail.
- The IsDefault property may be modified while the component is active.



```
property ServerEngine : TffBaseServerEngine
```

↪ References the server engine component attached to the component.

A client component uses a server engine component to communicate with a FlashFiler Server. Use this property to connect the client component to a server engine component. FlashFiler is delivered with two server engines, TffServerEngine and TffRemoteServerEngine. Use TffRemoteServerEngine to connect to a FlashFiler Server running as a separate application on the local machine or a machine accessible via a network connection. Use TffServerEngine when embedding a FlashFiler Server within the client application.

If no server engine is specified, the client automatically creates an instance of TffRemoteServerEngine based on values stored in the Windows registry. If the SingleEXE define is enabled in file FFDEFINE.INC, the client automatically creates an instance of TffServerEngine.

If you set this property while the component is active, an EffDatabaseError exception having error code ffdse\_CEMustBeClosed is raised.

The following example makes a direct call to the server engine to add an alias:

```
Client.CheckActive(False);  
Check(Client.ServerEngine.DatabaseAddAlias(  
    'AliasName', 'c:\AliasPath', Client.ClientID));
```

See also: IsConnected

```
property SessionCount : Integer
```

↪ Returns the number of TffSession components attached to this client.

Use this property to retrieve the number of TffSession components managed by this client, regardless of their active state.

See also: Sessions

```
property Sessions[aInx : Integer] : TffSession
```

↳ Returns a specific TffSession component associated with the client.

Use this property to access a specific TffSession component managed by the client. aInx is the index into the list of managed sessions. The lower bound of aInx is zero and the upper bound is the value returned by the SessionCount property minus one.

The following example frees all session associated with a client:

```
procedure FreeSessions(Client : TffBaseClient);
var
  Idx : Integer;
begin
  Assert(Assigned(Client));
  with Client do
    for Idx := Pred(SessionCount) downto 0 do
      Sessions[Idx].Free;
    end;
```

See also: SessionCount

```
property Timeout : Longint
```

Default: The value of the DefaultTimeOut constant

↳ Specifies the timeout used for client operations and serves as the base timeout for owned components.

This property specifies the number of milliseconds the client is willing to wait for operations to complete on the FlashFiler Server. When the client is first opened, the value of this property is sent to the server. When the value of this property changes, the new value is sent to the server engine and is used for all subsequent client-related operations.

The value of this property is also used by owned components whose Timeout property is set to -1.

See also: DefaultTimeOut

```
property UserName : TffName  
TffName = string[ffcl_GeneralNameSize];  
ffcl_GeneralNameSize = 31;
```

Default: Empty string

↳ Specifies the user name passed to the FlashFiler Server when a connection is established.

The value of this property is passed to the FlashFiler Server when the client establishes a connection with the server. You may not use this property to determine the user's name after establishing a connection with the server.



# TffClient Component

The TffClient component is used to establish a connection with a FlashFiler Server. It manages one or more TffSession components. TffClient inherits from TffBaseClient and the documentation for TffBaseClient covers everything available to TffClient.

If you are using TffClient components in a multithreaded application, you must have a separate TffClient component per thread. See “Multithreaded applications” on page 118 for more details.

## Hierarchy

TComponent (VCL)

❶ TffComponent (FFLLBASE) .....	78
❷ TffDBListItem (FFDBBase) .....	131
❸ TffBaseClient (FFDB) .....	134
TffClient (FFDB)	

7

## Properties

❸ AutoClientName	❸ IsDefault	❸ Sessions
❸ ClientID	❷ IsOwned	❸ Timeout
❸ ClientName	❸ ServerEngine	❸ UserName
❸ CommsEngineName	❸ SessionCount	

## Methods

❸ CheckActive	❶ FreeInstance	❶ NewInstance
❷ CheckInactive	❸ GetServerName	❷ Open
❷ Close	❸ GetServerNamesEx	
❷ ForceClosed	❸ IsConnected	



# TffCommsEngine Component

The TffCommsEngine component is provided for backwards compatibility with FlashFiler 1 applications. The purpose of the TffCommsEngine component is to establish a connection with a remote or embedded FlashFiler Server. Except for two differences, it is very similar to the TffClient component. The first difference is that you may not directly connect a TffCommsEngine component to a server engine component. The second difference is that you may specify the communications protocol to be used by the TffCommsEngine component via its Protocol property.

If you are using TffCommsEngine components in a multithreaded application, you must have a separate TffCommsEngine component per thread. See “Multithreaded applications” on page 118 for more details.

## Hierarchy

TComponent (VCL)

❶ TffComponent (FFLLBASE) .....	78
❷ TffDBListItem (FFDBBase) .....	131
❸ TffBaseClient (FFDB) .....	134
TffCommsEngine (FFDB)	

## Properties

❶ AutoClientName	❷ IsOwned	❸ Sessions
❶ ClientID	Protocol	❸ Timeout
❶ ClientName	❸ ServerEngine	❸ UserName
❶ CommsEngineName	ServerName	
❶ IsDefault	❸ SessionCount	

## Methods

❷ CheckActive	❶ FreeInstance	❶ NewInstance
❷ CheckInactive	❸ GetServerName	❷ Open
❷ Close	❸ GetServerNamesEx	ProtocolClass
❷ ForceClosed	❸ IsConnected	

# Reference Section

Protocol	property
----------	----------

property Protocol : TffProtocolType  
TffProtocolType = (ptSingleUser, ptTCPIP, ptIPXSPX, ptRegistry);  
Default: ptRegistry

↪ Specifies the protocol used to connect to a remote server engine.

Use this property to specify the communications protocol for a connection with the FlashFiler Server. Behind the scenes, the component instantiates one instance each of TffRemoteServerEngine and TffLegacyTransport. It configures the transport to use the specified protocol.

The following table lists the meaning of each value for this property.

Value	Meaning
ptSingleUser	Use Single User Protocol (i.e., SUP, Windows messaging) to connect with a FlashFiler Server on the same computer.
ptTCPIP	Use Winsock TCP/IP to connect with a FlashFiler Server on the same computer or another computer via a LAN, WAN, or an internet connection.
ptIPXSPX	Use Winsock IPX/SPX to connect with a FlashFiler Server on the same computer or another computer via a LAN.
ptRegistry	Look in the registry for the communications protocol.

When the value of this property is set to `ptRegistry`, the component does the following in order to determine the actual protocol:

1. If the procedure `FFClientConfigOverrideProtocol` in the `FFCLCFG` unit has been called, use the overridden protocol.
2. Otherwise, look for value `Protocol` in registry key `HKEY_LOCAL_MACHINE\Software\TurboPower\FlashFiler\Client Configuration`. The `Protocol` string value, if it exists, will have the following values: `Single User`, `NetBios`, `TCP/IP`, or `IPX/SPX`. The relevant protocol is then used.
3. Otherwise, the `Single User Protocol` is used.

See also: `ProtocolClass`

## **ProtocolClass**

**method**

```
function ProtocolClass : TffCommsProtocolClass;
```

7

↳ Returns the protocol class associated with the current value of the `Protocol` property.

Use this function to obtain the class associated with the current protocol. If the `Protocol` property is currently set to `ptRegistry`, this function uses the rules specified in the `Protocol` property description to determine the protocol.

See also: `Protocol`

property ServerName: string

Default: Empty string

- ↪ Specifies the name and address of the FlashFiler Server to which the component must connect.

This property is used only when the component attempts to establish a connection with a FlashFiler Server. If you leave ServerName empty, the component broadcasts for available servers using the TffBaseClient.GetServerNames method. This property remains empty, even if a server is found.

If you set the ServerName property, it must match the name of an available server. The format of the name depends upon the value of the Protocol property.

If the current protocol is ptSingleUser then no value is needed for ServerName.

If the current protocol is ptIPXSPX, ServerName must be of the form servername@xx-xx-xx-xx-xx-xx, where the servername is the exact name of the server and xx-xx-xx-xx-xx-xx is the IPX address of the server. The FlashFiler Server shows this information on its main screen.

If the current protocol is ptTCPIP, ServerName must be of the form servername@iii.iii.iii.iii, where servername is the exact name of the server and iii.iii.iii.iii is the IP address of the server. The FlashFiler Server shows this information on its main screen.

See also: Protocol





## TffSession Component

The TffSession component is the FlashFiler equivalent of the VCL TSession component. The session object manages a group of databases within an application. The session object is dependent upon either an explicit or automatic TffBaseClient descendent component.

If you are using TffSession components in a multithreaded application, you must have a separate TffSession component per thread. See “Multithreaded applications” on page 118 for more details.

### Automatic session

Like the BDE, FlashFiler automatically creates a session object on initialization. In FlashFiler, this automatically created session has its Name property set to “[automatic]”. TffDatabase and TffTable components use this session by default. Since this default session is automatically created, FlashFiler gives you the same simple experience of use as does the standard VCL: you do not need to explicitly create or use ancillary components; the FlashFiler Client takes care of simple server Client requirements.

The use of multiple sessions is optional unless you need the ability to connect to multiple servers concurrently, or need to access FlashFiler within multiple threads.

Just as FlashFiler automatically creates the automatic session on initialization, the Client also destroys the automatic session on finalization. Destroying the automatic session explicitly is not recommended.

# Hierarchy

TComponent (VCL)	
❶ TffComponent (FFLLBASE) .....	78
❷ TffDBListItem (FFDBBase) .....	131
TffSession (FFDB)	

## Properties

Active	CommsEngineName	ServerEngine
AutoSessionName	DatabaseCount	SessionID
Client	Databases	SessionName
ClientName	IsDefault	Timeout
CommsEngine	❷ IsOwned	

## Methods

AddAlias	❷ ForceClosed	IsAlias
AddAliasEx	❶ FreeInstance	ModifyAlias
❷ CheckActive	GetAliasNames	❶ NewInstance
❷ CheckInactive	GetAliasPath	❷ Open
❷ Close	GetDatabaseNames	OpenDatabase
CloseDatabase	GetServerDateTime	SetLoginParameters
DeleteAlias	GetTableNames	SetLoginRetries
DeleteAliasEx	GetTaskStatus	
FindDatabase	GetTimeout	

## Events

OnChooseServer	OnLogin
OnFindServers	OnStartUp

## Reference Section

Active

property

property Active : Boolean

Default: False

🔗 Specifies whether a session is active.

Use the Active property to determine the current state of the session. Active is False by default, meaning there are no open database clients or datasets associated with the session. A session may be active even if there are currently no active database clients.

Setting Active to True starts the session and triggers the session's OnStartup event handler. If the owning client is not active, the session automatically activates the client.

Setting Active to False closes any open databases owned by the session or tables connected to the session. This action has no affect on the owning client component.

The following example closes all FlashFiler sessions within an application:

```
procedure CloseAllSessions;
var
  ClientIdx  : Integer;
  SessionIdx : Integer;
begin
  for ClientIdx := Pred(Clients.Count) downto 0 do
    with Clients[ClientIdx] do
      for SessionIdx := Pred(SessionCount) downto 0 do
        Sessions[SessionIdx].Active := False;
      end;
    end;
  end;
```

See also: OnStartup

```
procedure AddAlias(const aName : string;  
    const aPath : string);
```

➤ Adds a new database alias to the FlashFiler Server.

Use this method to create a new database alias on the server. `aName` is the name of the alias to create and must have a value. It must be unique among the other alias names on the server. If the alias name already exists on the server, this method raises an `EffDatabaseError` exception having error code `DBIERR_NameNotUnique`.

`aPath` is a directory on the computer hosting the FlashFiler Server, or a UNC path accessible by the hosting computer. For example, specifying “C:\DATABASE” as the path will create an alias that points to the server’s C: drive, not the client’s. To create an alias pointing to the client machine’s C: drive, `aPath` must be specified in UNC format (e.g., `\\ClientMachine\Share\Database`). `aPath` must have a value.

The value of the `aPath` parameter may match the path of an existing alias, however the FlashFiler Server recognizes this fact and treats the aliases as pointing to the same database.

See also: `AddAliasEx`

```
function AddAliasEx(  
    const aName : string; const aPath : string) : TffResult;
```

➤ Adds a new database alias to the FlashFiler Server.

This method is identical in functionality to the `AddAlias` method previously described. The only difference is that this method returns an error code whereas `AddAlias` raises an exception when an error occurs.

The result of the function reflects the status of the `AddAlias` operation. If an error occurs the result will be a value other than `DBIERR_NONE`.

`aName`, and `aPath` must be non-empty strings. If the alias is successfully added to the server, this function returns `DBIERR_NONE`. If an alias having the same name already exists on the FlashFiler Server then this function returns `DBIERR_NAMENOTUNIQUE`.

See also: `AddAlias`

```
property AutoSessionName : Boolean
```

Default: False

↳ Indicates whether the component is to generate its own unique name.

Use this property to have a session component automatically generate a unique value for its Name property. Set this property to True to obtain a unique name. The name is guaranteed to be unique within a FlashFiler application, making it useful for multithreaded applications where a unique client component is required for each thread.

FlashFiler generates the unique name by appending integer representation of the object's pointer to the string "FFSession\_".

When set to False, you must assign a proper value to the component's SessionName property.

If you set this property to True while the component is open (i.e., active) then an EffDatabaseError exception having error code ffdse\_SessMustBeClosed is raised.

See also: Active, SessionName

---

**Client****read-only property**

---

```
property Client : TffBaseClient
```

↳ Specifies the client owning the session.

Use this run-time property to obtain a handle on the client managing the session component. The client managing the session is the client whose ClientName property matches the session's ClientName property. Until the session's ClientName property is set, this property is set to nil.

To change the client managing this session, set the session's Active property to False and then alter the session's ClientName property.

See also: Active, ClientName

## ClientName property

---

property ClientName : string

Default: “[automatic]”

↪ Specifies the value of the ClientName property of the client owning the session.

The ClientName property is important because it identifies which client provides a communications link to the session. By default, the ClientName property is set to “[automatic]”. The session may, however, be linked to any TffBaseClient descendent component within an application.

This property can be changed only while the session is inactive. If you attempt to change this property while the session is active, an EffDatabaseError exception having error code ffdse\_SessMustBeClosed is raised.

See also: Active, Client

## CloseDatabase method

---

```
procedure CloseDatabase(aDatabase : TffBaseDatabase);
```

↪ Closes a specific database managed by the session.

Use this method to close a database managed by the session. The aDatabase parameter specifies the database component to close. If FlashFiler Client dynamically created the database component, the component is freed.

**Note:** Dynamic database components are also closed automatically when the last dataset associated with the implicit database component is closed.

## CommsEngine read-only property

---

```
property CommsEngine : TffBaseClient
```

↪ Specifies the client owning this session.

Use this run-time property to determine which client owns the session. This property is provided for backwards compatibility with FlashFiler 1. This property provides functionality identical to the Client property described previously.

See also: Client, CommsEngineName

```
property CommsEngineName : string
```

Default: “[automatic]”

↪ Specifies the value of the ClientName property of the client owning this session.

This property is provided for backwards compatibility with FlashFiler 1. The CommsEngineName property is important because it identifies which client provides a communications link to the session. By default, the CommsEngineName property is set to “[automatic]”. The session may, however, be linked to any TffBaseClient descendent component within an application.

This property can be changed only while the session is inactive. If you attempt to change this property while the session is active, an EffDatabaseError exception having error code `ffdse_SessMustBeClosed` is raised.

See also: CommsEngine

## **DatabaseCount**

**read-only property**

```
property DatabaseCount : Integer
```

↪ Indicates the number of database components managed by the session.

Use DatabaseCount to determine the number of database components attached to the session. The count includes not only the explicit database objects dropped on the form and connected to this session, but also the temporary databases that are automatically created when a table is attached directly to an alias.

DatabaseCount is often used with the Databases property to iterate through the current set of databases associated with the session.

See also: Databases

```
property Databases[aInx : Integer] : TffBaseDatabase
```

↳ Provides access to the database components managed by the session.

Use this property to access the database components attached to the session. The array includes not only the explicit database objects dropped on the form and connected to this session, but also the temporary databases that are automatically created when a table is attached directly to an alias.

The following example shows how to use this property by setting the read-only property for all databases managed by a specific session:

```
procedure SetDatabasesReadOnly(aSession : TffSession;
    aReadOnly : Boolean);
var
    Idx : Integer;
begin
    with aSession do
        for Idx := 0 to Pred(DatabaseCount) do
            if Databases[Idx] is TffDatabase then
                TffDatabase(Databases[Idx]).ReadOnly :=
                    aReadOnly;
            end;
        end;
    end;
```

See also: DatabaseCount

## DeleteAlias

method

```
procedure DeleteAlias(const aName : string);
```

↳ Deletes a specific alias from the FlashFiler Server.

Use this method to delete a specific alias on the FlashFiler Server. aName is the name of the database alias to be deleted. Deleting an alias does not affect any FlashFiler clients that currently have the database open. Deleting an alias does not delete any of the tables or files contained by the database.

If the FlashFiler Server does not contain the specified alias then an `EffDatabaseError` exception having error code `DBIERR_UNKNOWNDB` is raised.

See also: DeleteAliasEx



```
function DeleteAliasEx(const aName : string) : TffResult;
```

↪ Deletes the given alias at the server.

This method is identical in functionality to the `DeleteAlias` method previously described. The only difference is that, when an error occurs, this function returns an error code instead of raising an exception. If the alias is deleted successfully, this function returns `DBIERR_NONE`. If the alias does not exist on the server, this function returns `DBIERR_UNKNOWNDB`.

See also: `DeleteAlias`

```
function FindDatabase(  
    const aName : string) : TffBaseDatabase;
```

↪ Returns the database object having a `DatabaseName` set equal to the value of parameter `aName`.

Use this function to find a database owned by the session and having the value of parameter `aName` for its `DatabaseName` property. `aName` specifies the value that must be in the database's `DatabaseName` property. You must specify a value for `aName`. This method calls `FindFFDatabaseName` directly, passing `Self` as the `aSession` parameter.

See also: `DatabaseName`, `FindFFDatabaseName`

## GetAliasNames

method

```
procedure GetAliasNames(aList : TStrings);
```

↪ Retrieves the list of alias names visible to this session.

Use this method to retrieve the list of aliases from the FlashFiler Server. aList is an instance of a class inheriting from TStrings. aList is cleared prior to the aliases being fetched from the server. This method calls GetAliasNamesEx and activates the session if it is not already activated.

See also: GetAliasNamesEx, GetAliasPath

## GetAliasNamesEx

method

```
procedure GetAliasNamesEx(  
    aList : TStrings; const aEmptyList : Boolean);
```

↪ Retrieves the list of alias names visible to this session.

Use this method to retrieve the list of aliases from the FlashFiler Server. aList is an instance of a class inheriting from TStrings. If aEmptyList is True then aList is cleared prior to the aliases being fetched from the server. Otherwise, the contents of aList are left as is.

See also: GetAliasNames, GetAliasPath

## GetAliasPath

method

```
procedure GetAliasPath(const aName : string; var aPath : string);
```

↪ Retrieves the server-relative path for an alias.

Use this method to obtain the path for an alias. aName is the name of the alias. You must specify a value for aName. This method interrogates the server for the path associated with the alias and returns it in parameter aPath. If the alias is unknown, aPath is set to an empty string. No exception is raised.

**Note:** aPath is relative to the server. If aPath is returned as “C:\prod\database” then C: refers to the server’s C: drive, not the client’s C: drive. The following example populates a TStrings object with a list of aliases and their associated paths for the specified session:

```
procedure PopulateWithAliasData(aSession : TffSession;
    aList : TStrings;
var
    Idx : Integer;
    Path : string;
begin
    Assert(Assigned(aSession));
    Assert(Assigned(aList));
    with aSession do begin
        Alist.BeginUpdate;
        try
            GetAliasNames(aList);
            for Idx := 0 to Pred(aList.Count) do begin
                GetAliasPath(aList.Strings[Idx], Path);
                AList.Strings[Idx] := AList.Strings[Idx] + ' ' +
                    Path);
            end;
        finally
            aList.EndUpdate;
        end;
    end;
end;
```

See also: GetAliasNames, GetAliasNamesEx

## GetDatabaseNames

method

```
procedure GetDatabaseNames(aList : TStrings);
```

↳ Returns a list of the database names and alias names visible to the specified session.

Use this function to obtain the value of the DatabaseName property for all active database objects managed by the session as well as the alias names visible to the session. aList is an instance of a class inheriting from TStrings. This function clears aList before populating it with values.

Values are added in the following order: first, the DatabaseName of all database components having their Active property set to True and their SessionName property set equal to the SessionName property of the session; second, all aliases known to the session.

See also: GetFFDatabaseNames

```
function GetServerDateTime(  
    var aServerNow : TDateTime) : TffResult;
```

↪ Retrieves the current date and time from the server.

Use this method to retrieve the date and time of the FlashFiler Server as a TDateTime value. The server's current date and time are returned in parameter aServerNow. This function does not respect any client time zone settings. If the server and client are in different time zones, the client is responsible for converting the server's time zone to the client's time zone.

---

**GetTableNames****method**

---

```
procedure GetTableNames(  
    const aDatabaseName : string; const aPattern : string;  
    aExtensions : Boolean; aSystemTables : Boolean;  
    aList : TStrings);
```

↪ Retrieves the names of all tables in the specified database.

Use this method to obtain a list of all the tables within a specific database (i.e., alias). aDatabaseName may be either the value of a database's DatabaseName property or the name of an alias on the FlashFiler Server. If this parameter is set to an empty string, an empty list is returned.

Use aPattern to specify a wildcard pattern for matching table names. If aPattern is an empty string, all tables within the database are returned. If aPattern contains a pattern, only those table names matching the pattern are returned.

Set aExtensions to True in order to have the file extension returned with the table name. Otherwise, if aExtensions is False, the file extension is omitted from the returned table names.

Set aSystemTables to True in order to include server-specific tables (e.g., FFSINFO, FFSALIAS) as well. Otherwise, if aSystemTables is False then server-specific tables are excluded.

aList is an instance of a class inheriting from TStrings. This method clears aList before retrieving the table names.

```
function GetTaskStatus(
    const aTaskID : Longint; var aCompleted : Boolean;
    var aStatus : TffRebuildStatus) : TffResult;

TffRebuildStatus = packed record {Rebuild operation status}
    rsStartTime : longint;      {start time in ticks}
    rsSnapshotTime : longint;   {snapshot time in ticks}
    rsTotalRecs : longint;      {total count of records to read}
    rsRecsRead : longint;       {count of records read}
    rsRecsWritten : longint;     {count of records written}
    rsPercentDone : longint;     {RecsRead*100/TotalRecs}
    rsErrorCode : TffResult;     {error result for process}
    rsFinished : boolean;       {process has finished}
end;
```

↪ Returns the status for a server-side task.

7

When a client program initiates a pack, re-index, or restructure of a table, the task can take quite some time if the table is large. These rebuild tasks are done entirely at the server and control returns to the client application immediately after the task starts. This leaves the client free to do other things during the rebuild operation. However, there is no spontaneous notification from the server when the rebuild task completes. An application can use the Session's `GetTaskStatus` method to periodically check the status of the rebuild task.

`aTaskID` identifies the rebuild task. This is the unique ID returned by the function that started the rebuild task. `aCompleted` is `True` if the task is complete. `aStatus` contains information about the current progress of the rebuild task. `aStatus` does not contain valid information if the task has completed.

There is a difference between the `aCompleted` parameter and the `rsFinished` field of the `TffRebuildStatus` record. When you poll the task status with this routine, you will eventually get a status with `aCompleted` set to `False` and `rsFinished` set to `True` (`rsPercentDone` will equal 100). When the server sends back this finished status record, it takes the opportunity to remove the task identified by `aTaskID` from its list of tasks. If you then poll the task status again, the server can no longer find an entry for `aTaskID`. Rather than return an error code, it just sets `aCompleted` to `True` (the task identified by `aTaskID` can't be found, so the task must have completed). Task IDs are only removed in the following situations: when the client detaches from a server, all of its tasks are removed; and when the client polls the task status and the status is marked as finished.

See also: `TffBaseDatabase.PackTable`, `TffBaseDatabase.ReindexTable`,  
`TffBaseDatabase.RestructureTable`

```
function GetTimeout : Longint;
```

↪ Retrieves the current timeout value for the session.

The `GetTimeout` method retrieves the actual timeout being used by a `Session` object. When the session's `Timeout` property is set to `-1`, this method returns the `Timeout` of the client component managing the session. The session's `Timeout` property is returned when it contains a value other than `-1`.

When needing a timeout value in code, use `GetTimeout` instead of reading the `Timeout` property directly.

See also: `Timeout`

```
function IsAlias(const aName : string) : Boolean;
```

↪ Indicates whether the given name is the name of an alias.

Use this method to determine if a name is the name of an alias on the `FlashFiler Server`. `aName` is the alias name to verify. The name comparison is case-insensitive. This function returns `True` if the name matches an alias name. Otherwise, it returns `False`.

```
property IsDefault : Boolean
```

Default: `False`

↪ Indicates whether the session is the default session for the application.

If this session is the application's default session, this property returns `True` otherwise it returns `False`. Another way to obtain a handle on the default session is to call function `FindDefaultFFSession` or function `GetDefaultFFSession`.

**Note:** When a database, table, or query is dropped onto a form, its `SessionName` property is automatically set to the default session's `SessionName` property.

See also: `FindDefaultFFSession`, `GetDefaultFFSession`, `TffDatabase.SessionName`, `TffDataSet.SessionName`

```
function ModifyAlias(
    const aName : string; const aNewName : string;
    const aNewPath : string) : TffResult;
```

↳ Modifies the name and/or path of the specified alias.

Use this method to change the name or path of an existing alias. `aName` is the name of an existing alias. `aNewName` is a new name for the alias. You must specify a non-empty string for `aName` and `aNewName`. If you do not wish to change the name of the alias, set `aNewName` equal to `aName`.

`aNewPath` is the new path for the alias. You must specify a non-empty string for `aNewPath`.

Internally, the FlashFiler Server deletes the original alias and replaces it with a new alias whose name and path are set to `aNewName` and `aNewPath`, respectively. If the alias is successfully modified, this function returns `DBIERR_NONE`. If the server already contains an alias having the name specified by parameter `aNewName`, the error code `DBIERR_NAMENOTUNIQUE` is returned. If the server does not contain an alias having the name specified by parameter `aName`, the error code `DBIERR_UNKNOWNDB` is returned.

## OnChooseServer

event

```
property OnChooseServer : TffChooseServerEvent

TffChooseServerEvent = procedure(
    aSource : TObject; aServerNames : TStrings;
    var aServerName : TffNetAddress;
    var aResult : Boolean) of object;

TffNetAddress = string[ffcl_NetAddressSize];

ffcl_NetAddressSize = 63;
```

↳ Defines an event handler called when there is more than one possible server to which the session may connect.

The `OnChooseServer` event is called when no value is specified for the `ServerName` property of the `TffBaseClient` and the session attempts to connect to a server. Internally, the FlashFiler Client broadcasts for available servers. If more than one responds to the broadcast, this event is called so that the application may choose the server.

The `aSource` parameter is the session component initiating the connection.

`aServerNames` parameter is a list of server names responding to the broadcast. Each item in the list contains the full server name and network address in the form `servername@netaddress`.

The event handler is responsible for determining which server, if any, the session will use. This can be done either by pre-selecting a server or by allowing the user to choose via a custom dialog.

If no server is acceptable set `aResult` to `False`. Otherwise, set `aResult` to `True` and the `aServerName` parameter to the name and network address of the chosen server. The value of `aServerName` should have an exact match in the `aServerNames` list.

**Note:** If you do not set this event, FlashFiler Client displays its own dialog for the user to make a choice.

See also: `TffBaseClient.GetServerNames`, `OnFindServers`

## OnFindServers

event

```
property OnFindServers : TffFindServersEvent
```

```
TffFindServersEvent = procedure(  
    aSource : TObject; aStarting : Boolean) of object;
```

↳ Defines an event handler called before and after the owning client tries to find available servers.

This event provides the opportunity to display a splash screen, informative text, or change the cursor during what could be a lengthy operation. The event is fired twice: once just before the session's owning client starts broadcasting for servers and once just after the broadcast operation is over.

The `aSource` parameter is the session component initiating the broadcast for available servers. `aStarting` indicates whether the operation is beginning (`True`) or completing (`False`).

See also: `OnChooseServers`



```
property OnLogin : TffLoginEvent

TffLoginEvent = procedure(
    aSource : TObject; var aUserName : TffName;
    var aPassword : TffName; var aResult : Boolean) of object;

TffName = string[ffcl_GeneralNameSize];

ffcl_GeneralNameSize = 31;
```

✚ Defines an event handler called when the user must login to the server.

The OnLogin event is called during the session's activation process, when the server requires the user to log on. This event provides the opportunity to display a custom logon dialog that enables the user to log on to the server. If no handler is specified for this event, FlashFiler Client displays its own logon dialog.

The aSource parameter is the session component initiating the connection. The event handler is responsible for obtaining the user name password from the user. Place the user name and password in the aUserName and aPassword parameters, respectively. Set aResult to True if a user name and password were obtained. Otherwise set aResult to False in order to abort the log on process.

**OnStartup****event**

```
property OnStartup : TNotifyEvent
```

✚ Defines an event handler called just before a session is made active.

Use the OnStartup event to take specific actions when a session is activated. OnStartup is triggered when the session is activated by either setting the Active property to True directly, or by opening a database, query, or table associated with the session.

**OpenDatabase****method**

```
function OpenDatabase(const aName : string) : TffBaseDatabase;
```

✚ Opens the database having the given DatabaseName property.

Use this method to open a database whose DatabaseName property is set to the value specified by the aName parameter. The session searches for the database component. If found, the database is opened and returned by this function. If the database is not found, aName is assumed to be the name of an alias on the server. A temporary database is created and the temporary database is returned by this function.

If the database cannot be opened, an exception is raised.

## ServerEngine

read-only property

```
property ServerEngine : TffBaseServerEngine
```

↪ References the ServerEngine component through which the session routes its requests.

FlashFile is delivered with two server engine components, TffServerEngine and TffRemoteServerEngine. All requests in an application are routed through a server engine. This run-time property returns the server engine through which the session's requests are being routed.

See also: SessionID

## SessionID

read-only property

```
property SessionID : TffSessionID
```

```
TffSessionID = type TffWord32;
```

```
TffWord32 = type DWORD;
```

↪ Identifies the unique ID assigned to the session by a server engine.

When a session establishes a connection with the server engine, the server engine assigns the session a unique ID. This run-time property returns the ID assigned to the session. Before the session is active, this property returns zero. The session's unique ID is used when routing a request to the server engine.

See also: ServerEngine

## SessionName

property

```
property SessionName : string
```

Default: Empty string

↪ Specifies the unique name of the session.

The SessionName property is used to uniquely identify a session to the database, query, and table components. The property may be set only when the session's Active property is set to False. If this property is changed while the session is active, an EffDatabaseError exception having error code ffdse\_SessMustBeClosed is raised.

The value of SessionName must be unique across all sessions for all client components in an application.

See also: AutoSessionName

```
procedure SetLoginParameters(  
    const aUser : TffName; const aPassword : TffName);
```

✚ Specifies the UserID and Password used to login to a FlashFiler server.

The SetLoginParameters routine is used to specify a UserID and Password to connect to a FlashFiler server. This routine may be used in place of an OnLogin event when the UserID and Password to be used is already known.

The required aUser parameter specifies a UserID that is known by the server. The optional aPassword parameter specifies a string of characters less than 32 characters in length.

See also: SetLoginRetries

```
procedure SetLoginRetries(const aRetries : Integer);
```

✚ Specifies the number of times the login request will be tried before an open operation fails.

When a session connects to a secure FlashFiler server it must supply a valid UserID and Password. This method specifies the number of times the open operation will be tried before an exception is raised.

The required aRetries parameter must specify an integer with a value of at least 1.

See also: SetLoginParameters

---

property Timeout : Longint

Default: -1

- ↪ Specifies the timeout for session operations and the base timeout for owned databases, queries, and tables.

This property specifies the number of milliseconds in which the FlashFiler Server must complete a session-specific operation. If the server is unable to complete the operation in the specified amount of time, an instance of `EffException` having error code `fferrReplyTimeout` is raised.

If this property has the value `-1` then the session uses the `Timeout` property of its owning `TffBaseClient` component. When this property is modified, the new timeout value is sent to the FlashFiler Server and used in subsequent operations for the session.

This property is also used by owned components (i.e., databases, queries, and tables) whose `Timeout` property has the value `-1`.

See also: `GetTimeout`

# TffBaseDatabase Class

The TffBaseDatabase class defines the foundational interface for a database component. Never create a TffBaseDatabase component directly.

## Hierarchy

TComponent (VCL)	
❶ TffComponent (FFLLBASE) .....	78
❷ TffDBListItem (FFDBBase) .....	131
TffBaseDatabase (FFDB)	

## Properties

FailSafe	InTransaction	❷ IsOwned
----------	---------------	-----------

## Methods

❷ CheckActive	GetFFDataDictionary	PackTable
❷ CheckInactive	GetFreeDiskSpace	ReindexTable
❷ Close	GetTimeout	Rollback
CloseDataSets	InTransaction	StartTransaction
Commit	IsSQLBased	TableExists
❷ ForceClosed	❶ NewInstance	TransactionCorrupted
❶ FreeInstance	❷ Open	TryStartTransaction

## Reference Section

### CloseDataSets

method

```
procedure CloseDataSets;
```

↪ Closes all open tables owned by this database.

Use this method to close all tables and queries managed by the database component. The database's `Connected` property is left as is.

See also: `TffDatabase.Connected`

### Commit

method

```
procedure Commit;
```

↪ Applies all updates for the current transaction to the database.

This procedure tells the server to write all updates for the currently active transaction to disk. Once this procedure has completed, the changes have been permanently applied to the tables involved in the transaction. The current transaction is the last transaction started explicitly by a call to `StartTransaction`.

Before calling `Commit`, an application should check the status of the `InTransaction` property. If there is no transaction active, the `Commit` method raises an `EffDatabaseError` exception having error code `DBIERR_NOACTIVETRAN`.

If the `TransactionCorrupted` method was called after starting the transaction but prior to calling `Commit`, an `EffDatabaseError` exception having error code `DBIERR_FF_CorruptTrans` is raised. The same exception and error code are raised if the transaction was corrupted on the server (e.g., a record could not be deleted).

The following example illustrates the use of the Commit method:

```
aDatabase.StartTransaction
with aTable do
  try
    Edit;
    FieldByName('Status').AsString := cnShipped;
    Post;
    aDatabase.Commit;
  exception
    aDatabase.Rollback;
    raise;
end;
```

See also: InTransaction, RollBack, StartTransaction, TryStartTransaction, TransactionCorrupted

## FailSafe

property

property FailSafe : Boolean

Default: False

↳ Sets the FailSafe mode for a database.

Use this property to enable or disable fail-safe transactions on the FlashFiler Server for this database. Set this property to True in order to enable fail-safe transactions. Set this property to False in order to disable fail-safe transactions.

If fail-safe transactions are enabled, the FlashFiler Server creates a journal file for each transaction. The journal file is used to recover data lost during a system crash. By default, fail-safe transactions are disabled. See “Transaction Journal Recovery” on page 47 for more information.

```
function GetFFDataDictionary(
    TableName : TffTableName; Stream : TStream) : TffResult;

TffTableName = string[ffcl_TableNameSize];

ffcl_TableNameSize = 31;
```

↪ Retrieves the DataDictionary for a specific table.

Use this method to retrieve the data dictionary for a table within the database. The TableName parameter contains the name of the table within the alias on the FlashFiler Server. You must specify a non-empty string for TableName. If a table having the specified table name does not exist, the error code DBIERR\_NOSUCHTABLE is returned. If the format of TableName is invalid (i.e., it contains a file extension) then error code DBIERR\_INVALIDTABLENAME is returned.

aStream is an instance of a class inheriting from TStream. If the specified table is found, the FlashFiler Server writes its data dictionary to aStream. The following example shows how to retrieve the data dictionary for a specific table:

```
function GetDataDictionary (aDatabase : TffBaseDatabase;
    const ATableName : string) : TffDataDictionary
var
    Stream : TMemoryStream;
begin
    Assert(Assigned(aDatabase));
    Stream := TMemoryStream.Create;
    try
        Check(aDatabase.GetFFDataDictionary(ATableName,
            Stream));
        Stream.Position := 0;
        Result := TffDataDictionary.Create(4096);
        Result.ReadFromStream(Stream);
    finally
        Stream.Free;
    end;
end;
```

See also: TffDataDictionary



```
function GetFreeDiskSpace(var aFreeSpace : Longint : TffResult;
```

↪ Returns the amount of disk space available on the drive hosting the database.

Use `GetFreeDiskSpace` to determine the amount of disk space available for the database on the server. `aFreeSpace` represents the amount of free space in kilobytes. If a value of `-1` is returned then the server could not find the path associated with the database. To use this method, the database must be connected to the server. If the database is not connected, an `EffDatabaseError` exception having error code `ffdse_DBMustBeOpen` is raised.

```
function GetTimeout : Longint;
```

↪ Returns the current Timeout value for the database component.

The `GetTimeout` method retrieves the actual timeout being used by a database component. When the database's timeout property is set to `-1`, this method returns the Timeout of the session component managing the database. The database's Timeout property is returned when it contains a value other than `-1`. For this reason, you should use `GetTimeout` instead of reading the Timeout property directly.

See also: Timeout

```
property InTransaction : Boolean
```

↪ Indicates whether a transaction has been started on this database component.

Use this property to determine if the database component is in the midst of a transaction. If the `StartTransaction` method has been called successfully and `Commit` or `Rollback` have not been called, this property returns `True` otherwise it returns `False`. The database does not know if any other database components are in the midst of a transaction.

In the following example, the code commits only starts a transaction if the database is not already in a transaction:

```
if (not aDatabase.InTransaction) then  
    aDatabase.StartTransaction;
```

See also: Commit, Rollback, StartTransaction

```
function IsSQLBased : Boolean;
```

↳ Returns whether the database is an SQL database.

In this version of FlashFiler, this routine always returns False. This routine is present for compatibility with the BDE.

## PackTable

method

```
function PackTable(const aTableName : TffTableName;
    var aTaskID : LongInt) : TffResult;

TffTableName = string[ffcl_TableNameSize];

ffcl_TableNameSize = 31;
```

↳ Starts a pack operation for the specified table.

Use this method to pack a table. Pack creates a new table and copies the records from the original table into the new table. Once the operation is complete, the original table is deleted, and the new table is renamed to replace the original table. The server must be able to open the original table exclusively otherwise the pack operation will not work.

After a pack, the table is not necessarily smaller. The table size is a multiple of the block size, so if there were fewer deleted records than will fit in a block, the table size is unaffected.

aTableName is the name of the original table, excluding a file extension. You must specify a non-empty string for aTableName. This method returns immediately. If the pack operation is started successfully, the unique ID of the server-side pack operation is stored in parameter aTaskID and this method returns DBIERR\_NONE. To monitor the progress of the pack, use the value of aTaskID in conjunction with the GetTaskStatus method of TffSession.

If the database component's ReadOnly property is set to True, the pack operation does not start and this method returns DBIERR\_READONLYDB.

If the specified table is a server table, the pack operation does not start and this method returns DBIERR\_SYSFILEOPEN.

If the table name has an invalid format (e.g., includes a file extension) then the pack operation does not start and this method returns DBIERR\_INVALIDTABLENAME.

If the specified table does not exist, the pack operation does not start and this method returns DBIERR\_NOSUCHTABLE.

If the table is marked as read-only by the computer's file system, the pack operation does not start and this method returns DBIERR\_TABLEREADONLY.

See also: TffSession.GetTaskStatus

```
function ReindexTable(const aTableName : TffTableName;  
    const aIndexNum : Integer; var aTaskID : Longint) : TffResult;  
  
TffTableName = string[ffcl_TableNameSize];  
  
ffcl_TableNameSize = 31;
```

👉 Starts a re-index operation for the specified table.

Use this method to rebuild an index of a table. `aTableName` is the name of the table (excluding file extension) within the database. You must specify a non-empty string for `aTableName`. `aIndexNum` is the number of the index to be rebuilt and is base zero.

`ReindexTable` reconstructs the index from scratch by reading through the table's data file. You may use this method to re-index the Sequential Access Index (i.e., index zero), a composite index, or a user-defined index.

If the re-index operation is started successfully, the unique ID for the server-side re-index operation is returned in variable `aTaskID`. To monitor the progress of the re-index, use the value of `aTaskID` in conjunction with the `GetTaskStatus` method of `TffSession`.

If the database component's `ReadOnly` property is set to `True`, the re-index operation does not start and this method returns `DBIERR_READONLYDB`.

If the specified table is a server table, the re-index operation does not start and this method returns `DBIERR_SYSFILEOPEN`.

If the table name has an invalid format (e.g., includes a file extension) then the re-index operation does not start and this method returns `DBIERR_INVALIDTABLENAME`.

If the specified table does not exist, the re-index operation does not start and this method returns `DBIERR_NOSUCHTABLE`.

If the table is marked as read-only by the computer's file system, the re-index operation does not start and this method returns `DBIERR_TABLEREADONLY`.

See also: `TffSession.GetTaskStatus`

```
procedure Rollback;
```

↳ Terminates the current transaction.

Use this method to abort all changes made since calling method `StartTransaction`. This method cancels all record updates, inserts, and deletions made by the current transaction. It then ends the current transaction. The current transaction is the last transaction started calling method `StartTransaction`. The other way of ending a transaction is to use `Commit`, where the updates are applied to the database.

Before calling `Rollback`, an application should check the status of the `InTransaction` property. If there is no transaction active, the rollback routine raises an `EffDatabaseError` exception having error code `DBIERR_NOACTIVETRANS`.

If you call this method while the database component is closed, an `EffDatabaseError` exception having error code `ffdse_DBMustBeOpen` is raised.

The following example rolls back a transaction if an exception occurs:

```
aDatabase.StartTransaction
with aTable do
  try
    Edit;
    FieldByName('Status').AsString := cnShipped;
    Post;
    aDatabase.Commit;
  exception
    aDatabase.Rollback;
    raise;
  end;
```

See also: `Commit`, `InTransaction`, `StartTransaction`, `TryStartTransaction`

```
procedure StartTransaction;
```

↳ Starts a new transaction for the database.

Use this method to inform the FlashFiler Server that all subsequent changes to tables managed by this database component are part of a transaction. Once the transaction starts, the server caches all record modifications made by tables attached to this database component. The changes are applied to the database once the Commit method is called. The changes are canceled if the Rollback method is called.

If the database component has already started a transaction, an `EffDatabaseError` exception having error code `DBIERR_ACTIVETRAN` is raised. As an alternative, use the `TryStartTransaction` method, which does not raise an exception.

If you call this method and the database component is not connected to the server, an `EffDatabaseError` exception having error code `ffdse_DBMustBeOpen` is raised.

There may be only one active transaction per database component. However, there may be multiple active transactions on the FlashFiler database. For example, a client application could open the same alias twice using two different database components and then call `StartTransaction` on each database component.

The FlashFiler Server allows only one transaction to modify a table at any one time. For example, if transaction A modifies table `PurchaseOrders`, transaction B may not modify table `PurchaseOrders` until transaction A commits or rolls back its changes.

To see if a database component has already started a transaction, use the `InTransaction` method.

The following example starts a transaction for a database, if a transaction is not already in progress:

```
if (not aDatabase.InTransaction) then  
    aDatabase.StartTransaction;
```

See also: `Commit`, `InTransaction`, `Rollback`, `TransactionCorrupted`, `TryStartTransaction`

## TableExists

method

```
function TableExists(const aTableName : TffTableName) : Boolean;  
TffTableName = string[ffcl_TableNameSize];  
ffcl_TableNameSize = 31;
```

↪ Verifies the existence of a table.

Use this method to determine if a specific table exists within the database on the FlashFiler Server. `aTableName` is the name of the table whose existence is to be verified. You must specify a value for `aTableName`, however, `aTableName` does not need to include a file extension.

If you call this method while the database is closed, an `EffDatabaseError` exception with error code `ffdse_DBMustBeOpen` is raised.

## TransactionCorrupted

method

```
procedure TransactionCorrupted;
```

↪ Flags an in-progress transaction as corrupt.

Use this method to mark an active transaction as corrupted. If the transaction is subsequently committed, an `EffDatabaseError` exception having error code `DBIERR_FF_CorruptTrans` is raised.

See also: `Commit`, `InTransaction`, `Rollback`, `StartTransaction`, `TryStartTransaction`

## TryStartTransaction

method

```
function TryStartTransaction : Boolean;
```

↪ Starts a transaction if one is not already in progress.

Use this method in situations where the application does not know if a transaction has been started. If a transaction has not been started, this method starts a new transaction and returns `True`. If a transaction was started prior to `TryStartTransaction` then this method returns `False`.

This method, in conjunction with the `TransactionCorrupted` method is useful when creating independent methods that are coded with transactions in mind. Use the `TryStartTransaction` method in combination with the `TransactionCorrupted` method to create routines that work whether or not a transaction is active.

If a transaction is already active, then the procedure uses the current transaction. The following example shows a procedure that empties a table of all it's records within a transaction:

```

procedure ManuallyEmptyTable(aTable : TffTable);
var
    DB : TffDatabase;
    TransactionStarted : Boolean;
begin
    Assert(Assigned(aTable));
    DB := TffDatabase(aTable.Database);
    TransactionStarted := DB.TryStartTransaction;
    try
        with aTable do begin
            First;
            while not EOF do Delete;
        end;
        if TransactionStarted then
            DB.Commit;
    except
        if TransactionStarted then
            DB.Rollback;
        else
            TransactionCorrupted;
            raise;
        end;
    end;
end;

```

See also: Commit, InTransaction, Rollback, TransactionCorrupted, StartTransaction



## TffDatabase Component

The TffDatabase component is the FlashFiler equivalent of the VCL TDatabase component. This component manages a group of tables within an application. The database component is dependent upon either an explicit or the default session component.

There are two ways the database component is used in FlashFiler.

An explicit database is one that is added to the application in the form of a non-visual component, or by a component that is created at run time by your custom code. This component can be shared by a number of datasets to provide central management functions to an explicit group of tables.

An implicit database is one that is maintained internally by the FlashFiler client. This component cannot be referenced directly; instead it is accessed through the dataset's Database property. This type of database is known as a temporary database.

If you are using TffDatabase components in a multithreaded application, you must have a separate TffDatabase component per thread. See “Multithreaded applications” on page 118 for more details.

7

## Transactions

Use the TffDatabase component to wrap data changes within a transaction. To start a transaction, use the StartTransaction method. Changes made within a transaction are not applied until the transaction is committed via the Commit method. If an error occurs within the context of a transaction, the changes may be canceled using the Rollback method.

FlashFiler allows concurrent transactions per FlashFiler database, however only one transaction may be active per TffDatabase component. To determine if a database component has started a transaction, use the InTransaction property.

When a database component starts a transaction, all TffTable and TffQuery components owned by the database component participate in the transaction. For example, if table component A modifies records in table 1 and table component B modifies records in table 2 then their changes are cached until the Commit method is called. If Rollback is called, their changes to both tables 1 and 2 are canceled.

FlashFiler Server manages transactions on a per directory basis. If two applications open the same FlashFiler database using two different aliases (i.e., the aliases both point to the same directory on the server), FlashFiler recognizes the situation and manages the transactions properly.



A FlashFiler table within a directory may be modified by only one transaction at a time. For example, if transaction A modifies a record in table 1, transaction B cannot modify records in table 1 until transaction A has committed or rolled back. This means that transactions should be kept as short as possible. It also means that the user of a client application should never be allowed to control the length of a transaction. For example, do not start a transaction and then commit the transaction only after the user happens to press the Save button.

Table and query components not participating in a transaction, whether they be in separate client application or managed by a database component other than the one starting the transaction, do not see changes made by a transaction until the changes have been committed. Datasets outside of a transaction always read clean data. Datasets participating in a transaction read dirty data if they look at records that have been modified by the transaction.

Datasets outside of a transaction do not perform repeatable reads. For example, if a dataset reads record A at time T1, a transaction modifies and commits record A at time T2, and then the dataset re-reads record A at time T3 then the dataset sees the value from time T2, not time T1. If repeatable reads are required (i.e., after a dataset reads record A it can then go back and read the same value for record A again), start a transaction and perform the reads within the context of the transaction.

# Hierarchy

TComponent (VCL)

- ❶ TffComponent (FFLLBASE) ..... 78
  - ❷ TffDBListItem (FFDBBase) ..... 131
    - ❸ TffBaseDatabase (FFDB) ..... 177

TffDatabase (FFDB)

## Properties

AliasName	Exclusive	Session
Connected	❸ FailSafe	SessionName
DatabaseID	❸ InTransaction	Temporary
DatabaseName	❷ IsOwned	Timeout
DataSetCount	ReadOnly	
DataSets	ServerEngine	

## Methods

❷ CheckActive	❶ FreeInstance	❸ PackTable
❷ CheckInactive	❸ GetFFDataDictionary	❸ ReindexTable
❷ Close	❸ GetFreeDiskSpace	RestructureTable
❸ CloseDataSets	GetTableNames	❸ Rollback
❸ Commit	❸ GetTimeout	❸ StartTransaction
Create	❸ InTransaction	❸ TableExists
CreateTable	❸ IsSQLBased	❸ TransactionCorrupted
Destroy	❶ NewInstance	❸ TryStartTransaction
❷ ForceClosed	❷ Open	

## Reference Section

### AliasName

property

```
property AliasName : string
```

Default: Empty string

↪ Specifies the alias opened by the database.

Use this property to control the alias opened by a database component. AliasName must be a valid alias or path on the server. Its validity is checked when the database is opened. In order for a database to be opened (i.e., Connected property set to True), a valid name must be provided for both the AliasName and DatabaseName properties.

This property may be modified only when the database component is inactive. If this property is modified while the database is open, an EffDatabaseError exception having error code ffdse\_DBMustBeClosed is raised.

See also: Connected, DatabaseName

### Connected

property

```
property Connected : Boolean
```

Default: False

↪ Indicates whether the database component is connected to the FlashFiler Server.

By default, this property returns False indicating the database component is not connected to the server. To connect the database to a server, set this property to True. In order for the connection to succeed, a value must be supplied for both the AliasName and DatabaseName properties. If the database's session is inactive, setting this property to True also causes the session to open.

If this property is set to True and an alias matching the database's AliasName property is not found on the server, an EffDatabaseError exception having error code DBIERR\_UNKNOWNDB is raised. If the AliasName property references a path that does not exist on the server, an EffDatabaseError exception having error code DBIERR\_INVALIDDIR is raised.

See also: AliasName, DatabaseName

```
constructor Create(aOwner : TComponent); override;
```

↳ Creates a new database instance.

Use this method to dynamically create a database component at run time. This method initializes a new database component. The `SessionName` property is automatically set to the name of the default session, thereby setting the `Session` property to the default session component. The new database component is automatically added to the list of databases for the default session. It also creates an empty list of table components ready for the `DataSets` property.

See also: `DataSets`, `SessionName`

## CreateTable

## method

```
function CreateTable(const aOverWrite : Boolean;
    const aTableName : TffTableName;
    aDictionary : TffDataDictionary) : TffResult;

TffTableName = string[ffcl_TableNameSize];

ffcl_TableNameSize = 31;
```

↳ Creates a table based on the structure defined in `aDictionary`.

Use this method to create a new table on the FlashFiler Server. `aOverWrite` indicates whether or not an already-existing table may be overwritten. If `aOverWrite` is `True` then an existing table is overwritten. If `aOverWrite` is `False` and the table already exists, this method returns error code `DBIERR_TABLEEXISTS`.

`aTableName` is the name of the new table, excluding file extension. A non-empty string must be specified for `aTableName`. If `aTableName` is improperly formatted (e.g., it includes a file extension) then this method returns error code `DBIERR_INVALIDTABLENAME`.

The `aDictionary` parameter must contain an instance of `TffDataDictionary` defining the structure of the new table. For more information regarding `TffDataDictionary`, see page 572.

If the table is created successfully, this method returns `DBIERR_NONE`.

If the database component's `ReadOnly` property is set to `True`, the table is not created and this method returns `DBIERR_READONLYLDB`.

If `aTableName` refers to a table that exists and is opened by another client, the table is not rebuilt and this method returns `DBIERR_TABLEOPEN`.

The following example creates a table called TestTable in a given database:

```
procedure CreateTestTable(aDatabase : TffDatabase);
var
    Dict : TffDataDictionary1
    FldArray : TffFieldList;
    IdxHelpers : TffFieldIHList;
begin
    Database.Connected := True;
    Dict := TffDataDictionary.Create(8192);
    try
        with Dict do begin
            AddField('EMP_NO', '', fftInt16, 0, 0,
                True, nil); { 0 }
            AddField('FIRST_NAME', '', fftShortString, 15, 0,
                False, nil); { 1 }
            AddField('LAST_NAME', '', fftShortString, 20, 0,
                False, nil); { 2 }
            AddField('DEPT_NO', '', fftShortString, 3, 0,
                False, nil); { 3 }
            {Add an internal index on the DEPT_NO field}
            FldArray[0] := 3; { Field #3 }
            {Use default index helpers}
            IdxHelpers[0] := '';
            AddIndex('Dept', 'by Dept_No', 0, 1, FldArray,
                IdxHelpers, True, True, True);
        end;
        Check(aDatabase.CreateTable(True, 'TestTbl '
            Dict));
    finally
        Dict.Free;
    end;
end;
```

See also: TffDataDictionary

## DatabaseID

read-only property

```
property DatabaseID : TffDatabaseID
```

```
TffDatabaseID = type TffWord32;
```

```
TffWord32 = type DWORD;
```

↪ Returns the unique identifier assigned to the database component by FlashFiler Server.

This run-time property returns the unique ID given to the database by the FlashFiler Server when the database is opened. Prior to the database being opened, this property returns the value zero. This property is useful when making direct calls into a server engine.

See also: Connected

## DatabaseName

property

```
property DatabaseName : string
```

Default: Empty string

↪ Specifies the unique name of the database component.

Each database component within a client application must have a unique DatabaseName. This property is used to connect a TffTable and TffQuery with a database component. This property may be set only when the database is not connected to the FlashFiler Server (i.e., Connected is set to False). If this property is changed while the database is connected, an EfdDatabaseError exception having error code ffdse\_DBMustBeClosed is raised.

In order to connect a database to the FlashFiler Server, values must be supplied for both the AliasName and DatabaseName properties.

See also: AliasName, Connected

## DataSetCount

read-only property

```
property DataSetCount : Integer
```

↪ Specifies the number of active tables attached to the database component.

Use this property to determine how many active TffTable and TffQuery components are managed by the database component. This property is often used with the DataSets property to iterate through the current set of datasets associated with the database. Note that inactive tables and queries are not included in DataSetCount.

See also: DataSets

## DataSets

read-only property

```
property DataSets[aInx : Integer] : TffDataSet
```

↳ Provides access to the active tables and queries managed by the database component.

Use this run-time property to iterate through the active datasets managed by the database component. The datasets contained in the list can be any combination of TffQuery or TffTable components.

See also: DataSetCount

## Destroy

destructor

```
destructor Destroy; override;
```

↳ Deletes a database instance.

This destructor closes all active datasets managed by the database component. Those datasets created with a nil owner are freed. The database is closed and is removed from the list of databases maintained by the owning session object.

See also: DataSets

## Exclusive

property

```
property Exclusive : Boolean
```

Default: False

↳ Defines whether this database component has exclusive access to the database on the FlashFiler Server.

Use this property to prevent other client applications or sessions within the current client application from accessing the database on the FlashFiler Server. This property may be set only when the Connected property is False. Otherwise, an EffDatabaseError exception having error code ffdse\_DBMustBeClosed is raised.

If this property is set to True then no other client applications, or other sessions within the same client application, may open the database on the FlashFiler Server.

See also: Connected

```
procedure GetTableNames(aList : TStrings);
```

↳ Returns a list of table names in the current database.

Use this method to determine which tables already exist in the database on the FlashFiler Server. `aList` is an instance of a class inheriting from `TStrings`. This method clears `aList` before retrieving the table names. This method then populates `aList` with the name of each table found in the alias on the server.

**Note:** If the database is not already connected to the server, it is temporarily connected in order to obtain the list of tables. It is closed, immediately after retrieving the table names.

See also: `Connected`

---

**ReadOnly****property**

```
property ReadOnly : Boolean
```

Default: `False`

↳ Indicates whether this database will be used for read-only access.

Use this property to have the database component open all tables and queries in read-only mode. No inserts, updates, or deletions are allowed on its datasets. Other sessions can open the same alias in read-write mode. This property may be modified only when the database is closed. Otherwise, an `EffDatabaseError` exception having error code `ffdse_DBMustBeClosed` is raised.

---

**RestructureTable****method**

```
function RestructureTable(  
    const aTableName : TffTableName; aDictionary : TffDataDictionary;  
    aFieldMap : TStrings; var aTaskID : LongInt) : TffResult;
```

```
TffTableName = string[ffcl_TableNameSize];
```

```
ffcl_TableNameSize = 31;
```

↳ Changes the format of a table.

Use this method to change the structure of a table. Fields can be added, changed, or removed. Indexes can be added, changed, removed, internalized, or externalized. Block sizes can be changed. Any attribute of the data dictionary can be altered while preserving the existing data.



`aTableName` is the name of the table to be restructured. This parameter must have a non-empty string value. If the table does not exist then error code `DBIERR_NOSUCHTABLE` is returned. If `aTableName` has an invalid format (e.g., a file extension is specified) then error code `DBIERR_INVALIDTABLENAME` is returned.

`aDictionary` is an instance of `TffDataDictionary` defining the structure of the new table. See page 572 for more information on `TffDataDictionary`.

`aFieldMap` is a list of strings containing field assignments of the form:

```
newfieldname = oldfieldname
```

Use `aFieldMap` to specify how data is copied from the old structure to the new structure. If a field in the old structure should be copied to the same field in the new structure, `newfieldname` and `oldfieldname` will have equal values. If a field in the new table does not have an entry in the field map, it gets a null value in the new table. If a field in the old table does not appear in the field map, it is lost. If `FieldMap` is nil then all existing data is discarded and the new table is empty.

This method starts a restructure task on the FlashFiler Server. The unique ID of the task is returned in the `aTaskID` parameter. Use the value in `aTaskID` in conjunction with the `GetTaskStatus` method of `TffSession` in order to monitor the progress of the restructure.

If the restructure task is started successfully, this method returns `DBIERR_NONE`.

If a destination field in `aFieldMap` is not compatible with the source field, the restructure operation is not started and this method returns error code `DBIERR_INVALIDFLDXFORMA`.

If a field map is specified and the position of the user-defined indexes has changed relative to the other indexes, the field data cannot be preserved. The restructure operation is not started and this method returns error code `DBIERR_INVALIDRESTROP`.

If the data dictionary references user-defined indexes that do not exist, the restructure operation is not started and this method returns error code `fferrResolveTableLinks`.

If the database's `ReadOnly` property is set to `True`, the restructure operation is not started and this method returns the error code `DBIERR_READONLYDB`.

If the table having the name specified by parameter `aTableName` is already open, the restructure operation is not started and this method returns error code `DBIERR_TABLEOPEN`.

One of the common reasons for restructuring a table is to change the data type of an existing field (for example from a 16-bit integer to a 32-bit integer). However, not every data type can be converted into every other datatype. See "Converting Data Types" on page 531 for a list of the legal datatype conversions.

The following example changes the structure of the table TestTbl and updates a progress bar during the operation:

```

procedure RestructureTestTable(aDatabase : TffDatabase;
    aProgressBar : TProgressBar);
var
    Dict : TffDataDictionary;
    FldArray      : TffFieldList;
    IdxHelpers    : TffFieldIHList;
    FieldMap      : TStringList;
    TaskID        : LongInt;
    TaskStatus    : TffRebuildStatus;
    Done          : Boolean;
begin
    aProgressBar.Max := 100;
    aProgressBar.Position := 0;
    Dict := TffDataDictionary.Create(8192);
    try
        with Dict do begin
            AddField('EMP_NO', '', fftInt16, 0, 0, False, nil);
            AddField('LAST_NAME', '', fftShortString, 20, 0,
                False, nil);
            AddField('FIRST_NAME', '', fftShortString, 15, 0,
                False, nil);
            AddField('DEPT_NO', '', fftShortString, 3, 0,
                False, nil);
            {Add an external index on the LAST_NAME, FIRST_NAME
            fields}
            AddFile('Main Index File', 'MIF', 4096,
                ftIndexFile);
            FldArray[0] := 1; {Field #1}
            FldArray[1] := 2; {Field #2}
            { Use default index helpers }
            IdxHelpers[0] := '';
            IdxHelpers[1] := '';
            AddIndex('FirstName', 'by First Name', 1, 2,
                FldArray, IdxHelpers, True, True, True);
            {Add an internal index on the DEPT_NO field}
            FldArray[0] := 3; {Field #3}
            AddIndex('Dept', 'by Dept_No', 0, 1, FldArray,
                IdxHelpers, True, True, True);
        end;
    end;
end;

```

```

FieldMap := TStringList.Create;
try
  {<new fieldname>=<old fieldname>}
  with FieldMap do begin
    Add('EMP_NO=EMP_NO');
    Add('LAST_NAME=LAST_NAME');
    Add('FIRST_NAME=FIRST_NAME');
    Add('DEPT_NO=DEPT_NO');
  end;
  Check(aDatabase.RestructureTable('TestTbl',
    Dict, FieldMap, TaskID));
  {If you don't need to preserve the existing
   data, pass nil in place of the FieldMap
   parameter and the loop to check the task status
   isn't needed}
  Done := False;
  while not Done do begin
    Check(aDatabase.Session.GetTaskStatus(TaskID,
      Done, TaskStatus));
    aProgressBart.Position := rsPercentDone;
    Application.ProcessMessages;
  end;
finally
  FieldMap.Free;
end;
finally
  Dict.Free;
end;
end;

```

See also: `TffDataDictionary`, `TffSession.GetTaskStatus`

## ServerEngine

read-only property

```
property ServerEngine : TffBaseServerEngine
```

↪ References the `ServerEngine` component through which the database routes its requests.

FlashFiler is delivered with two server engine components, `TffServerEngine` and `TffRemoteServerEngine`. All requests in an application are routed through a server engine. This run-time property returns the server engine through which the session's requests are being routed.

```
property Session : TffSession
```

↪ Specifies the session component managing this database component.

Use this run-time property to obtain a handle on the session component managing this database component. The session managing the database is the session whose `SessionName` property matches the database's `SessionName` property. Until the database's `SessionName` property is set, this property is set to `nil`.

To change the session managing the database, set the database's `Connected` property to `False` and then alter the database's `SessionName` property.

See also: `Connected`, `SessionName`

```
property SessionName : string
```

Default: “[automatic]”

↪ Identifies the `SessionName` of the session managing the database component.

Use this property to connect the database component with a session component. When a database is first created, this property is set to “[automatic]”. This serves to connect the database with the automatic session.

This property can only be changed if the database is inactive. Otherwise, an `EffDatabaseError` exception having error code `ffdse_DBMustBeClosed` is raised.

The `SessionName` property must be set before the database can be opened.

See also: `Connected`, `Session`

```
property Temporary : Boolean
```

Default: False

- ↳ Identifies a database that was temporarily created in order to map an alias.

TffDatabase components do not have to be explicitly used within an application. Instead, the TffTable's DatabaseName property may refer directly to an alias on the FlashFiler Server. In this case, when the table is opened, the owning session creates and opens a temporary database component in order to provide an owning database component for the table. This automatic, alias-mapping database object is temporary and its Temporary property is set to True.

Database components explicitly placed on a form or created programmatically are not temporary databases and have this property set to False.

```
property Timeout : Longint
```

Default: -1

- ↳ Specifies the timeout for database operations and the base timeout for owned tables and queries.

This property specifies the number of milliseconds in which the FlashFiler Server must complete a database-specific operation. If the server is unable to complete the operation in the specified amount of time, an instance of EffException having error code fferrReplyTimeout is raised.

If this property has the value -1, the database uses the Timeout property of its owning session. When this property is modified, the new timeout value is sent to the FlashFiler Server and used in subsequent operations for the database.

This property is also used by owned components (i.e., queries and tables) whose Timeout property has the value -1.

See also: TffBaseDatabase.GetTimeout

---

# TffDataSet Class

The TffDataSet class is the functional ancestor to the FlashFiler TffTable, and TffQuery components, in the sense that it provides basic functionality such as dataset navigation and retrieval and modification of field data. Do not create an instance of this class. Instead, use TffTable or TffQuery.

The TffDataSet class is described here because it provides the layer of common functionality available in all FlashFiler dataset components.

## Hierarchy

TDataSet (VCL)

    TffDataSet (FFDB)

## Properties

Active	CursorID	FilterTimeout
ActiveBuffer	Database	ObjectView
AggFields	DatabaseName	ServerEngine
AutoCalcFields	Dictionary	Session
BlockReadSize	Filter	SessionName
BOF	Filtered	SparseArrays
Bookmark	FilterEval	Timeout
CanModify	FilterOptions	Version

## Methods

AddFileBlob	CreateBlobStream	IsSequenced
Append	DeleteTable	Locate
AppendRecord	EmptyTable	Lookup
BookmarkValid	GetCurrentRecord	PackTable
Cancel	GetFieldData	Post
CheckBrowseMode	GetRecordBatch	RenameTable
ClearFields	GetRecordBatchEx	RestructureTable
Close	GetTimeout	SetTableAutoIncValue
ControlsDisabled	GotoCurrent	Translate
CompareBookmarks	InsertRecordBatch	

## Events

AfterCancel  
AfterClose  
AfterDelete  
AfterEdit  
AfterInsert  
AfterOpen  
AfterPost  
AfterRefresh  
AfterScroll

BeforeCancel  
BeforeClose  
BeforeDelete  
BeforeEdit  
BeforeInsert  
BeforeOpen  
BeforePost  
BeforeRefresh  
BeforeScroll

OnCalcFields  
onDeleteError  
OnEditError  
OnFilterRecord  
OnNewRecord  
OnPostError  
OnServerFilterTimeout

## Reference Section

### Active

### property

```
property Active : Boolean
```

Default: False

↪ Defines whether the dataset is open or closed.

Use Active to open or close a dataset, as well as to identify whether the dataset is open or closed. Alternatively, you may use the Open and Close methods to open and close the table, respectively.

If this property is set to True, the dataset is open otherwise it is closed. Set this property to True in order to open the dataset. In order for a dataset to open, its DatabaseName and TableName properties must have legitimate values.

Activating a dataset requires a connection to the FlashFiler Server. The owning client component establishes a new connection to the server if one is not already present.

Set this property to False in order to close the dataset. Doing so notifies the FlashFiler Server that the client no longer needs the dataset. The client, session, and database managing the table remain connected to the server.

See also: AfterClose, AfterOpen, BeforeClose, BeforeOpen, Close, Open, DatabaseName, TableName

### ActiveBuffer

### method

```
function ActiveBuffer : PChar;
```

↪ Points to a buffer of characters containing the active record.

The ActiveBuffer function can be used to retrieve the physical contents of the record buffer. The buffer represents the data as it is on the server and in the file. Therefore, any StDate or StTime fields will appear in their native format. It is normally not necessary to read the ActiveBuffer directly.

See also: RecordSize



```
function AddFileBlob(const aField : Word;
    const aFileName : TffFullFileName) : TffResult;

TffFullFileName = string[255];
```

✚ Inserts a reference to a file into a BLOB field.

FileBLOBs are FlashFiler's unique way of storing large quantities of data in minimal space. FileBLOBs are useful when a single file on the server contains data you would like to keep in a database, but without replicating the data in your BLOB file.

The `aField` parameter references the number of the BLOB field. Fields are numbered starting at 1 just as they are in the BDE.

The `aFileName` parameter is a string containing the full server relative path to the BLOB data file. `aFileName` must be a non-empty string.

**AfterCancel****event**

```
property AfterCancel : TDataSetNotifyEvent
```

✚ Defines an event handler that is called when a cancel operation is complete.

The `AfterCancel` event is triggered after making a call to the dataset's `Cancel` method. The `AfterCancel` event is called after the changes to the active record are removed and the `DataSet`'s state has been updated to `dsBrowse`. Use this event if you need to take specific action once a cancel operation is complete.

See also: `BeforeCancel`, `Cancel`

**AfterClose****event**

```
property AfterClose : TDataSetNotifyEvent
```

✚ Defines an event handler that is called after a dataset is closed.

The `AfterClose` event is triggered after the dataset's `Active` property is set to `False`. Use the `AfterClose` event if you need to take specific action once a dataset is closed.

See also: `Active`, `AfterOpen`, `BeforeClose`, `BeforeOpen`, `Close`

## AfterDelete

event

```
property AfterDelete : TDataSetNotifyEvent
```

↳ Defines an event handler that is called after a record is removed from the dataset.

The AfterDelete event is triggered by a call to the dataset's Delete method. This event is fired after the delete operation is complete and the dataset's state has been updated to dsBrowse. Use the AfterDelete event if you need to take specific action once a record has been deleted.

See also: BeforeDelete, Delete

## AfterEdit

event

```
property AfterEdit : TDataSetNotifyEvent
```

↳ Defines an event handler that is called after a dataset enters dsEdit mode.

The AfterEdit event is triggered when a call to the Edit method is made. The AfterEdit event is called after the dataset enters dsEdit mode. Use the AfterEdit event if you need to take specific action once a dataset enters dsEdit mode.

See also: AfterPost, BeforeEdit, BeforePost, Edit

## AfterInsert

event

```
property AfterInsert : TDataSetNotifyEvent
```

↳ Defines an event handler that is called after a dataset enters dsInsert mode.

The AfterInsert event is triggered after a new record is inserted or appended to the dataset. The AfterInsert event is called after the record has been added and the dataset's state has been updated to dsInsert. Use the AfterInsert event if you need to take specific action once a record has been inserted.

See also: Append, BeforeInsert, Insert

## AfterOpen

event

```
property AfterOpen : TDataSetNotifyEvent
```

↳ Defines an event handler that is called after a dataset is opened.

The AfterOpen event is triggered when the dataset has completed the open process. A table is opened when its Active property is set to True. Use the AfterOpen event if you need to take specific action after the dataset is opened.

See also: Active, AfterClose, BeforeClose, BeforeOpen, Open

## AfterPost

event

```
property AfterPost : TDataSetNotifyEvent
```

↳ Defines an event handler called after a record has been posted.

The AfterPost event is triggered after a record modification or addition is sent to the server. The AfterPost event is called after the record has been posted and the dataset's state has been updated to dsBrowse. Use the AfterPost event if you need to take specific action once a record has been posted.

See also: AfterEdit, BeforeEdit, BeforePost, Edit, Insert, Post

## AfterRefresh

event

```
property AfterRefresh : TDataSetNotifyEvent
```

↳ Defines an event handler called after a dataset has been refreshed.

The AfterRefresh event is triggered after a refresh operation takes place. A refresh operation is normally initiated in code by calling the Refresh method or by a connected data-aware control that requests an update of record data. Use the AfterRefresh event if you need to take specific action when data is refreshed.

See also: BeforeRefresh, Refresh

## AfterScroll

event

```
property AfterScroll : TDataSetNotifyEvent
```

↳ Defines an event handler that is called after the dataset scrolls to another record.

The AfterScroll event is triggered after a scroll operation takes place. A scroll operation is normally initiated by a call to FindFirst, FindKey, FindLast, FindNext, FindPrior, First, Last, Locate, MoveBy, Next, or Prior. This event is fired after all operations against the dataset are complete. Use the AfterScroll event if you need to take specific action after the cursor changes position.

See also: BeforeScroll

```
property AggFields : TFields
```

↳ Contains the aggregate fields.

FlashFiler does not support aggregate fields, therefore this property always returns an instance of a TFields object that contains no fields.

See also: Fields

---

**Append****method**

```
procedure Append;
```

↳ Adds a new empty record to the end of a dataset.

The Append method moves the cursor to the end of the file and prepares a new record for insertion. After Append completes, the dataset is in Insert mode. Since FlashFiler reuses empty space within a table, the Append method provides no benefit over the Insert method. In fact, especially when using indexes, use the Insert method instead of Append.

See also: AppendRecord, Insert, Post

---

**AppendRecord****method**

```
procedure AppendRecord(const Values : array of const);
```

↳ Posts a complete record to the end of a dataset.

The AppendRecord method moves the cursor to the end of the file and inserts a complete record to the dataset. The AppendRecord operation then posts the new record, leaving the dataset in Browse mode. Since FlashFiler reuses empty space within a table, the Append method provides no benefit over the Insert method. In fact, especially when using indexes, use the Insert method instead of Append.

Values is an array of constants that are used as field values in the new record. The values must be specified in field order. The array does not need to specify a value for every field in the table. Instead, it must specify a value for each field until a series of null values is all that remains for the remaining portion of the record.

As an example, look at the record structure of a table named ZipCode in the following table:

FieldName	DataType	Size
Zipcode	ShortString	9
City	ShortString	30
State	ShortString	20
County	Shortstring	30
AreaCode	Shortstring	3
Longitude	Double	
Latitude	Double	

The following example inserts four records into the ZipCode table:

```
Table1.AppendRecord(['66064', 'Osawatomie', 'Kansas',  
    'Miami', '913']);  
Table1.AppendRecord(['73127', 'Oklahoma City', 'Oklahoma',  
    null, '405']);  
Table1.AppendRecord('80863', 'Woodland Park', 'Colorado',  
    'Teller'];  
Table1.AppendRecord(null, 'Colorado Springs', 'Colorado',  
    'El Paso', '719');
```

Notice that empty fields within the array are marked null, and the last two fields in the record are not specified.

See also: Append, InsertRecord, Post, SetFields

## AutoCalcFields

property

property AutoCalcFields : Boolean

Default: True

☞ Specifies how the OnCalcFields event should be triggered.

A dataset's OnCalcFields event is triggered under the following circumstances:

- The dataset is opened.
- The cursor position within the dataset changes.
- A field within the current record is modified.
- The dataset is placed into dsEdit state via a call to the Edit method.

Depending upon how often these circumstances occur, the triggering of OnCalcFields can become an enormous bottleneck. Use this property to reduce the number of times the OnCalcFields event is called, thereby increasing the perceived responsiveness of the application. Set this property to False to prevent the OnCalcFields event from being called for individual field modifications. Set this property back to True when it is acceptable for the OnCalcFields event to once again be called for individual field modifications.

See also: OnCalcFields

---

**BeforeCancel****event**

```
property BeforeCancel : TDataSetNotifyEvent
```

↳ Defines an event handler that is called before changes to the active record are aborted.

The BeforeCancel event is triggered when calling the dataset's Cancel method. The event is called before changes to the current record are removed, so they are still available for inspection via the Field.As\* and Field.Value methods. Use the BeforeCancel event if you need to take specific action before the changes to the active record are removed. Since the active record's contents are about to be changed, it does not make sense to have the event handler modify the record.

See also: AfterCancel, Cancel

---

**BeforeClose****event**

```
property BeforeClose : TDataSetNotifyEvent
```

↳ Defines an event handler that is called before a dataset is closed.

The BeforeClose event is triggered when the dataset's Active property is set to False. Use the BeforeClose event if you need to take specific action before a dataset is finally closed.

See also: Active, AfterClose, AfterOpen, BeforeOpen, Close, Open

---

**BeforeDelete****event**

```
property BeforeDelete : TDataSetNotifyEvent
```

↳ Defines an event handler that is called before the active record is removed from the dataset.

The BeforeDelete event is triggered when calling the dataset's Delete method. This event is fired before any action is taken to remove the active record. Use the BeforeDelete event if you need to take specific action before a record is deleted.

See also: AfterDelete, AfterInsert, BeforeInsert, Delete

## BeforeEdit

event

```
property BeforeEdit : TDataSetNotifyEvent
```

↳ Defines an event handler that is called before a dataset enters dsEdit mode.

The BeforeEdit event is triggered when calling the dataset's Edit method. The BeforeEdit event is called just before the dataset enters dsEdit mode, so modifications of the record within the event are not possible. Use the BeforeEdit event if you need to take specific action before a dataset is edited.

See also: AfterEdit, AfterPost, BeforePost, Edit

## BeforeInsert

event

```
property BeforeInsert : TDataSetNotifyEvent
```

↳ Defines an event handler that is called before a dataset enters dsInsert mode.

The BeforeInsert event is triggered before a new record is inserted or appended to the dataset. The BeforeInsert event is fired before any state changes are made. Use the BeforeInsert event if you need to take specific action before the new record is inserted.

See also: AfterInsert, AfterDelete, BeforeDelete, Insert

## BeforeOpen

event

```
property BeforeOpen : TDataSetNotifyEvent
```

↳ Defines an event handler that is called before a table is opened.

The BeforeOpen event is triggered when the dataset's Active property is set to True. Use the BeforeOpen event if you need to take specific action before a dataset is opened.

See also: Active, AfterClose, AfterOpen, BeforeClose, Open

## BeforePost

event

```
property BeforePost : TDataSetNotifyEvent
```

↳ Defines an event handler that is called before the record is posted.

The BeforePost event is triggered before a record is posted to the server. The BeforePost event is called before the DataSet's state has been updated to dsBrowse. Use the BeforePost event if you need to take specific action before a record change/addition is sent to the server.

See also AfterEdit, AfterPost, BeforeEdit, Post

---

```
property BeforeRefresh : TDataSetNotifyEvent
```

↳ Defines an event handler called before a dataset is refreshed.

The BeforeRefresh event is triggered before a refresh operation takes place. A refresh operation is normally initiated in code by calling the refresh method, or by a connected data-aware control that requests an update of record data. Use the BeforeRefresh event if you need to take specific action before data is refreshed.

See also: AfterRefresh, Refresh

---

```
property BeforeScroll : TDataSetNotifyEvent
```

↳ Defines an event handler that is called before the dataset scrolls to another record.

The BeforeScroll event is triggered before a scroll operation takes place. A scroll operation is normally initiated by a call to FindFirst, FindKey, FindLast, FindNext, FindPrior, First, Last, Locate, MoveBy, Next, or Prior. This event is fired before any operation against the table is carried out. Use the BeforeScroll event if you need to take specific action before the cursor changes position.

See also: AfterScroll

---

```
property BlockReadSize : Integer
```

↳ Specifies the number of records to be read before updating connected controls.

The BlockReadSize property can be set when a quick scan of the dataset is required without updating connected controls.

Updating controls connected to a dataset, especially grids, is a time consuming process. Therefore it is often necessary to scan through a dataset without updating any connected controls. Setting this property to a value greater than 0 will put the dataset's state in BlockRead mode, and keep connected controls from being updated. Setting this property to zero will put the dataset's state back in dsBrowse mode.



The following example scans an OrderDetail dataset and returns the total cost of the order without updating any connected controls:

```
function GetOrderCost(OrderDetails : TffTable) : Double;
begin
    Result := 0;
    with OrderDetails do
        BlockReadSize := 1;
        try
            Result := Result + FieldByName('Cost').Value;
        finally
            BlockReadSize := 0;
        end;
end;
```

See also: Next, State

## BOF

read-only property

7

```
property Bof : Boolean
```

↳ Returns True when the cursor is positioned at the beginning of a dataset.

Use this property to determine if the dataset's cursor is positioned to the Beginning Of File (BOF). If the value of this property is True then the cursor is positioned at BOF. BOF can also be True if a dataset or range is empty.

**Note:** When the cursor is placed on the second record and a call to Prior is made, the BOF property will be left at False. This is consistent with the BDE, and only a second call to Prior or a call to First will set BOF to True.

See also: EOF, First

## Bookmark

property

```
property Bookmark : TBookmarkStr
```

```
TBookmarkStr = string;
```

↳ Returns the bookmark for the current record or sets the cursor to the specified bookmark.

Use this property to retrieve a bookmark for the dataset's current record. Once retrieved, it is the caller's responsibility to free the bookmark with a call to the dataset's FreeBookmark method.

Use the bookmark as a placeholder for the current record. After positioning to another record or performing other operations, assign the saved bookmark to this property. This repositions the dataset's cursor to the record identified by the bookmark.

See also: BookmarkValid, FreeBookmark, GetBookmark, GotoBookmark

## BookmarkValid

method

```
function BookmarkValid(aBookmark : TBookmark) : Boolean;
```

↳ Returns True if the specified bookmark is valid.

Use this method to verify a bookmark. This property returns True if the bookmark is valid. aBookmark is the bookmark to validate. Actions that invalidate a bookmark include changing the current index of the dataset and deleting the record associated with the bookmark.

See also: Bookmark, FreeBookmark, GetBookmark, GotoBookmark

## Cancel

method

```
procedure Cancel;
```

↳ Discards modifications to the active record.

A call to the dataset's Cancel method causes all current modifications to the record to be discarded, resetting the dataset's State to dsBrowse. Use Cancel to undo changes to a record based upon the request of a user.

See also: Post, State

## CanModify

read-only property

```
property CanModify : Boolean
```

↳ Returns True if the dataset may be modified.

Use this property to determine if records may be inserted, deleted, or modified. If the dataset is modifiable, this property has the value True. If the dataset is read-only, or if it is inactive then this property has the value False.

```
procedure CheckBrowseMode;
```

↳ Automatically records changes to a record based on a set of internal criteria.

A call to `CheckBrowseMode` posts or cancels changes to a record based on the following criteria:

Dataset State	Action
dsEdit, dsInsert	Calls <code>UpdateRecord</code> . If the record has been modified, it will be posted. Otherwise a call to the cancel method will be made.
dsSetKey	A call to post will be made.

If the dataset is inactive, an exception is raised.

See also: `State`

7

**ClearFields****method**

```
procedure ClearFields;
```

↳ Reverts a record's fields to their initialized state.

Use the `ClearFields` to discard all data in a record's fields. Once the data is cleared, the record is reset to its default values and any connected controls are told to retrieve the new data. If the dataset is not in `dsEdit`, `dsInsert`, or `dsSetKey` mode an exception is raised. If the dataset is not in `dsSetKey` mode, the operation recalculates any calculated fields.

See also: `State`

**Close****method**

```
procedure Close;
```

↳ Closes the dataset.

Use this method to close the dataset. This method sets the dataset's `Active` property to `False`. Once the close operation completes, data may not be read from or written to the dataset. The `BeforeClose` event is called before the dataset is closed. Once the dataset closes, the `AfterClose` event is called.

See also: `Active`, `AfterClose`, `BeforeClose`, `Open`

```
function CompareBookmarks(  
    Bookmark1, Bookmark2 : TBookmark) : integer;
```

↪ Compares two bookmarks for equality.

This function is used to compare two bookmarks. Bookmark1 is the first bookmark to compare. Bookmark2 is the second bookmark to compare. Either parameter may be specified as nil.

The two bookmarks are compared and if they are equal the result will be zero. Any other value means the bookmarks are unequal or that they do not belong to records in the same dataset.

Internally, the CompareBookmarks method is composed of two steps. The first step attempts a comparison on the client side. If either bookmark is nil, the method returns immediately. If both bookmarks are non-nil, then the bookmarks are sent to the FlashFiler Server for comparison. If either bookmark is invalid, an EffDatabaseError exception having error code DBIERR\_INVALIDBOOKMARK is raised.

See also: Bookmark, BookmarkValid

```
function ControlsDisabled : Boolean;
```

↪ Returns True when connected data-aware controls are blocked from receiving changes to the dataset.

Use this method to determine if a call to DisableControls is in effect. If data-aware controls connected to this dataset are not receiving updates from the dataset, this method returns True.

Since calls to the DisableControls and EnableControls may be nested, there are cases where a single call to EnableControls only decrements the internal count instead of actually updating the connected controls. The following example uses ControlsDisabled to force the connected controls to revert to their normal state of receiving updates.

```
procedure ForceEnableControls(Dataset : Tdataset);  
begin  
    while Dataset.ControlsEnabled do  
        Dataset.EnableControls  
    end;
```

See also: DisableControls, EnableControls

```
function CreateBlobStream(
    aField : TField; aMode : TBlobStreamMode) : TStream;

TBlobStreamMode = (bmRead, bmWrite, bmReadWrite);
```

↪ Creates a stream for reading from and writing to a BLOB field.

Use this method to obtain a stream allowing access to a BLOB field within the current record. aField is a BLOB field within the current record. aMode specifies the access given to the stream. If aMode is set to bmRead then the stream may read from but not write to the BLOB. If aMode is bmWrite then the stream may write to but not read from the BLOB. If aMode is bmReadWrite then the stream may read from or write to the BLOB.

**Note:** The returned BLOB stream is of type TffBLOBStream, not TBLOBStream.

If you need to create a BLOB stream for a FlashFiler BLOB field, use this routine in preference to calling the TffBLOBStream Create constructor and explicitly creating a BLOB stream.

BLOB streams may not be created while a filter is in effect. Doing so results in the raising of an EffDatabaseError exception having error code ffdse\_BLOBFltNoFldAccess.

If the dataset is read-only and aMode is specified as bmWrite or bmReadWrite, a EffDatabaseError exception having error code ffdse\_BLOBTblNoEdit is raised.

See also: TffBLOBStream

```
property CursorID : TffCursorID

TffCursorID = type TffWord32;

TffWord32 = type DWORD;
```

↪ Returns the unique identifier assigned by the server engine to the dataset.

The read-only CursorID property is used by the dataset when communicating with the server engine. Before a dataset is made active, this property has a value of 0. This property is required when making direct calls into a server engine.

```
property Database : TffBaseDatabase
```

↪ References the database component managing this dataset.

Use this property to obtain a handle on the database handling this dataset. The database managing the dataset is the database whose `DatabaseName` property matches the dataset's `DatabaseName` property. Until the dataset's `DatabaseName` property is set, this property is set to `nil`.

If the value of the `DatabaseName` property is an actual alias on the FlashFiler Server, the database component referenced by this property is a temporary database component.

To change the database managing the dataset, set the dataset's `Active` property to `False` and then alter the dataset's `DatabaseName` property.

See also: `Active`, `DatabaseName`, `TffDatabase.Temporary`

```
property DatabaseName : string
```

Default: Empty string

↪ Identifies the `DatabaseName` of the database component managing this dataset or an alias on the FlashFiler Server.

Use this property to connect the dataset to a database component or to a specific alias on the FlashFiler Server. The validity of this property is checked only when the dataset is opened. If the value of this property is an alias on the server, a temporary database component is created behind the scenes.

To obtain the database component managing the table, including the temporary database component created when this property is set to the name of an alias, use the `Database` property. This property may be changed only when the `Active` property is set to `False`.

See also: `Active`, `Database`, `TffDatabase.Temporary`

```
procedure Delete;
```

✚ Deletes the current record of the dataset.

Use this method to delete the current record of the dataset. When `Delete` is called, `TffDataSet` posts any outstanding changes to the prior record, calls the `BeforeDelete` event, deletes the record, frees the resources associated with the record, positions to the next active record in the dataset, puts the dataset in `dsBrowse` mode, and calls the `AfterDelete` event.

If `Delete` is called and there is no current record in the dataset, a `EffDatabaseError` exception having error code `DBIERR_NOCURREC` is raised.

Use this method with caution. A deleted record may not be recovered. Once the record has been deleted, it is added to the table's chain of deleted records. The chain of deleted records is used to recycle the space occupied by deleted records. When a new record is inserted into the table, the FlashFiler Server takes the first record from the chain and stores the new record in the space once occupied by the deleted record.

See also: `AfterDelete`, `BeforeDelete`

---

**DeleteTable****method**

```
procedure DeleteTable;
```

✚ Deletes the physical table on the FlashFiler Server.

Use this method to remove the table's files from the directory representing the database alias. This operation requires exclusive access to the physical table. If the table is already opened, a `EffDatabaseError` exception having error code `DBIERR_TABLEOPEN` is raised.

---

**Dictionary****property**

```
property Dictionary : TffDataDictionary
```

✚ Provides access to the dataset's data dictionary object.

After opening a dataset, use this property to obtain information about the structure of the dataset.

See also: `TffDataDictionary`

```
procedure Edit;
```

↳ Enables editing of the current record in the dataset.

Use the Edit method to change field values of the current record in the dataset. If the dataset is not currently positioned on a record, calling Edit has the same effect as calling the Insert method. When you call the Edit method, TffDataSet verifies that any changes to the current record have been posted, calls the BeforeEdit event, retrieves the current record from the FlashFiler Server, puts the dataset into dsEdit state, broadcasts the state change to the associated components, and then calls the AfterEdit event.

At this point, the application may change the fields within the current record. Once the changes have been made, the application may call Post to send the changes to the FlashFiler Server or Cancel to abort the changes just made.

See also: AfterEdit, BeforeEdit, Cancel, Insert, Post

## EmptyTable

method

```
procedure EmptyTable;
```

↳ Empties the dataset of all records.

Use EmptyTable to remove all records from the dataset. If the dataset contains an auto-increment field, it will be reset to one (i.e., the first record inserted to the dataset has its autoinc field set to one). This operation requires exclusive access to the physical files. If the exclusive access cannot be obtained, an EfficDatabaseError exception having error code DBIERR\_NEEDEXCLACCESS is raised. If the files are marked as read-only, an EfficDatabaseError exception having error code DBIERR\_TABLEREADONLY is raised.

**Note:** This method does not respect any ranges or filters that may be in effect. Every record in the table is deleted.



`property Filter : string`

Default: Empty string

↳ Defines the filter to apply to the dataset.

When a filter is active, only the records that pass the filter criteria will be available to the application. Filters may be evaluated on the client or on the server. Evaluating filters on the server provides much better performance because the server sends to the client only those records that have matched the filter. Client-side filter evaluation results in all records being passed to the client, whether or not they match the filter.

See the VCL help file for more details on how to create the filter string.

See also: Filtered, FilterEval, OnFilterRecord

`property Filtered : Boolean`

Default: False

↳ Defines whether the filter is to be applied to the dataset.

When Filtered is True, the dataset observes any filter conditions defined in the Filter property or compiled into the OnFilterRecord event. When Filtered is False, any filter conditions are ignored.

See also: Filter, OnFilterRecord

## FilterEval

property

```
property FilterEval : TffFilterEvaluationType  
TffFilterEvaluationType = (ffeLocal, ffeServer);
```

Default: ffeServer

↪ Indicates whether the filter is to be evaluated on the server or the client.

Performance is best when the filter is evaluated on the server. However, if the filter is on a large table and not many records match the filter then performance may be slow and other clients may be blocked. Use the OnServerFilterTimeout event to monitor and handle these problems.

Performance is slower when the filter is evaluated on the client. The reason for this is each record must be transferred from the server to the client for evaluation.

If you have a handler defined for OnFilterRecord and FilterEval is set to ffeServer, OnFilterRecord will be called only for those records matching the Filter property.

See also: FilterResync, FilterTimeout, OnServerFilterTimeout

## FilterOptions

property

```
property FilterOptions : TFilterOptions  
TFilterOptions = set of TFilterOption;  
TFilterOption = (foCaseInsensitive, foNoPartialCompare);
```

Default: empty set

↪ Defines the behavior of the filter.

FilterOptions defines a set that defines how the Filter property will be used when Filtered is set to True. The filter options set includes values that specify case sensitivity and partial compare options. More information on these options can be found in the VCL documentation.

See also: Filter, Filtered, FilterEval, OnFilterRecord

```
property FilterTimeout : TffWord32
```

```
TffWord32 = type DWORD;
```

Default: 500

- ↪ Specifies the number of milliseconds in which the server must find the next record in a server-side filter.

FilterTimeout is the number of milliseconds the server has in which to retrieve a record when a server-side filter is in effect. This property has no effect on evaluation of client-side filters.

If the server does not find a record matching the Filter property within the allotted time, the OnServerFilterTimeout event is called.

See also: Filter, FilterEval, OnServerTimeout

**GetCurrentRecord****method**

```
function GetCurrentRecord(aBuffer : PChar) : Boolean; override;
```

- ↪ Returns the current record.

Use this method to obtain a copy of the current record. aBuffer is a buffer into which the record is copied and must be as large as the dataset's record size. Obtain the record size via the dataset's Dictionary. The data returned in the buffer is in physical, not logical, format.

See also: Dictionary, TffDataDictionary

**GetFieldData****method**

```
function GetFieldData(
    aField : TField; aBuffer : Pointer): Boolean; override;
```

- ↪ Returns the value of the specified field object for the current record.

Use this method to obtain the value of a specific field from the current record. aField is the field whose value is to be retrieved. aBuffer is a buffer into which the field value is copied. It must be large enough to hold the largest value possible for the field (e.g., an ANSI NULL String field capable of holding 50 characters requires a buffer capable of holding 51 characters). If the field's value is null, this method returns False. Otherwise, this method copies the field's value into aBuffer and returns True.

```
function GetRecordBatch(
    RequestCount : Longint; var ReturnCount : Longint;
    pRecBuff : Pointer) : TffResult;
```

👉 Reads a batch of records from the specified table.

Use this method to obtain a set of records from the FlashFile Server with a single message. Reducing message counts is a key factor in improving FlashFile application performance over a network. RequestCount is the number of records to retrieve. pRecBuff is a pointer to a buffer capable of holding the number of records specified in RequestCount. The maximum number of records that may be retrieved in one batch is 65,500 divided by the record's physical size (i.e., TffDataDictionary.RecordLength). If more records are requested than can be handled, this function raises an EfficDatabaseError exception having error code DBIERR\_ROWFETCHLIMIT.

The ReturnCount parameter is filled with the number of records retrieved and copied into pRecBuff. If ReturnCount is less than RequestCount, the table cursor hit the end of table.

⚠ **Caution:** GetRecordBatch returns the physical representation of each record. You cannot use field objects to access the data in each field. Instead, you must use the methods of the dataset's data dictionary object (the Dictionary property) to extract the data from each field. The data dictionary knows about null fields and where to find the field data in the record.

The following example uses this function to read through all the records in a table. It checks the third field (a Boolean value) and counts the number of True and False values:

```
var
    IsNull : Boolean;
    Eof : Boolean;
    InCount : LongInt;
    OutCount : LongInt;
    Ratio : Double;
    RecLen : LongInt;
    RequestCount : LongInt;
    BlockSize : LongInt;
    ReturnCount : LongInt;
    i : LongInt;
    Inside : Boolean;
    Dict : TffDataDictionary;
    pOneRec : PffByteArray;
    pRecBuff : PffByteArray;
    pCurrentRec : PffByteArray;
    Error : TffResult;
    Start : TDateTime;
```

```

begin
    Start := Now;
    Table.First;
    Eof := False;
    InCount := 0;
    OutCount := 0;
    while not Eof do begin
        RequestCount := 3000;
        Dict := Table1.Dictionary;
        RecLen := Dict.RecordLength;
        BlockSize := RequestCount * RecLen;
        FFGetZeroMem(pOneRec, RecLen);
        FFGetZeroMem(pRecBuff, BlockSize);
        try
            Error := Table.GetRecordBatch(Table1, RequestCount,
                ReturnCount, pRecBuff);
            if Error = DBIERR_NONE then begin
                if ReturnCount < RequestCount then
                    Eof := True;
                pCurrentRec := pRecBuff;
                for i := 0 to pred(ReturnCount) do begin
                    Move(pCurrentRec^, pOneRec^, RecLen);
                    with Dict do begin
                        GetRecordField(2, pOneRec, IsNull, @Inside);
                    end;
                    if Inside then
                        inc(InCount)
                    else
                        inc(OutCount);
                    {move pCurrentRec to next record}
                    inc(pChar(pCurrentRec), RecLen);
                end;
            end;
        finally
            FFFreeMem(pOneRec, RecLen);
            FFFreeMem(pRecBuff, BlockSize);
        end;
    end;
    PiLabel.Caption := IntToStr(InCount) + '/' + IntToStr
        (OutCount) + ' = ' +
        FloatToStr(InCount /
            (1.0 * OutCount));
    Label3.Caption := DateTimeToStr(Now - Start);
end;

```

```
function GetRecordBatchEx(  
    RequestCount : Longint; var ReturnCount : Longint;  
    pRecBuff : Pointer; var Error : TffResult) : TffResult;  
  
TffResult = longint;
```

↪ Reads a batch of records from the specified table.

The `GetRecordBatchEx` method closely resembles the `GetRecordBatch` method with one additional parameter. The `Error` parameter returns the status of the internal `GetNextRecord` operation that is done by the server engine.

See also: `GetRecordBatch`

```
function GetTimeout : Longint;
```

↪ Retrieves the current timeout value for the dataset.

The `GetTimeout` method retrieves the actual timeout being used by a dataset. When the dataset's `Timeout` property is set to `-1`, this method returns the `Timeout` of the database component managing the dataset. The dataset's `Timeout` property is returned when it contains a value other than `-1`. For this reason, use `GetTimeout` instead of reading the `Timeout` property directly.

See also: `Timeout`

```
procedure GotoCurrent(aDataSet : TffDataSet);
```

↪ Positions this dataset's cursor to the same record as the passed dataset.

Use this method to position the dataset's cursor to the same position the cursor in another dataset. `aDataSet` is the dataset whose cursor position is to be adopted. Both datasets must have matching `DatabaseName` and `TableName` properties. In other words, they must reference the same physical table at the server.

See also: `DatabaseName`, `TableName`

```
procedure Insert;
```

↳ Prepares the dataset for the addition of a new record.

Use this method to add new records to the dataset. When `Insert` is called, `TffDataSet` calls the `BeforeInsert` event, prepares an empty record, fills in default values as defined in the table's data dictionary, and calls the `after Insert` event. At this point, you may set the values of fields within the record. Once the fields have been assigned their values, use the `Post` method to add the record to the table. To abort the new record, use the `Cancel` method.

When the new record is posted, FlashFiler Server checks the table's chain of deleted records to see if it can re-use the space occupied by a deleted record. If the chain has at least one record, the server uses the space occupied by the first record in the chain. If the chain is empty, the FlashFiler Server looks for room in the last block of the table's data file. If no room is available, FlashFiler Server allocates another block to the data file.

When the record is inserted, the FlashFiler Server verifies, on a per index basis, the key fields in the record do not conflict with the key fields of another record already in the table. If a conflict is found, the insert is aborted, and an `EffDatabaseError` exception having error code `DBIERR_KEYVIOL` is raised.

The following example illustrates the insertion of a record into a table:

```
with aOrderShipTable do
  try
    Insert;
    FieldByName('CustomerID').asInteger := aCustomerID;
    FieldByName('OrderID').asInteger := anOrderID;
    FieldByName('DateShipped').asDateTime := Now;
    FieldByName('ShipMethod').asInteger := aShipMethodID;
    Post;
  except
    if aOrdersTable.Database.InTransaction then
      aOrdersTable.Database.Rollback;
    end;
```

See also: `AfterInsert`, `BeforeInsert`, `Cancel`, `Post`

```
function InsertRecordBatch(  
    Count : Longint; pRecBuff : Pointer;  
    Errors : PffLongintArray) : TffResult;  
  
PffLongintArray = ^TffLongintArray;  
TffLongintArray = array [0..16382] of longint;
```

↳ Inserts a batch of records into the dataset.

Use `InsertRecordBatch` to insert a batch of records with a single message to the FlashFiler Server. Reducing message counts is a key factor in improving FlashFiler application performance over a network. `Count` is the number of records to be inserted. `pRecBuff` is a pointer to a buffer containing the records to be inserted. The records must already be initialized and filled with data.

`Errors` is a pointer to an array of long integers where each array element corresponds to a record in the `pRecBuff` buffer. The server fills each element of the errors array based upon whether the corresponding record was inserted successfully. A record successfully inserted has its `Errors` array element set to `DBIERR_NONE`. A record that failed to insert has its `Errors` array element set to a non-zero error code. The application must check the `Errors` array to determine if any records failed to insert.

Note that `InsertRecordBatch` requires raw, physical records. The VCL `TField` objects cannot be used to set up the field values in each record. Instead, use the dataset's data dictionary object (the `Dictionary` property) to populate the fields. The dictionary has methods to initialize a record (i.e., set all fields to null) and to set individual fields within the record.

The following example shows how to insert a batch of records:

```
var  
    Count : LongInt;  
    pOneRec : PffByteArray;  
    pRecBuff : PffByteArray;  
    pCurrentRec : PffByteArray;  
    BlockSize : LongInt;  
    pErrors : PffLongIntArray;  
    RecLen : LongInt;  
    i : LongInt;  
    X, Y : Double;  
    Inside : Boolean;  
    Dict : TffDataDictionary;
```



```

begin
    Count := 3000;
    Dict := Table1.Dictionary;
    RecLen := Dict.RecordLength;
    BlockSize := Count * RecLen;
    FFGetZeroMem(pOneRec, RecLen);
    FFGetZeroMem(pRecBuff, BlockSize);

    try
        {fill some records}
        pCurrentRec := pRecBuff;
        for i := 0 to pred(Count) do begin
            with Dict do begin
                {initialize a record}
                InitRecord(pOneRec);
                X := 2.0 * (Random - 0.5);
                Y := 2.0 * (Random - 0.5);
                Inside := (X*X + Y*Y) <= 1.0;
                SetRecordField(0, pOneRec, @X);
                SetRecordField(1, pOneRec, @Y);
                SetRecordField(2, pOneRec, @Inside);
                {move those bytes into the array}
                Move(pOneRec^, pCurrentRec^, RecLen);
                inc(PChar(pCurrentRec), RecLen);
            end;
        end;
        {allocate space for the error array}
        FFGetZeroMem(pErrors, Count * sizeof(LongInt));
        try
            if Table1.InsertRecordBatch(Count, pRecBuff,
                pErrors) = DBIERR_NONE then;
                ShowMessage('Success');
            finally
                FFFreeMem(pErrors, Count * sizeof(LongInt));
            end;
        finally
            FFFreeMem(pRecBuff, BlockSize);
            FFFreeMem(pOneRec, RecLen);
        end;
    end;
end;

```

```
function IsSequenced : Boolean; override;
```

↳ Returns True if the dataset supports sequence numbers.

FlashFile does not support sequence or record numbers. This method always returns False.

See also: RecNo

```
function Locate(  
    const aKeyFields : string; const aKeyValues : Variant;  
    aOptions : TLocateOptions) : Boolean;
```

```
TLocateOptions = set of TLocateOption;
```

```
TLocateOption = (loCaseInsensitive, loPartialKey);
```

↳ Places the dataset cursor on the record that matches the criteria.

Use this method to find a record that matches a certain set of key values. `aKeyFields` is a list of field names delimited by semi-colons (e.g., "ID;SSN;BirthDate"). If the dataset has an index that matches the fields specified in `aKeyFields`, the index is used to search for the record. Otherwise, a filter is used which can reduce performance.

`aKeyValues` must contain one value per field in `aKeyFields`. Specify the record to find by placing the appropriate values into `aKeyValues`. If only one key value is required, use a variant. If multiple key values are required, use a variant array.

`aOptions` defines a set of locate options for case sensitivity and partial key matches. Use `loCaseInsensitive` to carry out a case-insensitive search. Use `loPartialKey` to indicate that only a partial key value has been supplied in `aKeyValues`.

If a matching record is found, this function sets the cursor position and returns True. Otherwise, False is returned.

See also: Lookup

```
function Lookup(
    const KeyFields : string; const KeyValues : Variant;
    const ResultFields : string): Variant;
```

↳ Retrieves field values for a record that matches the search criteria.

Use this method to locate a record and retrieve specific field values from the matching record. `aKeyFields` is a list of field names delimited by semi-colons (e.g., “D;SSN;BirthDate”). If the dataset has an index that matches the fields specified in `aKeyFields`, the index is used to search for the record. Otherwise, a filter is used which can reduce performance.

`aKeyValues` must contain one value per field in `aKeyFields`. Specify the record to find by placing the appropriate values into `aKeyValues`. If only one key value is required, use a variant. If multiple key values are required, use a variant array.

`aResultFields` is a list of field names, delimited by semi-colons, whose values are to be returned. If a matching record is not found, this function returns a variant with the value of `False`. If a matching record is found, the value returned depends upon the number of fields requested. If only one field is requested, a variant containing the field's value is returned. If more than one field is requested, a variant array containing each field's value is returned.

See also: `Locate`

## ObjectView

## property

```
property ObjectView : Boolean
```

↳ Specifies whether fields are to be stored hierarchically or flattened out in the `Fields` property.

FlashFiler does not support array, ADT, or object fields. Do not use this property. This property is documented because it is inherited from `TDataSet` as a public property.

```
property OnCalcFields : TDataSetNotifyEvent
```

↳ Defines an event handler that is called when a dataset recalculates fields.

Use this event handler to calculate the value of calculated fields within the dataset. The OnCalcFields event is triggered automatically when the dataset's AutoCalcFields property is set to True.

**Note:** When the AutoCalcFields property is True, an OnCalcFields event handler should not modify the dataset because such modifications re-trigger the OnCalcField event, leading to recursion.

If an application permits users to change data, OnCalcFields is frequently triggered. To reduce the frequency with which OnCalcFields occurs, set AutoCalcFields to False. When AutoCalcFields is False, OnCalcFields is not called when changes are made to individual fields within a record.

**Note:** When the dataset is the master table of a master-detail relationship, OnCalcFields occurs before detail sets have been synchronized with the master table.

See also: AutoCalcFields

## OnDeleteError

event

```
property OnDeleteError : TDataSetErrorEvent
```

```
type TDataSetErrorEvent = procedure(
    DataSet : TDataSet; E : EDatabaseError;
    var Action : TDataAction) of object;
```

```
TDataAction = (daFail, daAbort, daRetry);
```

↳ Defines an event handler that is called when an exception is raised by the dataset's Delete method.

The OnDeleteError event is triggered when an error is encountered during a delete operation. The OnDeleteError event can be used to take specific action when a delete operation fails.

The DataSet parameter references the dataset that raised the exception triggering this event.

The E parameter points to the exception object containing the exception information.

Use the Action parameter to define what action, if any, is to be taken after the event is complete. The Action parameter may be used to retry the delete operation if required.

See also: Delete

```
property OnEditError : TDataSetErrorEvent

type TDataSetErrorEvent = procedure(
    DataSet : TDataSet; E : EDatabaseError;
    var Action : TDataAction) of object;

TDataAction = (daFail, daAbort, daRetry);
```

↳ Defines an event handler that is called when an exception is raised by the dataset's Edit or Insert method.

The OnEditError event is triggered when an error is encountered when trying to place a dataset in edit or insert mode. Use the OnEditError event to take specific action when an edit or insert operation fails.

The DataSet parameter references the dataset that raised the exception triggering this event.

The E parameter points to the exception object containing the exception information.

Use the Action parameter to define what action, if any, is to be taken after the event is complete. The Action parameter may be used to retry the operation if necessary.

See also: Edit, Insert

## OnFilterRecord

event

```
property OnFilterRecord : TFilterRecordEvent

type TFilterRecordEvent = procedure(
    DataSet : TDataSet; var Accept : Boolean) of object;
```

↳ Defines an event handler used to filter records.

The OnFilterRecord event is triggered when the cursor moves from one record to another. This method is used in conjunction with the dataset's Filtered property to specify which records in the dataset are available to the application. When Filtered is set to True, the OnFilterRecord event is called to determine if the record satisfies the filter criteria. If the record satisfies the criteria, set the Accept parameter to True.

The DataSet parameter references the dataset that triggered the event.

Use the Accept parameter to indicate whether or not the record is to be made available to the application.

**Note:** The OnFilterRecord event can be used in conjunction with the Filter property and ranges. Be sure to check these if you are not getting the results you expect.

See also: Filter, Filtered, FilterEval, FilterTimeout

```
property OnNewRecord : TDataSetNotifyEvent
```

↳ Defines an event handler that is called when a new record is added to the dataset.

The OnNewRecord event is triggered when a new record is about to be added to the dataset. Use this event to take specific action to initialize the fields in the new record.

---

**OnPostError****event**

```
property OnPostError : TDataSetErrorEvent
```

```
type TDataSetErrorEvent = procedure(  
    DataSet : TDataSet; E : EDatabaseError;  
    var Action: TDataAction) of object;
```

```
TDataAction = (daFail, daAbort, daRetry);
```

↳ Defines an event handler that is called when an exception is raised by the dataset's Post method.

The OnPostError event is triggered when an error is encountered during a Post operation. Use the OnPostError event to take specific action when a post operation fails.

The DataSet parameter references the dataset that raised the exception triggering this event handler.

The E parameter points to the exception object containing the exception information.

Use the Action parameter to define what action, if any, is to be taken after the event is complete. The Action parameter may be used to retry the post operation if necessary.

See also: Post

---

**OnServerFilterTimeout****event**

```
property OnServerFilterTimeout : TffServerFilterTimeoutEvent
```

```
TffServerFilterTimeoutEvent = procedure(Sender : TffTable;  
    var Cancel : Boolean) of object;
```

↳ Defines an event that is called when the server is evaluating a server-side filter and has not found a matching record within the number of milliseconds specified by the FilterTimeout property.

This event provides the opportunity to cancel the filter or to notify the user that the server requires extra time to continue the filter. The default value of the Cancel parameter is False. To have the server continue the filter, leave the value of Cancel as is. To have the server stop

the filter, set Cancel to True. When Cancel is set to True, the server-side filter is aborted and an `EffDatabaseError` exception is raised. The exception has the error code `DBIERR_FF_FilterTimeout`.

If a handler is not specified for this event, server-side filters will continue through to completion regardless of the amount of time required.

See also: `Filter`, `Filtered`, `FilterEval`, `FilterTimeout`

## PackTable

method

```
function PackTable(var aTaskID : LongInt) : TffResult;  
TffResult = longint;
```

↪ Frees up unused space in a table by removing deleted records.

Use this method to remove the unused space in a dataset's physical file. If there are fewer deleted records than would fill the size of one block then the packed table will not be smaller in size. In general you do not need to use this routine because FlashFiler Server automatically reuses the space occupied by deleted records.

This method initiates the pack as a separate task at the server and then returns immediately. If you need to determine the progress of the pack operation, use the returned `TaskID` and call the `GetTaskStatus` method of the `TffSession` component.

The following example packs the table and monitors the progress of the pack:

```
procedure TForm1.Button2Click(Sender: TObject);  
var  
    TaskID: LongInt;  
    TaskStatus: TffRebuildStatus;  
    Done: Boolean;  
begin  
    Check(Table1.PackTable(TaskID);  
    Check(Table1.Session.GetTaskStatus(TaskID, Done,  
        TaskStatus));  
    while not Done do begin  
        ProgressGauge.Progress := TaskStatus.rsPercentDone;  
        Check(Table1.Session.GetTaskStatus(TaskID, Done,  
            TaskStatus));  
    end;  
    ProgressGauge.Progress := 100;  
end;
```

See also: `TffSession.GetTaskStatus`, `TffBaseDatabase.Pack`

**Post****method**

```
procedure Post;
```

↳ Completes an edit or insert operation.

Call Post in order to complete the insertion or modification of a record. An implicit call to Post is made when calls are made to methods that change the dataset's state or cursor position.

**Note:** If a transaction is active then the server applies the changes to a cache. The data is not written to the physical files until the transaction is committed via the Commit method of TffDatabase.

See also: Cancel, TffBaseDatabase.Commit, Edit, Insert, TffBaseDatabase.Rollback

**RecNo****property**

```
property RecNo : Integer
```

Default: -1

↳ Identifies the current record.

FlashFiler does not support this property. This property always returns the value -1. This property is documented because it is inherited from TDataSet as a public property.

See also: IsSequenced

**Refresh****method**

```
procedure Refresh;
```

↳ Retrieves the latest data from the database in order to update the dataset's contents.

Use this method to make sure the dataset contains the most recent set of records. This is especially useful when updating a data-aware grid.

**RenameTable****method**

```
procedure RenameTable(const aNewTableName : string);
```

↳ Changes the name of the physical table to the given name.

Use this method to rename the physical files of the current table. aNewTableName is the name of the table to be renamed. This includes all supporting BLOB and index files associated with the table. A call to RenameTable can only be made when exclusive access to the table is available. If this method fails, an exception will be raised.



```
function RestructureTable(aDictionary : TffDataDictionary;
    aFieldMap : TStrings; var aTaskID : LongInt) : TffResult;

TffResult = longint;
```

↳ Changes the structure of the table.

Use this method to change any part of the table definition. For example, fields can be added, changed, or removed; indexes can be added, changed, removed, internalized, or externalized; and block sizes can be changed. Any attribute of the data dictionary can be altered while preserving the existing data.

This method is identical in functionality to the RestructureTable method of TffDatabase. Please see “ServerEngine read-only property” on page 188 for more information.

The following example changes the structure of an “Employee” table:

```
var
    Dict: TffDataDictionary;
    FldArray: TffFieldList;
    IdxHelpers : TffFieldIHList;
    FieldMap: TStrings;
    TaskID: LongInt;
    TaskStatus: TffRebuildStatus;
    Done: Boolean;
begin
    Dict := TffDataDictionary.Create(8192);
    try
        with Dict do begin
            AddField('EMP_NO', '', fftInt16, 0, 0, False, nil);
            AddField('LAST_NAME', '', fftShortString, 20, 0,
                False, nil);
            AddField('FIRST_NAME', '', fftShortString, 15, 0,
                False, nil);
            AddField('DEPT_NO', '', fftShortString, 3, 0,
                False, nil);
```

```

        {Add an external index on the LAST_NAME,
          FIRST_NAME fields}
        AddFile('Main Index File', 'MIF', 4096,
              ftIndexFile);
        FldArray[0] := 1; { Field #1 }
        FldArray[1] := 2; { Field #2 }
        {use the default index helpers}
        IdxHelpers[0] := '';
        IdxHelpers[1] := '';
        AddIndex('Main', 'by Last Name, First Name', 1, 2,
              FldArray, IdxHelpers, True, True, True);

        {Add an internal index on the DEPT_NO field}
        FldArray[0] := 3; {Field #3}
        AddIndex('Dept', 'by Dept_No', 0, 1, FldArray,
              IdxHelpers, True, True, True);
    end;
    FieldMap := TStringList.Create;
    try
        {<new fieldname>=<old fieldname>}
        with FieldMap do begin
            Add('EMP_NO=EMP_NO');
            Add('LAST_NAME=LAST_NAME');
            Add('FIRST_NAME=FIRST_NAME');
            Add('DEPT_NO=DEPT_NO');
        end;
        Check(Table1.RestructureTable(Dict, FieldMap, TaskID));
        {If you don't need to preserve the existing data,
         pass nil in place of the FieldMap parm and the
         loop to check the task status isn't needed}
    finally
        FieldMap.Free;
    end;
    Dict.Free;
end;
Check(Table1.Session.GetTaskStatus(TaskID, Done,
    TaskStatus));
while not Done do begin
    ProgressGauge.Progress := TaskStatus.rsPercentDone;
    Check(Table1.Session.GetTaskStatus(TaskID, Done,
        TaskStatus));
end;
ProgressGauge.Progress := 100;
end;

```

See also: `TffSession.GetTaskStatus`, `TffDatabase.Restructure`

## ServerEngine

property

```
property ServerEngine : TffBaseServerEngine
```

↪ References the ServerEngine through which this dataset routes its requests.

Client-side components route their requests through either an instance of TffRemoteServerEngine or TffServerEngine. This property references the server engine used by the dataset. This property is nil until the dataset is opened.

## Session

property

```
property Session : TffSession
```

↪ References the session component managing this dataset.

Use this property to obtain a handle on the owning session component. Until the SessionName property is set and the dataset is opened, this property is set to nil. The value of this property will be the same as the Session property of the dataset's Database property.

To change the session to which this table is attached, set the Active property to False and alter the SessionName property. Note that this will also disconnect the table from its database.

See also: Active, Database, DatabaseName, SessionName

## SessionName

property

```
property SessionName : string
```

Default: Empty string

↪ Identifies the SessionName of the session component managing this dataset.

Use this property to connect the dataset to a session component. The validity of this property is checked only when the dataset is opened. Each dataset must have a value for this property. Note that the value of this property must be the same as the owning database's SessionName property.

To obtain the session component managing the table, use the Session property. This property may be changed only when the Active property is set to False.

See also: Active, TffDatabase.DatabaseName, Session

SetTableAutoIncValue

method

```
function SetTableAutoIncValue(  
    const aValue: TffWord32) : TffResult;  
  
TffWord32 = type DWORD; TffResult = longint;
```

- ↳ Sets the auto-increment seed for the dataset.
- Use this method to set a dataset's auto-increment counter. aValue is the new value for the auto-increment seed. The next record inserted into the dataset receives this value plus one.

SparseArrays

property

```
property SparseArrays: Boolean
```

- ↳ Determines whether a unique TField object is created for each element of an array field.
- FlashFile does not support array fields. Do not use this property. This property is documented because it is inherited from TDataSet as a public property.

State

property

```
property State : TDataSetState  
  
TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert,  
    dsSetKey, dsCalcFields, dsFilter, dsNewValue, dsOldValue,  
    dsCurValue, dsBlockRead, dsInternalCalc, dsOpening);
```

Default: dsInactive

Use the State property to determine the current mode of the dataset. State identifies what can be done with a dataset, such as changing a record. The state of the dataset changes based upon the dataset methods invoked by the application.

State	Meaning
dsInactive	The dataset is closed. No data is available.
dsBrowse	The dataset is open. Data may be viewed but not changed. This is the default state of an active dataset.
dsEdit	The current record is available for modification.
dsInsert	The current record is a newly created buffer that has not been posted to the database. The fields within the record may be changed. The record may be posted or canceled.

State	Meaning
dsSetKey	Record searching is enabled, or a SetRange operation is in process. A restricted set of data may be viewed but records may not be inserted or edited.
dsCalcFields	An OnCalcFields event is in progress. Non-calculated fields may not be edited and a new record may not be inserted. It is possible to set the value of calculated fields.
dsFilter	An OnFilterRecord event is in progress. A filtered set of data may be viewed but records may not be inserted or modified.
dsNewValue	Temporary state used internally to indicate that a field component's NewValue property is being accessed.
dsOldValue	Temporary state used internally to indicate that a field component's OldValue property is being accessed.
dsBlockRead	Data-aware controls are not updated and events are not triggered when the cursor moves.
dsInternalCalc	Temporary state used internally to indicate that values need to be calculated for a field whose FieldKind property is set to fkInternalCalc.
dsOpening	The dataset is in the process of opening but has not finished.

**Timeout****property**

```
property Timeout : Longint
```

Default: -1

↳ Specifies the timeout for dataset operations.

This property specifies the number of milliseconds in which the FlashFiler Server must complete a dataset operation (e.g., post a record, move to the next record). If the server is unable to complete the operation in the specified amount of time, an instance of `EffException` having error code `fferrReplyTimeout` is raised.

If this property has the value -1 then the dataset uses the `Timeout` property of its owning database component. When this property is modified, the new timeout value is set to the FlashFiler Server and used in subsequent operations for the dataset.

To determine the actual timeout value used by the dataset, use the `GetTimeout` method.

See also: `GetTimeout`

**Translate****method**

```
function Translate(  
    Src, Dest : PChar; ToOem : Boolean): Integer;
```

↳ Copies a string from `Src` to `Dest`.

FlashFiler provides no additional functionality to `Translate` leaving it to `TDataSet` to perform the operation. `TDataSet` simply copies data from `Src` to `Dest` returning the size of the string copied.

See also: `TDataSet.Translate`

**Version****property**

```
property Version : string
```

↳ Returns the version number of FlashFiler Client.

This property specifies the version number of FlashFiler Client being used in the IDE. It is published for easy reference from the object inspector.

# TffBaseTable Class

The TffBaseTable class builds upon TffDataset to provide the necessary index properties and methods required by any indexed FlashFiler table. These index properties and methods match closely with the TTable from the VCL to provide seamless compatibility.

Do not create instances of this class. Instead use TffTable.

## Hierarchy

TDataset (VCL)	
❶ TffDataset (FFDB) .....	191
TffBaseTable (FFDB)	

## Properties

❶ Active	Exclusive	KeyFieldCount
❶ ActiveBuffer	❶ Filter	KeySize
❶ AggFields	❶ Filtered	❶ ObjectView
❶ AutoCalcFields	❶ FilterEval	ReadOnly
❶ BlockReadSize	❶ FilterOptions	❶ ServerEngine
❶ BOF	❶ FilterTimeout	❶ Session
❶ Bookmark	IndexDefs	❶ SessionName
❶ CanModify	IndexFieldCount	❶ SparseArrays
❶ CursorID	IndexFieldNames	TableName
❶ Database	IndexFields	❶ Timeout
❶ DatabaseName	IndexName	❶ Version
❶ Dictionary	KeyExclusive	

## Methods

❶ AddFileBlob	CancelRange	DeleteIndex
AddIndex	❶ CheckBrowseMode	❶ DeleteTable
AddIndexEx	❶ ClearFields	EditKey
❶ Append	❶ Close	EditRangeEnd
❶ AppendRecord	❶ ControlsDisabled	EditRangeStart
ApplyRange	❶ CompareBookmarks	❶ EmptyTable
❶ BookmarkValid	❶ CreateBlobStream	FindKey
❶ Cancel	CreateTable	FindNearest

- ❶ GetCurrentRecord
- ❶ GetFieldData
  - GetIndexNames
- ❶ GetRecordBatch
- ❶ GetRecordBatchEx
- ❶ GetTimeout
- ❶ GotoCurrent
  - GotoKey

- GotoNearest
- ❶ InsertRecordBatch
- ❶ IsSequenced
- ❶ Locate
- ❶ Lookup
- ❶ PackTable
- ❶ Post
  - ReindexTable

- ❶ RenameTable
- ❶ RestructureTable
  - SetKey
  - SetRange
  - SetRangeEnd
  - SetRangeStart
- ❶ SetTableAutoIncValue
- ❶ Translate

## Events

- ❶ AfterCancel
- ❶ AfterClose
- ❶ AfterDelete
- ❶ AfterEdit
- ❶ AfterInsert
- ❶ AfterOpen
- ❶ AfterPost
- ❶ AfterRefresh
- ❶ AfterScroll

- ❶ BeforeCancel
- ❶ BeforeClose
- ❶ BeforeDelete
- ❶ BeforeEdit
- ❶ BeforeInsert
- ❶ BeforeOpen
- ❶ BeforePost
- ❶ BeforeRefresh
- ❶ BeforeScroll

- ❶ OnCalcFields
- ❶ OnDeleteError
- ❶ OnEditError
- ❶ OnFilterRecord
- ❶ OnNewRecord
- ❶ OnPostError
- ❶ OnServerFilterTimeout



## Reference Section

### AddIndex

method

```
procedure AddIndex(  
    const aName, aFields : string; aOptions : TIndexOptions);  
  
TIndexOptions = set of TIndexOption;  
  
TIndexOption = (ixPrimary, ixUnique, ixDescending,  
    ixCaseInsensitive, ixExpression, ixNonMaintained);
```

🔗 Adds an index to the table.

Use this method to create a new index for the table. `aName` is the name of the index to be created. `aFields` is a list of fields, delimited by semi-colons, to include in the index. The order of the field names in this index is significant.

Use `aOptions` to specify attributes for the index.

Option	Meaning
<code>ixCaseInsensitive</code>	The index sorts records case insensitively.
<code>ixDescending</code>	The index is sorted in a descending manner (e.g., an index containing one string field would have 'Z' before 'A').
<code>ixUnique</code>	Each composite key value in the index is unique. There may be no duplicate composite key values.

If `aOptions` contains values other than those listed, an exception is raised.

Adding an index causes the table to be restructured. The server requires exclusive access to restructure a table. If this table is currently opened by another table component, session, or application then this method raises an exception.

**Note:** The `AddIndexEx` method provides more control over the creation of a new index.

See also: `AddIndexEx`

```

function AddIndexEx(
    const aIndexDesc : TffIndexDescriptor;
    var aTaskID : LongInt) : TffResult;

TffIndexDescriptor = packed record
    idNumber : longint;
    idName : TffDictItemName;
    idDesc : TffDictItemDesc;
    idFile : longint;
    idKeyLen : longint;
    idCount : longint;
    idFields : TffFieldList;
    idFieldIHLprs : TffFieldIHLList;
    idDups : boolean;
    idAscend : boolean;
    idNoCase : boolean;

TffDictItemName = string[ffcl_GeneralNameSize];
ffcl_GeneralNameSize = 31;
TffDictItemDesc = string[ffcl_DescriptionSize];
ffcl_DescriptionSize = 63;
TffFieldList = array [0..pred(ffcl_MaxIndexFlds)] of longint;
TffFieldIHLList = array [0..pred(ffcl_MaxIndexFlds)] of
    TffDictItemName;
ffcl_MaxIndexFlds = 16;
TffResult = LongInt

```

↪ Adds an index to the table.

AddIndexEx adds an index to the FlashFiler Dictionary in the table and then asks the server to create and populate the physical index. Since this process may take some time, the server starts up an asynchronous task and returns the task's unique identifier immediately. Use the GetTaskStatus method of the TffSession component to monitor the progress of the operation.

aIndexDesc defines the structure of the new index.

aTaskID is the unique ID of the task created by the server to add the new index. Use this parameter for calls to the GetTaskStatus method of the TffSession component.

The following example adds the index by name to a table:

```
var
    FFIndexDesc : TffIndexDescriptor;
    TaskID : LongInt;
    Done : Boolean;
    TaskStatus : TffRebuildStatus;
begin
    Screen.Cursor := crHourglass;
    try
        Done := False;

        FFIndexDesc.idNumber := 0;
        FFIndexDesc.idName := 'byName';
        FFIndexDesc.idFile := 0;
        FFIndexDesc.idFields[0] := 1;
        FFIndexDesc.idCount := 1;
        FFIndexDesc.idDups0 := True;
        FFIndexDesc.idNoCase := True;

        Check(Table1.AddIndexEx(FFIndexDesc, TaskID));
        Check(Table1.Session.GetTaskStatus(TaskID, Done,
            TaskStatus));
        while not Done do begin
            ProgressGauge.Progress := TaskStatus.rsPercentDone;
            Check(Table1.Session.GetTaskStatus(TaskID, Done,
                TaskStatus));
        end;
        ProgressGauge.Progress := 100;
    finally
        Screen.Cursor := crDefault;
    end;
end;
```

See also: `TffSession.GetTaskStatus`

## ApplyRange

method

```
procedure ApplyRange
```

✎ Applies the range defined by calls to `SetRangeStart` and `SetRangeEnd`.

Use this method to activate the range defined by previous calls to `SetRangeStart` and `SetRangeEnd`. After the range is applied, only the records that fall within the range are available to the dataset. Ranges are processed by the server and are very fast.

See also: `CancelRange`, `SetRange`, `SetRangeEnd`, `SetRangeStart`, `EditRangeEnd`, `EditRangeStart`

**CancelRange****method**

```
procedure CancelRange;
```

↪ Cancels an applied range.

Use this method to remove any range in effect for the table. Canceling the range enables access to all the available records in the dataset.

See also: [ApplyRange](#), [SetRange](#)

**CreateTable****method**

```
procedure CreateTable;
```

↪ Creates a new table on the FlashFiler Server.

Use this method to create a new table using the dataset's current index and field definitions. If the table already exists, this method overwrites the table. If the table exists and is already open, an `EffDatabaseError` exception having error code `DBIERR_TABLEOPEN` is raised. To avoid overwriting an existing table, use the `TableExists` method of `TffBaseDatabase`.

This method obtains field definitions from the `FieldDefs` property. It obtains index definitions from the `IndexDefs` property.

See also: `TffDatabase.CreateTable`, `TffBaseDatabase.TableExists`

**DeleteIndex****method**

```
procedure DeleteIndex(const aIndexName : string);
```

↪ Deletes an index from the table.

Use this method to remove an index from the table. `aIndexName` is the name of the index. A non-empty string must be specified for `aIndexName`. The re-index operation restructures the table and requires exclusive access to the table. If exclusive access cannot be obtained, an exception is raised.

```
procedure EditKey;
```

- ✚ Places the dataset into key edit mode.

When performing a number of searches involving multiple key values, use this method to speed up the changing of keys. If only one or two of the keys actually change between searches, use this method to preserve the contents of the search key buffer and put the keys into a modifiable state. Use the `IndexFields` property to determine the fields in the current index.

See also: `TffDataSet.State`

---

**EditRangeEnd****method**

```
procedure EditRangeEnd;
```

- ✚ Puts the dataset into key edit mode for the end of a range.

Use this method to change the end value of an existing range. Call the `ApplyRange` method to activate the modified range.

See also: `ApplyRange`, `EditRangeStart`

---

**EditRangeStart****method**

```
procedure EditRangeStart;
```

- ✚ Puts the dataset into key edit mode for the start of a range.

Use this method to change the beginning value of an existing range. Call the `ApplyRange` method to activate the modified range.

See also: `ApplyRange`, `EditRangeEnd`

## Exclusive

property

```
property Exclusive : Boolean
```

Default: False

↳ Defines whether the table component has exclusive access to the physical table.

Use this property to prevent other client applications, or other table components within the same client application, from accessing the table. Once the physical table is opened with Exclusive set to True, no other session on any machine can open the table.

Set this property to True in order to open the table exclusively. If the owning database component has its Exclusive property set to True, then this property is True by default.

See also: TffDatabase.Exclusive

## FindKey

method

```
function FindKey(const aKeyValues : array of const) : Boolean;
```

↳ Places the table's cursor on the record with the given key.

Use this method to move the table's cursor to the first record in the dataset matching the values specified in parameter aKeyValues. Parameter aKeyValues is an array of field values, delimited by commas. Each value may be a constant, variable, or nil, and must match the data type of the corresponding field in the current index.

When FindKey is successful, the cursor is positioned on the matching record and the result is set to True. Otherwise, the result is set to False and the cursor remains in its position prior to calling this method.

See also: FindNearest, IndexName

## FindNearest

method

```
procedure FindNearest(const aKeyValues : array of const);
```

↳ Places the table's cursor on or about the record with the given key.

Use this method to move the table's cursor to the first record in the dataset whose key fields are greater than or equal to the values specified by parameter aKeyValues. Parameter aKeyValues is an array of field values, delimited by commas. Each value may be a constant, variable, or nil, and must match the data type of the corresponding field in the current index.

See also: FindKey, IndexName

## GetIndexNames

method

```
procedure GetIndexNames(aList : TStrings);
```

↳ Retrieves the names of all indexes for the table.

Use this method to retrieve the list of index names for the table. `aList` is an instance of a class inheriting from `TStrings`. `aList` is cleared before the index names are retrieved. The index names are placed into `aList`.

See also: `IndexDefs`

## GotoKey

method

```
function GotoKey : Boolean;
```

↳ Positions the table cursor.

Use this method to search for a record whose key values have been previously specified by calls to `SetKey` or `EditKey`. If the record is found, `GotoKey` positions the cursor on the record and returns `True`. Otherwise, the cursor position is left as is and this function returns `False`.

See also: `GotoNearest`

## GotoNearest

method

```
procedure GotoNearest;
```

↳ Positions the table cursor.

Use this method to search for a record whose key values have been previously specified by calls to `SetKey` or `EditKey`. This routine sets the cursor on or about the record with the initialized key.

See also: `GotoKey`

## IndexDefs

property

```
property IndexDefs : TIndexDefs
```

↳ References the list of index definitions for this table.

The `IndexDefs` property references a list of available index definitions for the table. The `IndexDefs` property mirrors the functionality of `IndexDefs` for the standard VCL `TTable` component and can be used to retrieve information about the indexes available to the table.

See also: `GetIndexNames`

## IndexFieldCount

read-only property

```
property IndexFieldsCount : Integer
```

↳ Returns the number of items in IndexFields.

Examine IndexFieldCount to determine the number of fields available to the IndexFields property. IndexFieldCount is often used with the IndexFields property to iterate through the current set of index fields associated with the index.

See also: IndexFields

## IndexFieldNames

property

```
property IndexFieldNames : string
```

Default: Empty string

↳ Specifies the columns to use as an index for a table.

Use this property as an alternative to IndexName to specify the index for a table. The order of field names in the property must mirror the order of fields in the index.

The IndexFieldNames and IndexName properties cannot be used simultaneously. Setting one property clears the other.

See also: IndexName

## IndexFields

property

```
property IndexFields[aIndex : Integer]: TField
```

↳ References the array of TField objects comprising the current index key.

The IndexFields property is a zero-based array that contains a series of field objects that correspond to the fields in the current index.

**Note:** This property should not be used to set the index fields. Instead, use the IndexFieldNames property for this purpose.

See also: IndexFieldCount, IndexFieldNames



## IndexName

property

```
property IndexName : string
```

Default: Empty string

↪ Returns the name of the currently selected index.

Use this property to determine the name of the current index or to change the current index of the dataset. When no index is specified, the dataset defaults to the Sequential Access Index (i.e., index number 0), which is the physical order of the records as they exist in the table.

See also: IndexFieldNames

## KeyExclusive

property

```
property KeyExclusive : Boolean
```

Default: False

↪ Specifies how the upper and lower boundaries of a range should be interpreted.

Use this property to indicate whether or not records matching the upper and lower boundaries of the range are to be included in the dataset. The default value of False indicates that records matching the upper and lower boundaries are included in the dataset. To exclude the records, set this property to True.

## KeyFieldCount

property

```
property KeyFieldCount : Integer
```

Default: The number of fields in the key.

↪ Defines the number of fields to use when starting a partial key search on a multi-field key.

Use this property to specify how many fields are to be used in a partial key search. By default, this property is set to include all fields in a search.

See also: Locate, FindKey, FindNearest, GotoKey, GotoNearest.

## KeySize

property

```
property KeySize : Integer
```

↪ Returns the physical length of the key for the current index.

Use this property to determine the size, in bytes, of the key for the current index.

See also: IndexName

```
property ReadOnly : Boolean
```

Default: False

↳ Defines whether the table is to be opened for read-only access.

To prevent modifications to the underlying data, set this property to True. The table's ReadOnly property is automatically set to True if the owning database component has its ReadOnly property set to True.

See also: TffDatabase.ReadOnly

```
function ReindexTable(const aIndexNum : Integer;  
    var aTaskID : Longint) : TffResult;
```

```
TffResult = Longint;
```

↳ Reconstructs the key values for the specified index.

ReindexTable reconstructs the key values for the index specified by IndexNum in the table. aIndexNum is the number of the index to be rebuilt. The first index in the table, the Sequential Access Index, is index number zero. The index number is incremented for each subsequent index defined for the table.

This method is identical in functionality to the Reindex method of the TffBaseDatabase class. The following example rebuilds index number 1 in the table and monitors the progress of the re-index:

```
var  
    TaskID : LongInt;  
    TaskStatus : TffRebuildStatus;  
    Done : Boolean;  
begin  
    Check(Table1.ReindexTable(1, TaskID));  
    Check(Table1.Session.GetTaskStatus(TaskID, Done,  
        TaskStatus));  
    while not Done do begin  
        ProgressGauge.Progress := TaskStatus.rsPercentDone;  
        Check(Table1.Session.GetTaskStatus(TaskID, Done,  
            TaskStatus));  
    end;  
    ProgressGauge.Progress := 100;  
end;
```

See also: TffSession.GetTaskStatus, TffBaseDatabase.Reindex

```
procedure SetKey;
```

↳ Enables setting of keys prior to a search.

Use this method to initiate the setting of keys for a search or a range. This method places the dataset in `dsSetKey` state and empties the contents of the current search key buffer. Use the `FieldByName` method to access and set the values of specific key fields. Use the `IndexFields` property to determine the fields used in the current index.

See also: `EditKey`, `IndexFields`, `State`

## SetRange

method

```
procedure SetRange(
    const aStartValues, aEndValues : array of const);
```

↳ Applies a range to the table.

Use this method as an alternative to calling `SetRangeStart`, `SetRangeEnd`, and `ApplyRange`. This method limits the records seen by a dataset to those fitting within an upper and lower boundary.

`aStartValues` is an array containing the values for the lower boundary of the range.

`aEndvalues` is an array containing the values for the upper boundary of the range.

`SetRange` remove any current range and overwrite any range values previously specified by calls to `SetRangeStart` and/or `SetRangeEnd`.

**Note:** This method uses the current index. This message fails with an exception if you are using persistent fields and have not added the fields in the index.

The following example limits a dataset of orders to all records having a specific status:

```
orders.IndexName := 'byOrderStatus';
orders.SetRange(['N'], ['N']);
```

The following example limits a dataset of contacts to all records between a certain age in a certain province:

```
contacts.IndexName := 'byProvinceAge';
contacts.SetRange(['MO', 20], ['MO', 34]);
```

Note that if the preceding example specified the range as `[20, 'MO']` and `[34, 'MO']`, the dataset would include records from provinces other than MO. The range in the preceding example applied to this table would return records `[20, 'MO']` through `[21, 'IN']`.

See also: `ApplyRange`, `KeyExclusive`, `SetRangeEnd`, `SetRangeStart`

```
procedure SetRangeEnd;
```

↳ Puts the table into key edit mode for the end of a range.

Use this method to prepare for setting of a range's upper boundary. This method erases any previous end range values and sets them to null. Use `FieldByName` to define the end range values. After assigning start and end range values, call `ApplyRange` to activate the range.

The code in the following example restricts an Orders table to all orders within a particular sales region:

```
with ordersTable do begin
    IndexName := 'bySalesRegion';
    SetRangeStart;
    FieldByName('SalesRegion').asInteger := aRegion;
    SetRangeEnd;
    FieldByName('SalesRegion').asInteger := aRegion;
    ApplyRange;
end;
```

See also: `ApplyRange`, `SetRange`, `SetRangeStart`

```
procedure SetRangeStart;
```

↳ Puts the table into key edit mode for the start of a range.

Use this method to prepare for setting of a range's lower boundary. This method erases any previous start range values and sets them to null. Use `FieldByName` to define the start range values. After assigning start and end range values, call `ApplyRange` to activate the range.

The code in the following example restricts an inventory receipts table to all products from a specific vendor shipped within a certain date range:

```
with receiptsTable do begin
    IndexName := 'yVendorReceiptDate';
    SetRangeStart;
    FieldByName('VendorID').asInteger := aVendorID;
    FieldByName('ReceiptDate').asDateTime := aStartDate;
    SetRangeEnd;
    FieldByName('VendorID').asInteger := aVendorID;
    FieldByName('ReceiptDate').asDateTime := anEndDate;
    ApplyRange;
end;
```

See also: `ApplyRange`, `SetRange`, `SetRangeEnd`

```
property TableName : string
```

Default: Empty string

↩ Returns the name of the physical table.

Use this method to read or set the name of the physical table accessed by the table component. When the table component is activated (i.e., opened), the table specified by this property must exist in the database.

Values must be specified for the TableName and DatabaseName properties prior to the table being opened.

See also: Active, DatabaseName, Open



# TffTable Component

The TffTable component is the FlashFiler equivalent of the VCL TTable component. The table component provides control functions to manage a set of records within an application. The table component is dependent upon a TffSession component and a TffDatabase component that is either created explicitly or automatically.

The TffTable component provides access to a single table in a FlashFiler database. The table provides access to all the records and fields of the dataset, along with the methods to manipulate them.

The TffTable component is very similar to the TTable component. Therefore the help resources, examples, and third party books that describe the TTable component generally apply to the TffTable component.

If you are using TffTable components in a multithreaded application, each thread must have its own TffTable components. TffTable components may not be shared across threads. See “Multithreaded applications” on page 118 for more details.

## Hierarchy

TDataSet (VCL)

- ❶ TffDataSet (FFDB) ..... 191
  - ❷ TffBaseTable (FFDB) ..... 232
    - TffTable (FFDB)

## Properties

- |                  |                   |                |
|------------------|-------------------|----------------|
| ① Active         | ① Filter          | MasterFields   |
| ① ActiveBuffer   | ① Filtered        | MasterSource   |
| ① AggFields      | ① FilterEval      | ① ObjectView   |
| ① AutoCalcFields | ① FilterOptions   | ② ReadOnly     |
| ① BlockReadSize  | ① FilterTimeout   | ① ServerEngine |
| ① BOF            | ② IndexDefs       | ① Session      |
| ① Bookmark       | ② IndexFieldCount | ① SessionName  |
| ① CanModify      | ② IndexFieldNames | ① SparseArrays |
| ① CursorID       | ② IndexFields     | ② TableName    |
| ① Database       | ② IndexName       | ① Timeout      |
| ① DatabaseName   | ② KeyExclusive    | ① Version      |
| ① Dictionary     | ② KeyFieldCount   |                |
| ② Exclusive      | ② KeySize         |                |

## Methods

- |                    |                    |                        |
|--------------------|--------------------|------------------------|
| ① AddFileBlob      | ② DeleteIndex      | ② GotoNearest          |
| ② AddIndex         | ① DeleteTable      | ① InsertRecordBatch    |
| ② AddIndexEx       | ② EditKey          | ① IsSequenced          |
| ① Append           | ② EditRangeEnd     | ① Locate               |
| ① AppendRecord     | ② EditRangeStart   | ① Lookup               |
| ② ApplyRange       | ① EmptyTable       | ① PackTable            |
| ① BookmarkValid    | ② FindKey          | ① Post                 |
| ① Cancel           | ② FindNearest      | ② ReindexTable         |
| ② CancelRange      | ① GetCurrentRecord | ① RenameTable          |
| ① CheckBrowseMode  | ① GetFieldData     | ① RestructureTable     |
| ① ClearFields      | ② GetIndexNames    | ② SetKey               |
| ① Close            | ① GetRecordBatch   | ② SetRange             |
| ① ControlsDisabled | ① GetRecordBatchEx | ② SetRangeEnd          |
| ① CompareBookmarks | ① GetTimeout       | ② SetRangeStart        |
| ① CreateBlobStream | ① GotoCurrent      | ① SetTableAutoIncValue |
| ② CreateTable      | ② GotoKey          | ① Translate            |

# Events

- |                |                 |                         |
|----------------|-----------------|-------------------------|
| ❶ AfterCancel  | ❶ BeforeCancel  | ❶ OnCalcFields          |
| ❶ AfterClose   | ❶ BeforeClose   | ❶ OnDeleteError         |
| ❶ AfterDelete  | ❶ BeforeDelete  | ❶ OnEditError           |
| ❶ AfterEdit    | ❶ BeforeEdit    | ❶ OnFilterRecord        |
| ❶ AfterInsert  | ❶ BeforeInsert  | ❶ OnNewRecord           |
| ❶ AfterOpen    | ❶ BeforeOpen    | ❶ OnPostError           |
| ❶ AfterPost    | ❶ BeforePost    | ❶ OnServerFilterTimeout |
| ❶ AfterRefresh | ❶ BeforeRefresh |                         |
| ❶ AfterScroll  | ❶ BeforeScroll  |                         |



## Reference Section

### MasterFields

property

```
property MasterFields : string
```

Default: Empty string

↳ Defines the fields in a master table used to establish a master/detail relationship.

After setting the MasterSource property, use this property to define the fields in the master table that control the detail records shown in this table. Set the value of this property to one or more field names, with each field name separated by a semi-colon.

When the current record of the master table changes, the table retrieves the values of these fields from the master table. It uses the values to set a range on this table. The data types of the fields specified by this property must match the data types of the key fields in this table's current index.

The code in the following example restricts the orders in a dataset to those matching a certain customer ID and date:

```
OrdersTable.MasterSource := dsCustomers;  
OrdersTable.MasterFields := 'CustomerID;OrderDate';
```

See also: MasterSource

### MasterSource

property

```
property MasterSource : TDataSource
```

↳ Defines the master table used to establish a master/detail relationship.

Use this property to specify the master dataset driving the records in this table's dataset. After setting this property, set the key fields of the relationship via the MasterFields property.

When the current record in the master dataset changes, a new range is applied to this table using fields from the master dataset as specified in the MasterFields property.

The code in the following example restricts the orders in a dataset to those matching a specific customer ID:

```
OrdersTable.MasterSource := dsCustomers;  
OrdersTable.MasterFields := 'CustomerID';
```

See also: MasterFields



## TffQuery Component

Use the TffQuery component to retrieve data using a SELECT statement. The retrieved data is referred to as a result set. TffQuery is ideally suited for those situations where the criteria for retrieving the data are complex or you need to gather data from two or more tables. For details concerning the supported SELECT statement syntax, see page 185 in Chapter 13.

**Note:** TffQuery does not support other SQL statements such as INSERT, DELETE, and UPDATE.

If you are using TffQuery components in a multithreaded application, each thread must have its own TffQuery components. TffQuery components may not be shared across threads. See “Multithreaded applications” on page 118 for more details.

In order to retrieve a result set using TffQuery, you must do the following:

- Make sure the query component is inactive.
- If you expect to modify the result set, set RequestLive to True.
- Add the SELECT statement via the SQL property.
- Prepare the SELECT statement using the Prepare method. This is not required but it improves performance in cases where the SELECT statement is re-used.
- Open the query.

An opened query contains zero or more records where each record is comprised of the fields identified in the SELECT statement. You may scan through the result set using TDataSet's First, Last, Next, and Prior methods. If the result set is live then you may insert new records as well as edit or delete records in the result set.

## Preparing queries

Before the FlashFiler Server can execute a query, the server must prepare the query. When preparing the query, the server allocates various resources in the form of a SQL prepared statement object. The server parses the query and, if possible, optimizes the query. This process does not take much time but it does become wasteful if done too often.

You can control how often the FlashFiler Server prepares a query. To prepare a query, use its Prepare method. If you open a query without calling Prepare, FlashFiler Server automatically prepares the query for you. If you know that a particular query will be used more than once, it is better to explicitly call the Prepare method once. The FlashFiler Server will parse and optimize once. It will not need to do so again until you change the query's SELECT statement.

Once you have finished using a query, it is best to call `TffQuery.UnPrepare`. Doing so frees up the resources allocated by the FlashFiler Server.

There are two points at which your application should catch query-related exceptions. First, FlashFiler Server checks the query's grammar when the query is prepared. If a syntax problem is found, FlashFiler Client raises an `EffDatabaseError` exception. Second, if a type mismatch or some other problem occurs during query execution, FlashFiler Client raises an `EffDatabaseError` exception. In either case, the exception's `ErrorCode` contains the error number and its `Message` property describes the problem.

## Parameterized queries

If you have a query that is used frequently and the only thing that varies between uses is a piece of data in the `Where` clause, the query is a good candidate for parameterization. For example, an auto shop application allows a mechanic to view the past history of an existing customer. When mechanic Bob requests information on two different customers, the following queries are executed:

```
select C.name, H.Visit, H.RepairAmt, H.RepairDescription
from Customer C, History H
where H.CustID = C.CustID and C.CustID = 1002992

select C.name, H.Visit, H.RepairAmt, H.RepairDescription
from Customer C, History H
where H.CustID = C.CustID and C.CustID = 1299810
```

The FlashFiler Server must parse and optimize the query each time Bob requests a customer's history. This is wasteful. Notice that the only difference between the two queries is the customer ID. The query can be transformed into a parameterized query by replacing the explicit customer ID with a parameter, as shown in the following statement:

```
select C.name, H.Visit, H.RepairAmt, H.RepairDescription
from Customer C, History H
where H.CustID = C.CustID and C.CustID = :CustID
```

Parameters are prefixed with a semicolon. If your parameter name contains multiple words, wrap the words with double quotes and prefix the whole with a colon. Use the following query as an example:

```
select Name, FavoriteBeverage
from Patrons where Bar = : "Bar Location ID"
```

The parser ignores double colons and colons within single or double quotes.

When the SQL statement is prepared, the FlashFiler Server parses and optimizes the query only once. After the statement is prepared, a value is specified for the parameter prior to each execution of the query. The FlashFiler Server does not need to reparse the query because only the parameter has changed.

At design time the IDE automatically generates TParam instances when you enter a SELECT statement into the SQL property. To access these instances, click the Params property editor button in the Object Inspector. The IDE displays a dialog listing the query's parameters. Select a parameter in order to display the parameter's properties in the Object Inspector.

During run time there are three ways to specify a parameter for a query:

- Set the parameter by name.
- Set the parameter by its position within the Params property.
- Set the parameter using a DataSource.

### Setting parameters by name

At run time you may search for a parameter using the ParamByName property. Once you have found the parameter, you may get or set its value. The following code example shows you how to find and set the parameter CustID:

```
for Index := low(CustArray) to high(CustArray) do begin
    aQuery.Close;
    aQuery.ParamByName('CustID').asString := CustArray[Index];
    aQuery.Open;
    {Do some work...}
end;
```

## Setting parameters by position

At run time, the `ParamCount` property indicates the number of parameters found in the `SELECT` statement. To access a specific property, retrieve the parameters list from the `Params` property and index into the list. In the following example, we assign the fields of a record to a query's parameters:

```
type
    TQueryRecord = packed record
        Model : string;
        Year : Integer;
        Manufacturer : LongInt;
    end;

function GetAuto(aQuery : TffQuery;
                 aRecord : TQueryRecord) : Boolean;
begin
    {Assumption: Query has been prepared.}
    with aQuery, aRecord do begin
        Close;
        Params[ciModel].AsString := Model;
        Params[ciYear].AsInteger := Year;
        Params[ciManufacturer].AsInteger := Manufacturer;
        Open;
        Result := (RecordCount > 0);
    end;
end;
```

## Setting parameters by DataSource

You can supply parameter values to a query by attaching a `TDataSource` descendant to the query's `DataSource` property. The `TDataSource` could be attached to a table or another query. When you open the query, `TffQuery` tries to match parameters to the fields within the `DataSource`. If a match is made, the value of the field is copied to the parameter. Parameters for which a match is not found must be programmatically set.

If the `DataSource` closes and re-opens then the query closes and re-opens. If the `DataSource` moves to a different record, the query refreshes its parameters from the new record in `DataSource` and re-executes. This behavior is ideal for implementing a master-detail relationship between two queries.

The following code shows you how to set up a query that obtains its parameters from another query:

```
{Gather all orders entered today.}
with orderQuery do begin
    SQL.Add('select * from orders where Created =: aDate');
    Prepare;
    {Assumption: All orders have Creation date with
      a 'null' time.}
    ParamByName('aDate').AsDateTime := Date;
    Open;
end;

{Connect the dataSet to the orders query.}
ordersDS.DataSet := orderQuery;

{Construct query for order lines.}
with orderLinesQuery do begin
    SQL.Add('select * from ordLines where orderID = :ID');
    DataSource := ordersDS;
    Prepare;
    Open;
end;

{Process the lines for each order.}
while not orderQuery.EOF do begin
    ProcessOrderLines(orderLinesQuery);
    orderQuery.Next;
end;
```

The example uses a TffQuery to retrieve all orders entered today. Note that this SELECT statement works only if the creation date of the orders is stored in a uniform manner (i.e., no time component). Next, the example connects a data set to the order query and constructs a query that grabs the lines associated with a specific order.

The order lines query connects to the orders query through the data set. The ID parameter in the order lines query is filled from the ID field in the order query. After the order lines query is opened, its result set contains all order lines associated with the first order. Finally, the example loops through the orders, processing the line items associated with each order.

## Live result sets

In this version of FlashFiler, all result sets are read-only. In order to insert, modify, or delete records, use a TffTable component to carry out the operation.

# Hierarchy

TDataSet (VCL)

    ❶ TffDataSet (FFDB) ..... 191

        TffQuery (FFDB)

## Properties

❶ Active	DataSource	Prepared
❶ ActiveBuffer	❶ Dictionary	RequestLive
❶ AggFields	❶ Filter	❶ SparseArrays
❶ AutoCalcFields	❶ Filtered	❶ ServerEngine
❶ BlockReadSize	❶ FilterEval	❶ Session
❶ BOF	❶ FilterOptions	❶ SessionName
❶ Bookmark	❶ FilterTimeout	SQL
CanModify	❶ ObjectView	StmtHandle
❶ CursorID	ParamCheck	Text
❶ Database	ParamCount	❶ Timeout
❶ DatabaseName	Params	❶ Version

## Methods

❶ AddFileBlob	❶ DeleteTable	❶ Lookup
❶ Append	❶ EmptyTable	❶ PackTable
❶ AppendRecord	❶ GetCurrentRecord	ParamByName
❶ BookmarkValid	❶ GetFieldData	❶ Post
❶ Cancel	❶ GetRecordBatch	Prepare
❶ CheckBrowseMode	❶ GetRecordBatchEx	❶ RenameTable
❶ ClearFields	❶ GetTimeout	❶ RestructureTable
❶ Close	❶ GotoCurrent	❶ SessionName
❶ ControlsDisabled	❶ InsertRecordBatch	❶ SetTableAutoIncValue
❶ CompareBookmarks	❶ IsSequenced	❶ Translate
❶ CreateBlobStream	❶ Locate	UnPrepare

## Reference Section

### CanModify

property

property CanModify : Boolean

↪ Indicates whether the result set may be modified.

Use this property to determine if the query's result set is modifiable. In this version of FlashFiler, all result sets are read-only. Therefore, the CanModify property always returns False.

See also: RequestLive

### DataSource

property

property DataSource : TDataSource

↪ Specifies the data source component used to fill parameters in the query's SQL statement.

There are two ways to fill the parameters associated with a parameterized query. One way is to programmatically set parameter values using the ParamByName method or the Params property. The other way is to use a DataSource to automatically fill any or all of the parameters.

When you attach a DataSource to the query, parameters having the same name as a field in the DataSource are filled with the value from the current record of DataSource. Parameters not having a matching field are empty and must be filled programmatically. If you programmatically set a parameter and it has a corresponding field in the DataSource, the programmatically set value is overwritten by the value from the DataSource.

For example, the following query contains the parameter SSN. If the DataSource contains an SSN field then the query is filled with the value of SSN for DataSource's current record:

```
select FirstName, LastName, State
  from Person
 where SSN = :SSN
```

If you navigate through the records of DataSource, the query's parameters are automatically updated and the query is executed. This provides a way for you to implement master-detail relationships using query components. For example, the following query retrieves all stock items stored at a warehouse:

```
select W.ID, P.ItemName, P.OEMNum,
       W.QtyOnHand, W.QtyOnOrder
  from Warehouse W, Parts P
 where P.OEMNum = W.OEMNum
```



DataSource1 is created and its DataSet property is set to this query component. A second parameterized query retrieves the location details for a specific stock item based upon the current row in the first query's result set:

```
select L.Section, L.Location, L.LocQty
from Location L
where L.WarehouseID = :W.ID
```

**Note:** DataSource's DataSet property may not reference this query component. That is regarded as a circular reference and is not allowed.

See also: Params

## ParamByName method

```
function ParamByName(const aName : string) : TParam;
```

↳ Retrieves the parameter having the specified name.

Use this method to find a parameter based upon its name. Once the parameter is found you may set or use its value. The following code indicates how a parameter may be used to set the value of an edit field:

```
efSSN.Text := FFQuery.ParamByName('SSN').AsString;
```

The following code indicates how a query parameter is set using this method:

```
FFQuery.ParamByName('DecisionMaker').asBoolean := True;
```

See also: DataSource, Params, ParamCount

## ParamCheck property

```
property ParamCheck : Boolean
```

Default: True

↳ Indicates whether the query's parameter list is to be updated when the SQL property changes.

When this property is set to True, each change to the SQL statement causes the query to clear its parameters and regenerate the parameters based upon the new SQL statement. This guarantees the parameter list matches the SQL statement.

Set this property to False if your query does not contain parameters.

See also: Params, SQL

```
property ParamCount : Word
```

↳ Returns the number of parameters generated for the current SQL statement.

Use this property to determine the number of parameters in the Params property. If ParamCheck is True, this property always matches the number of parameters in the SQL statement. Otherwise, the value returned by this property may be inaccurate.

See also: ParamCheck, Params, SQL

```
property Params : TParams
```

↳ Provides access to the parameters for the current SQL statement.

Use this property to read and/or programmatically modify the query's parameters. Use the ParamCount property to determine the number of parameters found in the current SQL statement. If ParamCheck is True, this property is accurate for the current SQL statement.

At design time, you may edit the individual parameters using the property editor attached to Params in the Object Inspector.

The following source code shows you how to programmatically set parameter values at run time.

```
aQuery.Params.Items[0].AsString := 'MO';  
aQuery.Params.Items[1].AsInteger := 35000;
```

See also: DataSource, ParamCheck, ParamCount, SQL

```
procedure Prepare;
```

↳ Prepares a SQL statement for execution.

Call this method to have the query send a SQL statement to the FlashFiler Server for parsing and optimization. Preparing a query before optimization improves performance because the FlashFiler Server does not have to re-parse and optimize the query. The following source code shows how Prepare may be used:

```
with FFQuery do begin  
  SQL.Add('Select * from Contacts');  
  Prepare;  
  Open;  
end;
```

If you open a query without first preparing the query, the query component automatically prepares the query. Preparing a query allocates server-side resources on the FlashFiler Server to the query. When you have finished using a query, it is best to call `UnPrepare` in order to free up the server-side resources.

If you call `Prepare` without first specifying a SQL statement then an `EffDatabaseError` exception with error code `ffdse_EmptySQLStatement` is raised.

If you call `Prepare` while the query is open, an `EffDatabaseError` exception with error code `ffdse_QueryMustBeClosed` is raised.

If you call `Prepare` and a failure occurs during preparation, an `EffDatabaseError` exception with error code `ffdse_QueryPrepareFail` is raised.

See also: `Prepared`, `UnPrepare`

### **Prepared** **property**

---

property `Prepared` : Boolean

↳ Indicates whether a query has been prepared for execution.

If `Prepared` returns `True`, the query is prepared. If `Prepared` returns `False`, the query is not prepared. For optimal performance, queries should be prepared prior to opening the query component.

You may also use this property to `Prepare` or `UnPrepare` the query. If you set this property to `True`, the query is prepared. If you set `Prepared` to `False`, the query is `UnPrepared`.

See also: `Prepare`, `UnPrepare`

### **RequestLive** **property**

---

property `RequestLive` : Boolean

Default: `False`

↳ Indicates whether a live result set is expected.

This property is normally used to request a live result set. In this version of FlashFiler, all result sets are read-only. Therefore, setting this property to `True` or to `False` always results in a read-only result set.

See also: `CanModify`

property SQL : TStrings

↳ Contains the SQL statement to be executed by the query component.

Use the SQL property to specify the SELECT statement that is to be executed by the query component. At design time you may edit the property using a String List editor invoked from the Object Inspector.

You may specify one SELECT statement. This property does not support multiple SELECT statements or scripts.

The SELECT statement may contain replaceable parameters where each parameter is prefixed by a colon (':'). If your parameter name contains multiple words, wrap the words with double quotes and prefix the whole with a colon as shown in the following query:

```
select Name, City, State
from Zoos where MetroSize = : "Population Count"
```

The parser ignores double colons and colons within single or double quotes. The following source code illustrates how to add a SQL statement to a query:

```
with aQuery do begin
    SQL.Add('Select TickerSymbol, High, Low, ');
    SQL.Add('ClosePrice, Volume');
    SQL.Add('from StockClosingInfo');
end;
```

See also: Text

---

# TffBlobStream Class

The TffBlobStream class is the FlashFiler equivalent of the VCL TBlobStream class. The BlobStream object provides a means to access Blob data and manage how that data is retrieved from the server.

For an example of how to use TffBlobStream, see “Order Entry” on page 598.

## Hierarchy

TStream (VCL)  
    TffBlobStream (FFDB)

## Properties

CurrPosition	ChunkSize
CurrSize	CancelTransfer

## Methods

Create	Read	Truncate
Destroy	Seek	Write

## Reference Section

### **CurrPosition**

**read-only property**

```
property CurrPosition : Longint
```

↪ Specifies the current offset into the data stream.

Use this property to retrieve the current offset into the associated BLOB field's data buffer.

### **CurrSize**

**read-only property**

```
property CurrSize : Longint
```

↪ Specifies the size of the BLOB's data.

CurrSize returns the total size of the data associated with the BLOB field.

### **ChunkSize**

**property**

```
property ChunkSize : Longint
```

Default: FFMaxBlobChunk

↪ Specifies the maximum size of a data packet when reading the BLOB from or writing the BLOB to the FlashFiler Server.

ChunkSize can be used to fine tune how BLOB data is retrieved and sent to the server. By default, the ChunkSize is set to the constant FFMaxBlobChunk. When this property is set to 0, BLOBs are written or retrieved in a single operation. This may cause a bottleneck on the server when dealing with large BLOBs.

See also: FFMaxBlobChunk

## CancelTransfer

property

```
property CancelTransfer : Longint
```

Default: False

↳ Aborts a read or write operation.

Read and write operations have the potential to be done in chunks when the `ChunkSize` is not 0. After every chunk of data is retrieved, the `CancelTransfer` property is queried to determine if the operation should continue. Set `CancelTransfer` to `True` in order to abort any read or write operation currently in progress.

**Note:** The `Read` and `Write` methods reset `CancelTransfer` to `False` every time they start.

See also: `ChunkSize`

## Create

method

```
constructor Create(  
    aField : TBlobField; aMode : TBlobStreamMode)  
  
TBlobStreamMode = (bmRead, bmWrite, bmReadWrite);
```

↳ Creates and initializes a new `TffBlobField` instance.

Use this method to create an instance of `TffBlobStream`. Note that it is preferable to use the `TffDataSet.CreateBlobStream` instead of using this constructor.

`aField` specifies the BLOB field with which the instance of `TffBlobStream` is to be associated. `aMode` determines the access given to the stream. If `aMode` is set to `bmRead` then the stream may read from but not write to the BLOB. If `aMode` is `bmWrite` then the stream may write to but not read from the BLOB. If `aMode` is `bmReadWrite` then the stream may read from or write to the BLOB.

See also: `TffDataSet.CreateBlobStream`, `Destroy`

## Destroy

method

```
destructor Destroy; override;
```

↳ Frees an instance of `TffBlobStream`.

`Destroy` release the instance of `TffBlobStream` and closes the associated BLOB on the server. The destructor notifies any linked controls if the BLOB data has been modified.

See also: `Create`

## Position

property

```
property Position : Longint
```

Default: 0

↪ Indicates the current position of the stream's cursor within the BLOB.

Use this property to read or set the position of the stream's cursor within the BLOB. Setting Position to zero moves the cursor to the beginning of the BLOB. After writing to the BLOB via the Write method, the value of Position is the last byte written. In order to read from the beginning of the BLOB, first set this property to zero.

See also: Read, Write

## Read

method

```
function Read(var aBuffer; aCount : Longint) : Longint;
```

↪ Reads data from the BLOB.

Use this method to retrieve data from the BLOB.

aBuffer is the destination for the BLOB data and must have at least Count bytes allocated to hold the data read from the BLOB. Read transfers up to Count bytes from the BLOB into Buffer, starting in the current position, and then advances the current position by the number of bytes actually transferred. This method returns the number of bytes actually transferred which may be less than the number requested in Count.

All the other data-reading methods of a BLOB stream (ReadBuffer, ReadComponent) call Read to do their actual reading.

**Note:** Do not call Read when the TffBlobStream was created in bmWrite mode.

See also: Write



```
function Seek(aOffset : Longint; aOrigin : Word) : Longint;
```

↳ Positions the stream to a specific offset in the BLOB.

Use Seek to move the current position within the BLOB by the indicated offset. Seek allows an application to read from or write to a particular location within the BLOB data.

aOffset is a starting position within the BLOB. aOrigin indicates how to interpret the aOffset parameter. aOrigin must have one of the following values:

Value	Meaning
soFromBeginning	aOffset is the number of bytes from the beginning of the BLOB data. Seek moves to the position specified by aOffset. aOffset must be greater than or equal to zero.
soFromCurrent	aOffset is the number of bytes to move from the current position. Seek moves to Position plus aOffset.
soFromEnd	aOffset is the number of bytes from the end of the BLOB. aOffset must be less than or equal to zero. Seek moves to EOF minus aOffset.

Seek returns the new value of the Position property.

See also: Position

```
property Size : Longint
```

↳ Indicates the size of the BLOB in bytes.

Use this method to determine the size of the BLOB in bytes.

```
procedure Truncate;
```

↳ Removes data from the BLOB.

Use Truncate to delete all BLOB data from the current position to the end of the BLOB. Calling Truncate when the current position is 0 deletes the entire BLOB.

**Note:** Do not call Truncate when the TiffBlobStream was created in bmRead mode.

See also: Create, Position

---

```
function Write(const aBuffer; aCount : Longint) : Longint;
```

↪ Writes data to the BLOB.

Use Write to append a buffer of data to a BLOB field, starting at the current position. aBuffer is a buffer containing the data to write. aBuffer must be at least aCount bytes in size. aCount is the number of bytes to write to the BLOB. This function returns the number of bytes successfully written to the BLOB.

**Note:** Do not call Write when the TffBlobStream was created in bmRead mode.

See also: Create, Position



---

## Chapter 8: Advanced Architecture

A major goal of FlashFiler is to open its architecture so that expert users may take full advantage of its capabilities. FlashFiler, as delivered, provides a strong client/server database engine. Its capabilities are supported by a number of components. This chapter shows you how to leverage those components in order to build even more capabilities into your products.

The first section of this chapter explains how information is transferred from FlashFiler Client to FlashFiler Server and back. The next section shows you how to add new transports, engines, and other components to the existing FlashFiler Server. If you are interested in embedding a server engine within your own application, you learn how to do so in the third section.

The fourth section uses a Chat example to show you how to use bidirectional transports, create a command handler, and create new messages passed between client and server. The remaining sections provide examples for adding new commands to FlashFiler Server and extending the functionality of FlashFiler's server engine. Specifics on the classes mentioned in this chapter can be found in the following places:

- Chapter 9, Transports and Threads
- Chapter 10, Command Handlers
- Chapter 11, Server Engines
- Chapter 12, Plugins and Extenders

## Following the Flow

The beginning of understanding FlashFiler's architecture is learning how requests and replies flow through FlashFiler. When a FlashFiler Client wants the FlashFiler Server to perform an action, it issues a request to the server. After the server completes the request, it sends a reply back to the client. The reply tells the client whether the request was completed successfully and may contain other information, depending upon the type of request.

This section contains several flowcharts describing the path of requests and replies within FlashFiler. The simplest scenario to illustrate is the routing of requests to an embedded server. In this scenario, the FlashFiler Client contains a table, database, session, client and embedded server engine as shown in Figure 8.1.

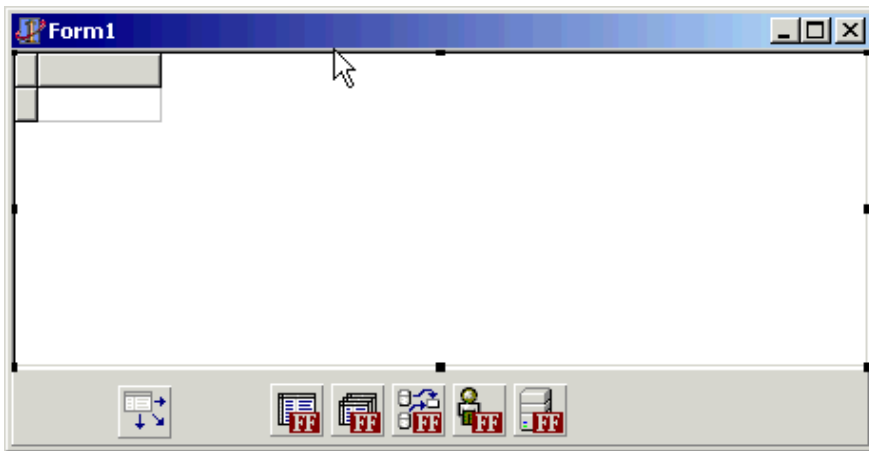


Figure 8.1: The main form of an application using an embedded server engine.

The requests are implemented as method calls within table, database, session, and client. There is no communications layer involved. The client components directly call methods of TffServerEngine. The result and output parameters of TffServerEngine's methods provide the reply. Figure 8.2 illustrates this scenario.

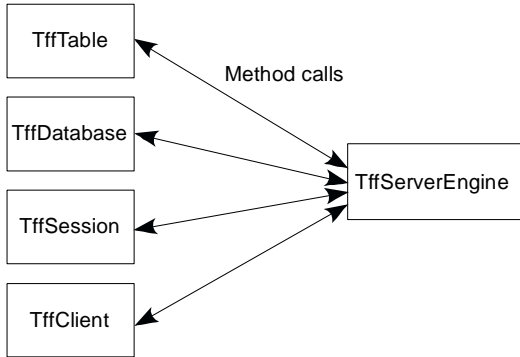


Figure 8.2: The call path for an embedded server engine.

If you replace the embedded server engine with a remote server engine (i.e., the client now talks to a server located on another workstation), this turns into a four-stage process. Figure 8.3 illustrates the components being used.

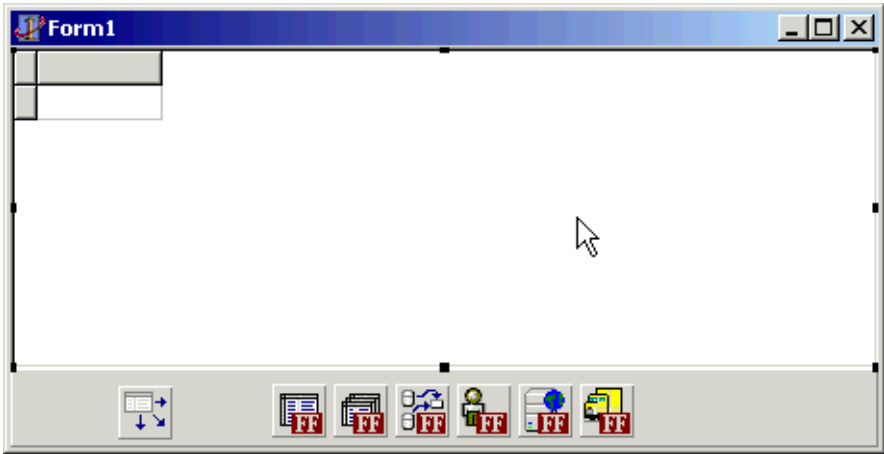


Figure 8.3: The main form of a client using a remote server engine.

In the first stage of the process, the client components call methods of `TffRemoteServerEngine`. The remote server engine component translates these into instances of `TffRequest`. The `TffRequest` instances (i.e., requests) are passed on to the transport's `Post` and `Request` methods. If the client expects a reply to the request, the client's thread waits until a reply is received from the server. Figure 8.4 depicts the first stage.

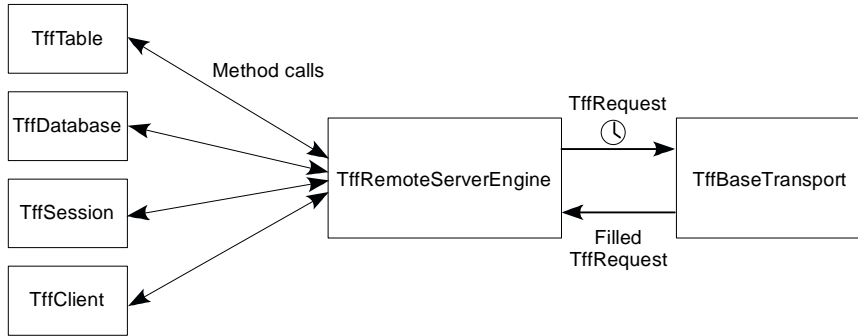


Figure 8.4: Remote server engine: stage 1

In the second stage, the transport places the request into a FIFO queue. By this time, the transport has been initialized and has established a connection with a like transport on the remote server. When the transport initializes, it creates a separate thread for the processing of requests and replies. The transport thread pulls the request from the FIFO queue and sends it to the corresponding transport on the remote server. Figure 8.5 illustrates the second stage.

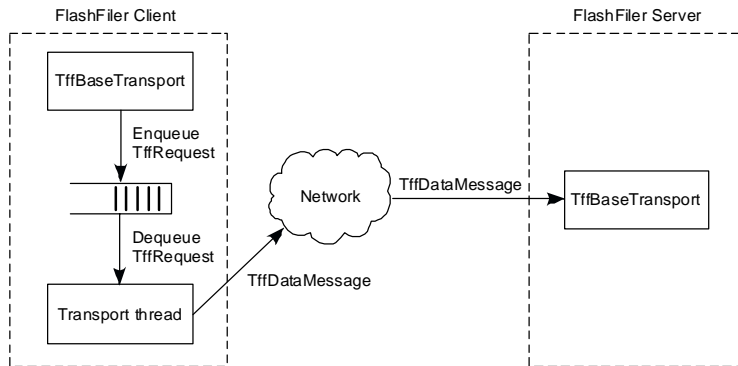


Figure 8.5: Remote server engine: stage 2

In the third stage, the transport on the remote server receives the request. As described in the Section Creating Your Own Server, the transport is connected to a thread pool. The transport uses the `ProcessThreaded` method of the `TffThreadPool` class to obtain a thread and have the thread process the request.

The thread submits the request to a command handler's `Process` method, with the command handler being obtained from the `TffBaseTransport.CommandHandler` property. The command handler passes the request to an embedded server engine for processing. If the embedded server engine does not recognize the request, the command handler passes the request to the connected plugin command handlers. If the request is not handled by a plugin command handler or plugin engine, the transport finally routes it to the engine manager. The third stage is shown in Figure 8.6.

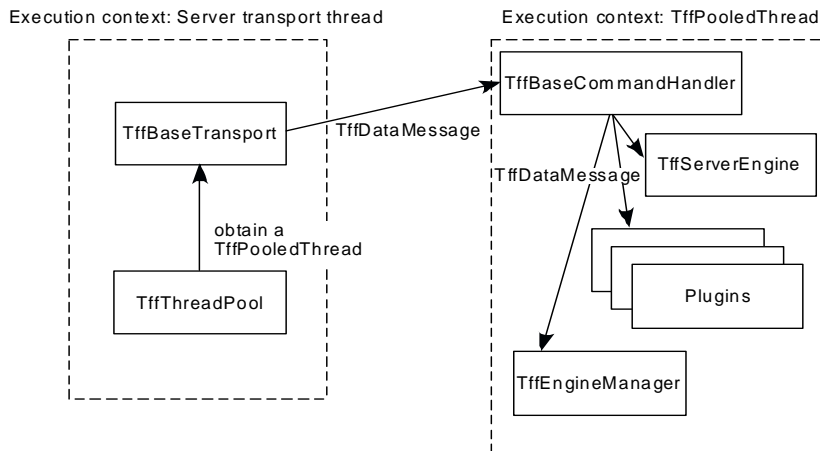


Figure 8.6: Remote server engine: stage 3

In the final stage, the engine or manager recognizing the request carries out the appropriate action. The command handler feeding the request to the engine is typically responsible for translating the engine's result and output parameters into a suitable reply. The reply then flows back through the remote server's transport to the client transport. The client transport



wakes up the waiting thread and the TffRemoteServerEngine translates the reply back into a result and output parameters recognizable by the client. This stage is described by Figure 8.7.

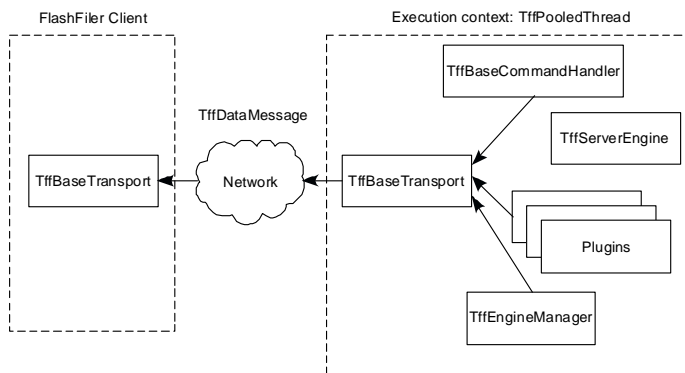


Figure 8.7: Remote server engine: stage 4

In the event that a client component does not expect a reply from the server, the client thread does not wait for a reply and the fourth stage is skipped.

# Modifying the FlashFiler Server

The FlashFiler Server (declared in project file FFSERV.DPR) consists of a user interface and an engine manager. The engine manager is a special type of data module that contains the working guts of the server. As shown in Figure 8.8, the engine manager contains a server engine, SQL engine, thread pool, security monitor, command handler, and several transports.

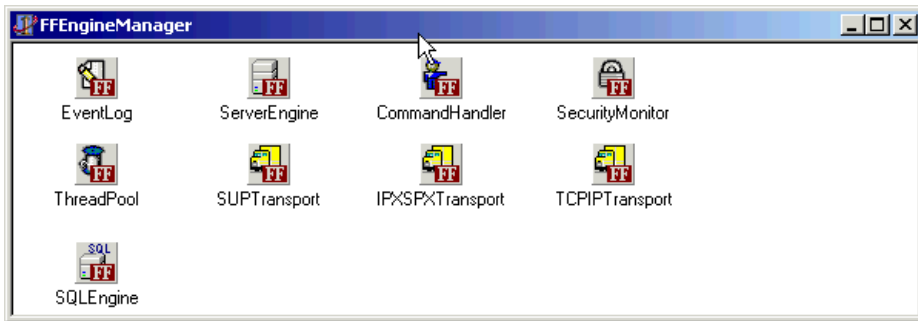


Figure 8.8: The Engine Manager data module.

When you activate the FlashFiler Server via its user interface, the FlashFiler Server tells the engine manager to start. The engine manager in turn tells each contained server engine to start. The server engines pass the start request on to their command handlers and the command handlers on to their transports. A similar sequence of actions takes place when you tell the FlashFiler Server to shut down.

As delivered, the engine manager contains three instances of TffLegacyTransport. One instance supports TCP/IP, another supports IPX/SPX, and another supports Single User Protocol (SUP). If you decide that your FlashFiler Server does not need to support SUP or IPX/SPX, delete the appropriate components from the engine manager. If a third-party vendor creates a new type of FlashFiler transport, it is just as easy to drop it on the engine manager's form and connect it to a command handler.

The engine manager also contains an instance of TffSecurityMonitor. TffSecurityMonitor verifies that a client has the appropriate rights for a given action within the server. For example, a client wishing to insert a record into a table must have Insert rights. If your application does not require security checking or if you want to use a third-party security alternative, feel free to delete or replace the standard security monitor.

The engine manager uses its Components property to find server engines. If you want to add another server engine to the engine manager, drop one onto the engine manager's form. You must also add at least one command handler and one transport to the engine manager, connecting the transport to the command handler and the command handler to the server

engine. This provides a path for requests to be received by the transport and routed to the new server engine. As shown in Figure 8.9, the server engine and its transports appear in the server UI.

The screenshot shows the TurboPower FlashFiler [Test] application window. It has a menu bar with 'Server', 'Config', 'Debug', and 'Help'. Below the menu bar is a toolbar with a play button and a stop button. The main window is divided into two sections: 'Servers' and 'Transports for Test'.

**Servers Table:**

Name	State	Clients	Sessions	Open Databases	Open Tables	Cursors	RAM (MB)
Production	Started	0	0	0	0	0	0
Test	Started	0	0	0	0	0	0

**Transports for Test Table:**

Name	Address	State	Clients	Messages	Messages/Sec
IPX/SPX (FF)	1234CDEF:00-00-00-00-00-02	Started	0	0	0.0000
Single User (FF)	Local	Started	0	0	0.0000
TCP/IP (FF)	111.111.111.111	Started	0	0	0.0000

The status bar at the bottom right shows 'Up: 0:00:00:20'.

Figure 8.9: The FlashFiler Server showing multiple server engines.

There are some limitations on what you may do with the existing FlashFiler Server.

1. You may replace the existing engine manager but the new engine manager must have the same name (i.e., TffEngineManager) as the existing engine manager. This is because the variable FFEngineManager is hard-coded in the server UI (see form UFFSMMAIN).
2. You may not add a second engine manager to the FlashFiler Server. This is because the server UI is hard-coded to recognize only one engine manager.
3. Third-party transports may not be configured using the server UI. This is because the server's Network Configuration screen (declared in form UFFSNET) is hard-coded to recognize the three pre-defined legacy transports.
4. When a server engine starts, it looks for server tables within the directory specified by its ConfigDir property. If you add server engines to the engine manager, you must specify a different ConfigDir for each server engine. Otherwise, the server engines will use the same set of server tables.

---

# Creating Your Own Server

In some situations, you may find it necessary to create an application that contains a FlashFiler server and have another set of client applications connect to that server. For example, an analysis program processes data supplied by a number of lightweight monitoring applications. The sole job of the monitoring applications is to obtain and submit data. For fastest performance, the analysis program has direct access to the server. It does not pay the cost of communicating with the server across a network.

To build such a system, you may use the engine manager supplied with FlashFiler (declared in form UFFEGMGR) or create a new engine manager. To create a new engine manager, do the following:

1. Start C++Builder or Delphi.
2. Start a new project or load the project that is to host the FlashFiler server.
3. Choose the File| New... menu item. The New Items dialog box appears.
4. Click on the Data Modules tab. The Data Modules tab contains a FlashFiler 2 Engine Manager object.
5. Select the FlashFiler 2 Engine Manager object and click the OK button. The engine manager is added to your project.
6. Place the appropriate components within the engine manager.

Since external clients will be connecting to the server, be sure to add at least one transport and command handler to the engine manager. For an example, see the OrdProc project located in Examples\CBuildr\Mythic\OrdProc or Examples\Delphi\Mythic\OrdProc.

To connect the project's tables to the server engine within the data module, you must do the following:

1. Add a TffClient to your project form or data module. Enter a value for the client's ClientName property.
2. Connect the TffClient.ServerEngine property to the engine manager's server engine.
3. Add a TffSession to your form or data module. Enter a value for the session's SessionName property.
4. Set the TffSession.ClientName property to TffClient.ClientName.
5. Add a TffTable to your form or data module.
6. Set the TffTable.SessionName property to TffSession.SessionName.

All calls made by the table, session, and client are routed through the engine manager's server engine. The previous steps assume you place the client components on a separate form or data module. You could just as well place the client components within the engine manager.

Before the client components may access the server engine, your application must call `TffEngineManager.Startup`. Doing so activates the server engine(s) and transports within the engine manager. When the application is ending, it must call `TffEngineManager.Shutdown`.

## Finding New Uses For Transports

Transports are vital for sending requests to a remote server and, when the requests reach the server, for routing the requests to the right engine. However, they can be used in other ways. For example, a plugin could spontaneously send a message to a client when a particular batch job has finished. The client application could contain a command handler that processes the server's message.

This section uses a Chat Server and Chat Client to demonstrate the bidirectional capabilities of TffLegacyTransport. Along the way, you learn how to create a custom command handler and network messages. The actual source code for this example is in Examples\CBuildr\Chat and Examples\Delphi\Chat.

Let's look at the user interface first. The Chat Server, shown in Figure 8.10, hosts any number of clients using TCP/IP. To start the server, enter a descriptive name in the Server Name field and click the Start button. The memo on its main window logs connections, disconnections, and the messages exchanged between Chat Clients.

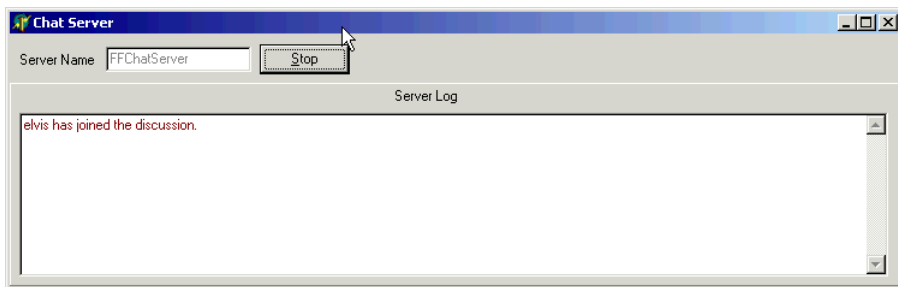


Figure 8.10: The Chat Server main window.

The Chat Client allows you to enter a chat name and connect to an active Chat Server. The memo lists the messages sent to and received from other chat clients. The list box to the right lists the users currently logged into the Chat Server, as illustrated in Figure 8.11.

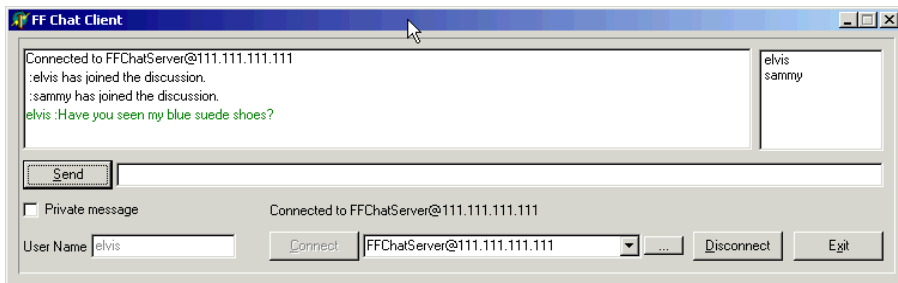


Figure 8.11: The Chat Client main window.

To send a message, enter the message into the entry field next to the Send button and then click the Send button. To send a private message, select a chat user and click the Private button.

## Chat Server

The user interface is the easy part. How do we actually tie together a Chat Server and Chat Client? Let's handle the Chat Server first. In this section, we will walk through the existing code and discuss the important parts.

The major steps in creating the Chat Server are as follows:

1. Add a transport to the server's main window and set the transport's properties.
2. Define the messages and data structures exchanged between client and server.
3. Create a command handler for the server.
4. Implement the command handler's `OnAddClient` and `OnRemoveClient` methods.
5. Implement the command handler's message handlers.
6. Implement the server's user interface methods.

8

### Add the Transport

Open the Chat Server project (declared in `ExChtSrv.DPR`) and click the `TffLegacyTransport` located on the main form. Notice the values of the transport properties in Table 8.1.

**Table 8.1:** *Transport property values.*

Property	Value
Enabled	True
Mode	<code>fftmListen</code>
Protocol	<code>ptTCP/IP</code>
RespondToBroadcasts	True

These property settings enable the transport and place the transport into listen mode. The server supports only those Chat Clients using TCP/IP. Because `RespondToBroadcasts` is set to `True`, the server may be seen by any Chat Client calling `TffLegacyTransport.GetServerNames`. Note that `TffLegacyTransport.GetServerNames` does not see servers across an Internet connection.

## Define the data structures

The Chat Server uses several data structures. All message constants, data structures, and command handlers are centralized in the EXCHTMSG unit. This allows us to share the code between Chat Server and Chat Client.

First, the Chat Server needs to maintain a list of the active clients. The TffLegacyTransport maintains information about the active connections but it does not track the user name associated with each connection. As a result, we define the following record structure.

```
type
  PffChatUser = ^TffChatUser;
  TffChatUser = packed record
    ClientID : TffClientID;
    UserName : TffName;
  end;
```

ClientID is the unique ID assigned to the client by the TffLegacyTransport. The server uses this field to match a user name with a specific connection in the TffLegacyTransport. UserName is the user name entered on the Chat Client window. As you will see later on, the Chat Server command handler maintains a list of TffChatUser records, one per active client.

Other than the messages needed to establish a connection, there are two types of messages traveling between Chat Client and Chat Server. First, the Chat Client may send a text message to the server. The server routes the message to a specific client, if the message is marked private, or to all Chat Clients. Because the server is routing data, we can use one message structure for both purposes. The server uses a second type of message to pass a list of the active clients to each client. The clients require this information in order to specify a recipient for a private message.

Each type of message requires a unique identifier. They are declared at the top of the implementation section in the EXCHTMSG unit and shown in the following lines of code:

```
const
  ffnmChatText = ffnmUser + 100;
  ffnmChatUsers = ffnmUser + 101;
```



The `ffnmUser` constant, declared in the `FFNETMSG` unit, identifies the lower boundary for custom FlashFiler messages. The range below `ffnmUser` is reserved for FlashFiler's use. Each type of message also needs a record structure to describe its contents as well as a pointer type to the record structure. The pointer type is useful for those cases where the structure must be dynamically allocated. The record structure for `ffnmChatText` is as follows:

```
type
  PffnmChatText = ^TffnmChatText;
  TffnmChatText = packed record
    IsPrivate : Boolean;
    UserName : TffName;
    Text : string[255];
  end;
```

The `IsPrivate` field is set to `True` if the client wishes to send a private message to a specific user. If `IsPrivate` is set to `True`, `UserName` must contain the name of the recipient. The `Text` field contains the message text.

When the server receives a text message from a client, it routes the message to one client or to all clients. When a client receives a message from the server, the output of the Chat Client should display the originator of the message. Given these two requirements, the server can use the `TffnmChatText` message as is, as long as it sets `UserName` to the name of the originator.

The record structure for `ffnmChatUsers` is as follows:

```
PffnmChatUsers = ^TffnmChatUsers;
TffnmChatUsers = packed record
  UserList : TffVarMsgField;
end;
```

A message's record structure must have a fixed length. However, in this situation, the number of users is dynamic. To work around this problem, we use a stream to store the data. `UserList` is a placeholder for the stream data. `TffVarMsgField` is declared in the `FFLLBASE` unit and serves as a placeholder for the dynamically sized data. When we send and receive this message, the custom command handlers are responsible for translating the stream data.

## Add a command handler

At this point we have a transport for sending and receiving requests. We've defined the necessary constants and data structures. The next step is to define a command handler that handles the adding and removing of chat clients, the issuing of TffnmChatUsers messages when a client is added or removed, and the forwarding of TffnmChatText messages. The command handler is named TffChatSrvHandler and is declared in the EXCHTMSG unit as follows:

```
TffChatSrvHandler = class(TffBaseCommandHandler)
protected
    FUsers : TffPointerList;
    FMemo : TMemo;
    procedure scInitialize; override;
    procedure scPrepareForShutdown; override;
    procedure scShutdown; override;
    procedure scStartup; override;
    procedure SendUserList(const extraClient : TffClientID);
    procedure nmCheckSecureComms(var Msg : TffDataMessage);
        message ffnmCheckSecureComms;
    procedure nmChatText(var Msg : TffDataMessage);
        message ffnmChatText;
public
    constructor Create(aOwner : TComponent);
    destructor Destroy; override;
    procedure OnAddClient(Sender : TffBaseTransport;
        const userID : TffName; const timeout : longInt;
        const clientVersion : longInt; var passwordHash : TffWord32;
        var aClientID : TffClientID; var errorCode : TffResult;
        var isSecure : Boolean; var aVersion : longInt);
    procedure OnRemoveClient(Sender : TffBaseTransport;
        const aClientID : TffClientID; var errorCode : TffResult);
    procedure Process(Msg : PffDataMessage); override;
    property Memo : TMemo read FMemo write FMemo;
end;
```

FUsers is the list of TffChatUser records. FMemo, accessible via the Memo property, is the TMemo to which the command handler logs connections, disconnects, and forwarded messages. The other important methods in this class are described in Table 8.2.

**Table 8.2:** *TffChatSrvHandler methods.*

Method	Description
scInitialize, scPrepareForShutdown, scShutdown, scStartup	We don't use these methods in this example. However, we must implement them because they are declared as virtual abstract methods in class TffBaseCommandHandler.
OnAddClient	We assign this method to the transport's OnAddClient event. This method is called whenever the transport receives a new connection request.
OnRemoveClient	We assign this method to the transport's OnRemoveClient event. This method is called whenever a client disconnects from the transport.
nmCheckSecureComms	This is the message handler for the ffnmCheckSecureComms message. ffnmCheckSecureComms is declared in the FFNETMSG unit. Once a client connection is established, the default behavior of the TffLegacyTransport is to send a ffnmCheckSecureComms message to the server transport. The purpose of the message is to verify the identity of the client. If the server can properly translate the message then the client is considered to have the correct identity. If we do not implement this method then the client will not be allowed to retain its connection.
nmChatText	This is the message handler for the ffnmChatText message (declared previously in this section). When a client sends a ffnmChatText message, this command handler's Process method routes the message to this message handler. This method is responsible for forwarding the message to the appropriate client(s).
Process	When the transport receives a message, it calls this method. This method is responsible for routing the message to the correct message handler.

## Implement OnAddClient and OnRemoveClient

The `scInitialize`, `scPrepareForShutdown`, `scShutdown`, and `scStartup` methods don't do anything, so we'll ignore them. The `OnAddClient` method is more interesting is shown in the following example:

```
procedure TffChatSrvHandler.OnAddClient(  
    Sender : TffBaseTransport; const userID : TffName;  
    const timeout : longInt; const clientVersion : longInt;  
    var passwordHash : TffWord32; var aClientID : TffClientID;  
    var errorCode : TffResult; var isSecure : Boolean;  
    var aVersion : longInt);  
var  
    aName : TffName;  
    aTransport : TffBaseTransport;  
    Index : longInt;  
    PUserRec : PffChatUser;  
    TextRequest : TffnmChatText;  
  
    { We use this function to determine if another chat user has  
      already registered this user's name. }  
    function NameUsed(const newName : TffName) : Boolean;  
    var  
        Index : longInt;  
    begin  
        Result := False;  
        for Index := 0 to pred(FUsers.Count) do begin  
            Result := (AnsiCompareText(  
                PffChatUser(FUsers.Pointers[Index])^.UserName,  
                newName) = 0);  
            if Result then  
                break;  
        end;  
    end;  
end;
```

```

begin
    passwordHash := 0;
    errorCode := 0;
    isSecure := False;
    aVersion := FFVersionNumber;

    {Does the user have a conflicting name? If so then add a
     numeric suffix.}
    aName := UserID;
    Index := 0;
    while NameUsed(aName) do begin
        inc(Index);
        aName := Copy(UserID, 1, 26) + '(' + IntToStr(Index) + ')';
    end;

    {Generate a clientID and add a new user info record.}
    aClientID := GetTickCount;
    New(PUserRec);
    PUserRec^.UserName := aName;
    PUserRec^.ClientID := aClientID;
    FUsers.Append(PUserRec);

    {Let everybody know about the new user.}
    with TextRequest do begin
        IsPrivate := False;
        UserName := '';
        Text := aName + ' has joined the discussion.';
    end;

    aTransport := TffBaseTransport.CurrentTransport;
    for Index := 0 to pred(aTransport.ConnectionCount) do
        aTransport.Post(
            0, aTransport.ConnectionIDs[Index], ffrmChatText,
            @TextRequest, sizeof(TextRequest),
            1000, ffrmNoReplyExpected);
    {Post an extra message specifically to the new clientID. The
     previous loop will try to send a message to them but they
     won't receive it because it goes to their old clientID.}
    aTransport.post(
        0, aClientID, ffrmChatText, @TextRequest,
        sizeof(TextRequest), 1000, ffrmNoReplyExpected);
    {Send the updated list of users to everyone.}
    SendUserList(aClientID);

    {Log the connection.}
    FMemo.Lines.Add(TextRequest.Text);
end;

```

This code performs several important steps. First, it initializes some of its output variables. In this situation we assume that no security is needed and that all client connections are accepted. Therefore, it sets the error code parameter to zero indicating the connection was successful. It sets `passwordHash` and `isSecure` to `False` indicating no security is necessary. It then sets `VersionNumber` to the current `FlashFiler` version number. This allows us to make sure the transports on either side of the fence are using compatible versions of `FlashFiler`.

If we had wanted to enforce some kind of security, we would set `isSecure` to `True` and return the hashed value of the user's password in `passwordHash`. As mentioned in "Security" on page 25, a user's hashed password is used to encrypt and decrypt messages between client and server.

`OnAddClient` next verifies that no other client is using the `userID` (i.e., user name) requested by the new client. If the user name is already in use, a unique number is suffixed to the requested user name.

`OnAddClient` is responsible for assigning the new client a unique client ID. This method uses `GetTickCount` to obtain a new client ID. If the Chat Server were multi-threaded, this might pose a problem. As is, this Chat Server is single-threaded so each client ID will be unique.

After generating the client ID, this method creates an instance of `TffChatUser` and stores it in `FUsers`. It then builds a `TffnmChatText` request containing text that tells the existing chat users about the new client. The command handler must use the transport to send the requests. The transport is not passed as one of this methods parameters, therefore the command handler must use the class function `TffBaseTransport.CurrentTransport`. This function returns the transport that caused the current thread to process this request.

Once `OnAddClient` obtains the transport, it uses the transport's `ConnectionCount` and `Connections` properties to post the `TffnmChatText` request to each client. Notice that `OnAddClient` takes an additional step to directly route the request to the new client ID. The reason it does so is that the transport is tracking the new client connection under a temporary client ID. When `OnAddClient` returns, the client is assigned the new ID generated within `OnAddClient`. This assignment occurs before the `TffnmChatText` requests are sent out. If we don't specifically post a message to the new client ID, the client would never see the message.

As its final two steps, `OnAddClient` sends a new user list to each client and logs the new connection in its main window.

The OnRemoveClient method, shown in the following example, is responsible for removing a client connection:

```

procedure TffChatSrvHandler.OnRemoveClient
(Sender : TffBaseTransport; const aClientID : TffClientID;
 var errorCode : TffResult);
var
  aTransport : TffBaseTransport;
  Index : longInt;
  PUserRec : PffChatUser;
  TextRequest : TffnmChatText;
begin
  errorCode := 0;

  { Find the user's info record. }
  PUserRec := nil;
  for Index := 0 to pred(FUsers.Count) do
    if PffChatUser(FUsers.Pointers[Index])^.ClientID = aClientID
    then begin
      PUserRec := PffChatUser(FUsers.Pointers[Index]);
      break;
    end;

  { Did we find a user record? }
  if assigned(PUserRec) then begin
    { Yes. Free the user information. }
    with TextRequest do begin
      IsPrivate := False;
      UserName := '';
      Text := PUserRec^.UserName + ' is no more.';
    end;
    FUsers.RemoveAt(Index);
    Dispose(PUserRec);

    { Let everyone know about the disconnect. }
    aTransport := TffBaseTransport.CurrentTransport;
    for Index := 0 to pred(aTransport.ConnectionCount) do
      aTransport.Post(
        0, aTransport.ConnectionIDs[Index], ffnmChatText,
        @TextRequest, sizeof(TextRequest), 1000,
        ffrmNoReplyExpected);
    { Log the disconnection. }
    FMemo.Lines.Add(TextRequest.Text);

    { Send the updated list of users to everyone. }
    SendUserList(0);
  end;
end;

```

This code locates the `TffChatUser` record in `FUsers` based upon the client's unique ID. It then sends a `TffnmChatText` request and new user list to the remaining clients.

## Implement message handlers

The command handler's `Process` method is very simple. As shown in the following code, it routes a message to a message handler using `TObject.Dispatch`. We don't provide a `DefaultHandler` so unrecognized messages fall into the bit bucket.

```
procedure TffChatSrvHandler.Process(Msg : PffDataMessage);
begin
    Dispatch(Msg^);
    bchFreeMsg(Msg);
end;
```

Note that the `Process` method calls `bchFreeMsg`. When a transport passes off a message to a command handler, it becomes the command handler's responsibility to free the data associated with the message. `TffBaseCommandHandler.bchFreeMsg` is provided as a default method for freeing the message data.

The `nmChatText` message handler is called by `TffChatSrvHandler.Process` when the transport receives a `ffnmChatText` request. The job of `nmCharText`, shown in the following lines of code, is to forward a `ffnmChatText` message to the appropriate clients:

```
procedure TffChatSrvHandler.nmChatText(var Msg : TffDataMessage);
var
    aTransport : TffBaseTransport;
    Index : longInt;
    InRequest : PffnmChatText;
    PRecipient : PffChatUser;
    PUserRec : PffChatUser;
begin
    aTransport := TffBaseTransport.CurrentTransport;

    {Get a handle on the incoming message's content.}
    InRequest := PffnmChatText(Msg.dmData);
```



```

{Find the chat user.}
PUserRec := nil;
for Index := 0 to pred(FUsers.Count) do
    if PffChatUser(
        FUsers.Pointers[Index])^.ClientID = Msg.dmClientID then
    begin
        PUserRec := PffChatUser(FUsers.Pointers[Index]);
        break;
    end;
{Did we find the user's record?}
if assigned(PUserRec) then begin
    {Yes. Log the message.}
    FMemo.Lines.Add(PUserRec^.UserName + ': ' + InRequest^.Text);
    {Is this a private message?}
    if InRequest^.IsPrivate then begin
        {Find the recipient's user record.}
        PRecipient := nil;
        for Index := 0 to pred(FUsers.Count) do
            if PffChatUser(FUsers.Pointers[Index])^.UserName =
                InRequest.UserName then begin
                PRecipient := PffChatUser(FUsers.Pointers[Index]);
                break;
            end;
        if assigned(PRecipient) then begin
            {Re-use the inbound request for the outbound message.}
            InRequest.UserName := PUserRec^.UserName;
            aTransport.Post(
                0, PRecipient^.ClientID, ffnmChatText, InRequest,
                sizeof(TffnmChatText), 1000, ffrmNoReplyExpected);
        end;
    end
else begin
    {No. Route it to everyone. Re-use the inbound request for
    the outbound message.}
    InRequest.UserName := PUserRec^.UserName;
    for Index := 0 to pred(aTransport.ConnectionCount) do
        aTransport.Post(
            0, aTransport.ConnectionIDs[Index], ffnmChatText,
            InRequest, sizeof(TffnmChatText),
            1000, ffrmNoReplyExpected);
    end;
end;
end;
end;

```

Before this method can forward the message, it must do three things. First, it must find the transport that forwarded the message to the command handler. It does so using class function `TffBaseTransport.CurrentTransport`. Second, it must obtain the data comprising the request. The message passed by the transport actually contains some header information as well as the original request. The header information, type `TffDataMessage`, is described in “Requests and replies” on page 318. The request content is referenced by the `dmData` field of `TffDataMessage`.

Last, this method must find the `TffChatUser` record associated with the originator of the request. To do so, the method grabs the originator’s client ID from the `TffDataMessage.dmClientID` field.

Once the method has obtained these pieces of information, it determines if the message is private. If so, it finds the `TffChatUser` record associated with `TffnmChatText.UserName` and forwards the message to that client. Otherwise, it forwards the message to each active client. In either case, this method replaces `TffnmChatText.UserName` with the name of the original user. As you will see later, the Chat Client is coded to pull out the user name and display it as the originator of the message.

## Implement server user interface methods

The last step required to implement the Chat Server is to fill out its user interface methods. As you have seen, most of the work is done within the command handler. The one thing that we haven’t done is connect the command handler to the transport. This is performed by the `TfrmServerMain.pbSrvCtrlClick` method:

```
procedure TfrmServerMain.pbSrvCtrlClick(Sender : TObject);
begin
    if tpMain.State = ffesInactive then begin
        efSrvName.Enabled := False;
        pbSrvCtrl.Caption := '&Stop';

        { Create a command handler. }
        FChatHandler := TffChatSrvHandler.Create(Self);
        FChatHandler.Memo := memChat;

        { Connect the command handler to the transport. }
        tpMain.CommandHandler := FChatHandler;
        tpMain.OnAddClient := FChatHandler.OnAddClient;
        tpMain.OnRemoveClient := FChatHandler.OnRemoveClient;
```

```

        { Start the transport. }
        tpMain.ServerName := efSrvName.Text;
        tpMain.State := ffesStarted;
    end
    else begin
        tpMain.State := ffesInactive;
        efSrvName.Enabled := True;
        pbSrvCtrl.Caption := '&Start';
    end;
end;

```

When you click the Start button on the Chat Server's form, the pbSrvCtrlClick routine does the following:

1. Creates an instance of TffChatSrvHandler.
2. Assigns the memChat TMemo to the command handler's Memo property. This ensures all output is routed to the proper memo.
3. Connects the command handler to the transport by assigning the command handler to tpMain.CommandHandler. This tells tpMain to route all requests to the command handler.
4. Assigns TffChatSrvHandler.OnAddClient and OnRemoveClient to the transport's OnAddClient and OnRemoveClient events, respectively.
5. Sets the transport's server name based upon what you typed in the main form.
6. Sets the transport's state to ffesStarted. This activates the transport's protocol thread, allowing the transport to receive requests and post messages to the clients.

At this point, we have a Chat Server that we can uniquely name, start, and stop at will. The routing of client messages is performed in the command handler. Clients are notified when new users connect or old users disconnect. Now it is time to implement the Chat Client.

## Chat Client

The Chat Client takes advantage of the messages and data structures defined in the Chat Server section. Since the bulk of the processing is done within the server-side command handler, the Chat Client is less involved than the Chat Server. The steps to implement the Chat Client are as follows:

1. Define a client-side command handler.
2. Implement the command handler's methods.
3. Mesh the command handler with the user interface.

### Define the command handler

The command handler is named `TffChatClntHandler` and is declared in the `EXCHTMMSG` unit as follows:

```
TffChatClntHandler = class(TffBaseCommandHandler)
protected
    FOutput      : TListBox;
    FUserList    : TListBox;
    procedure scInitialize; override;
    procedure scPrepareForShutdown; override;
    procedure scShutdown; override;
    procedure scStartup; override;
    procedure nmChatText(var Msg : TffDataMessage);
        message ffnmChatText;
    procedure nmChatUsers(var Msg : TffDataMessage);
        message ffnmChatUsers;
public
    procedure Process(Msg : PffDataMessage); override;
    property Output : TListBox read FOutput write FOutput;
    property UserList : TListBox read FUserList write FUserList;
end;
```

The Output property is the list box to which the command handler writes messages received from the Chat Server. The UserList property is the list box in which the command handler displays the user lists received from the Chat Server. The other important methods in this class are described in Table 8.3.

**Table 8.3:** *TfrmServerMain* methods.

Method	Description
scInitialize, scPrepareForShutdown, scShutdown, scStartup	We don't use these methods in this example. Regardless, we must implement them because class <code>TffBaseCommandHandler</code> declares them as virtual abstract.
nmChatText	This is the message handler for <code>ffnmChatText</code> messages received from the Chat Server. The Chat Client prefixes the message with the name of the originator (i.e., <code>TffnmChatText.UserName</code> ) and writes the message to the list box specified by its Output property.
nmChatUsers	This is the message handler for the <code>ffnmChatUsers</code> message received from the Chat Server. The Chat Server sends this message when a new client connects or an existing client disconnects. The Chat Client displays the list of chat users in the list box specified by its UserList property.
Process	When the client-side transport receives a message from the Chat Server, it calls this method. This method is responsible for routing the message to the correct message handler.

The most complicated part of a command handler is deciding what messages it can receive and what data is contained with each method. Once that is done, it is a matter of implementing a Process method and a message handler for each message.

### Implement the command handler methods

The state-related methods, `scInitialize`, `scPrepareForShutdown`, `scShutdown`, and `scStartup`, don't do anything in this example. In your own command handler, you may find them useful to perform initialization or shutdown actions.

The Process method, shown in the following lines of code, is almost identical to the one created for the Chat Server's command handler.

```
procedure TffChatClnHandler.Process(Msg : PffDataMessage);
begin
    Dispatch(Msg^);
    bchFreeMsg(Msg);
end;
```

As shown in the previous code, it uses TObject.Dispatch to route the message to a particular message handler. In the Chat Client, ffnmChatText messages are routed to TffChatClnHandler.nmChatText. ffnmChatUsers messages are routed to TffChatClnHandler.nmChatUsers. Once the message has been routed, it is the responsibility of the command handler to free the message data. It does so by calling TffBaseCommandHandler.bchFreeMsg.

The implementation of nmChatText is as follows:

```
procedure TffChatClnHandler.nmChatText(var Msg :
TffDataMessage);
var
    aMessage : string;
    PRequest : PffnmChatText;
begin
    {Get a handle on the message content.}
    PRequest := PffnmChatText(Msg.dmData);
    if PRequest^.IsPrivate then
        aMessage := ffc_Private
    else
        aMessage := '';

    aMessage :=
        aMessage + PRequest^.UserName + ' : ' + PRequest^.Text;
    FOutput.Items.Add(aMessage);
    FOutput.ItemIndex := pred(FOutput.Items.Count);
    FOutput.ItemIndex := -1;
end;
```

The message received by the transport and passed to this message handler contains header information (e.g., who sent the request, what is the size of the request) as well as the actual TffnmChatText structure forwarded by the Chat Server. The first thing this method does is obtain a handle on the TffnmChatText structure.

If the message is marked as private, the display text is prefixed with constant ffc\_Private. The display text is suffixed with the name of the user sending the message as well as the text entered by that user. The display text is added to the Output list box.

The implementation of nmChatUsers is as follows:

```
procedure TffChatClntHandler.nmChatUsers(
  var Msg : TffDataMessage);
var
  aList : TStringList;
  aStream : TMemoryStream;
  Index : longInt;
  PRequest : PffnmChatUsers;
begin
  {Get a handle on the message content.}
  PRequest := PffnmChatUsers(Msg.dmData);

  {Read the content into a stream.}
  aStream := TMemoryStream.Create;
  try
    aStream.Write(PRequest^, Msg.dmDataLen);
    aList := TStringList.Create;
    try
      aStream.Position := 0;
      aList.LoadFromStream(aStream);

      {Populate the listbox.}
      FUserList.Items.Clear;
      for Index := 0 to pred(aList.Count) do
        FUserList.Items.Add(aList.Strings[Index]);
      finally
        aList.Free;
      end;
    finally
      aStream.Free;
    end;
  end;
end;
```

The method first obtains a handle on the TffnmChatUsers record structure. The Chat Server saved the user list to the message in the form of a stream. This method creates a stream and reads the data from the record structure into the stream. The stream is then transformed back into a TStringList using TStringList.LoadFromStream. Finally, the UserList list box is populated with the contents of the string list.

## Integrate with the user interface

The Chat Client interface contains a number of features. In this section, we will focus on how it knows which Chat Servers are available, how it connects to a Chat Server, and how it sends a message to the Chat Server.

The Chat Client uses TCP/IP to communicate with Chat Server. When Chat Client starts, it sends out a broadcast message in order to find the available servers as shown in the following example:

```
procedure TfrmCltMain.RefreshServers;
var
  Index : longInt;
  ServerList : TStringList;
begin
  { Obtain a list of available chat servers. }
  ServerList := TStringList.Create;
  try
    tpClient.GetServerNames(ServerList, 1000);
    { Populate the combo box. }
    cmbServers.Items.Clear;
    cmbServers.Text := '';
    for Index := 0 to pred(ServerList.Count) do begin
      cmbServers.Items.Add(ServerList.Strings[Index]);
    end;

    if ServerList.Count > 0 then
      cmbServers.ItemIndex := 0
    else
      cmbServers.Text := '<No servers found>';

  finally
    ServerList.Free;
  end;
end;
```

RefreshServers creates an instance of TStringList and passes it to the tpClient.GetServerNames method. The transport does not have to be started in order to broadcast for available servers. If any servers are found, the GetServerNames method adds their names to the TStringList. If any servers are found, their names are placed into the cmbServers combo box and the combo box is positioned to the first server.



Once you choose a Chat Server, clicking the Connect button executes the following code in `TfrmCltMain.pbConnectClick`:

```
procedure TfrmCltMain.pbConnectClick(Sender : TObject);
begin
    {Requirement : Must have specified a user name.}
    if efUserName.Text = '' then begin
        ShowMessage('You must specify a user name.');
```

exit;

```
    end;

    {Requirement : Must have selected a server.}
    if cmbServers.ItemIndex < 0 then begin
        ShowMessage('You must select a chat server.');
```

exit;

```
    end;

    tpClient.ServerName := cmbServers.Items[cmbServers.ItemIndex];
    tpClient.State := ffesStarted;
    tpClient.EstablishConnection(
        efUserName.Text, 0, 1000, FClientID);
    SetCtrlStates;
    if tpClient.IsConnected then begin
        lbOutput.Items.Add('Connected to ' + tpClient.ServerName);
        lbOutput.ItemIndex := pred(lbOutput.Items.Count);
        lbOutput.ItemIndex := -1;
        efMessage.SetFocus;
    end;
end;
```

8

This method requires you to enter a user name and select a server. Assuming you have done so, this method sets `tpClient.ServerName` to the name of the chosen server. This must be done or the transport will have no idea to which server it is to connect. Next, the transport is started by setting its State to `ffesStarted`. Finally, we call the `TffBaseTransport.EstablishConnection` method.

If the transport successfully establishes a connection with the Chat Server, the Chat Server assigns the connection a unique client ID. The unique client ID is returned in variable `FClientID`. Chat Client must use `FClientID` to send requests to the Chat Server.

Once a connection is established to the server, you may type a message into the entry field and click the Send button. Clicking the Send button invokes `TfrmCltMain.pbSendClick` as shown in the following example:

```
procedure TfrmCltMain.pbSendClick(Sender : TObject);
var
    Request : TffnmChatText;
begin
    if efMessage.Text <> '' then begin
        Request.IsPrivate := chkPrivate.Checked;
        Request.UserName := '';

        {Private?}
        if chkPrivate.Checked then begin
            {Yes. Verify a recipient is checked.}
            if lbUsers.ItemIndex < 0 then begin
                showMessage('No one is selected.');
```

This method determines if you actually entered a message and formats the request based upon the state of the Private check box. To actually send the request to Chat Server, this method calls `TffBaseTransport.Post`. We use the `Post` method in this situation only because we don't expect a reply from the server. In other situations, you must decide if it is appropriate for the recipient of a message to reply. If the request is for information then it is given the server sends back a reply in which case you must make use of `TffBaseTransport.Request`.

This example showed you how to implement new functionality using transports and command handlers, apart from the standard FlashFiler Server. In the next section, you learn how to add new functionality using FlashFiler's plugin architecture.

# Creating Plugins

Plugins allow you to add new functionality to the FlashFiler Server. You can even implement your own server applications independent of FlashFiler Server. This section details an error logging plugin found in the Examples\CBuilder\Plugins and Examples\Delphi\Plugins subdirectories of your FlashFiler installation. The purpose of the error logging plugin is to record client application errors within a central log file.

A plugin consists of a plugin engine and a plugin command handler. The plugin engine is analogous to a server engine. The plugin engine provides functionality through a specific interface. In the same way that a server engine can be embedded in a client application, a plugin engine may also be embedded in a client application. Thus, a client application could make direct calls into the plugin engine or communicate with a remote plugin engine.

The plugin command handler is analogous to the server command handler. Commands not recognized by the server command handler are forwarded to the plugin command handlers. If a plugin command handler recognizes the message, it translates the message into a method call on the plugin engine. The plugin command handler is then responsible for replying to the client and freeing the memory associated with the message.

Figure 8.12 illustrates how requests are routed from a client application to a remote plugin engine and to an embedded plugin engine.

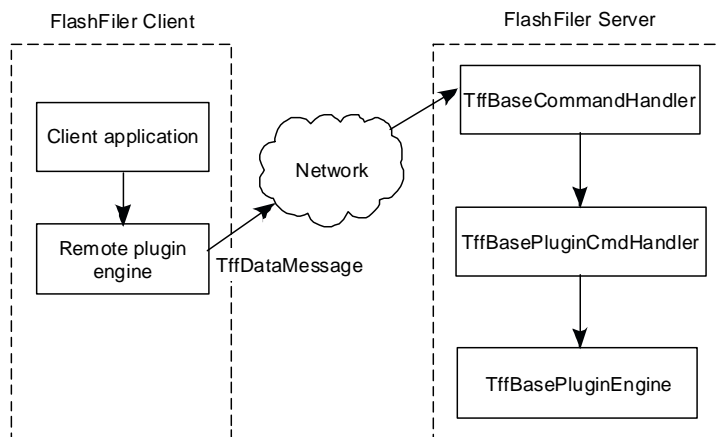


Figure 8.12: The Plugin request route.

To create the error logging plugin, we need to do five things:

1. Create an abstract plugin engine class describing the plugin's interface.
2. Create the real plugin engine that implements the logging functionality.
3. Create a plugin engine, similar to `TffRemoteServerEngine`, that allows a client application to access the plugin across a network.
4. Create a plugin command handler that routes requests from a server command handler to the real plugin engine.
5. Once you have reviewed the four classes, you will learn how the plugin fits into the FlashFiler Server and a client application. The entire plugin is contained in the EXERRPLG unit.

## Create the abstract plugin engine

The abstract class describes the plugin's interface. The real and remote plugin engine classes inherit from this class. The class is declared as follows:

```
TffBaseErrorPlugin = class(TffBasePluginEngine)
protected
    procedure scInitialize; override;
    procedure scPrepareForShutdown; override;
    procedure scShutdown; override;
    procedure scStartup; override;
public
    procedure WriteError(const ClientID : TffClientID;
        const ErrCode : LongInt; const UserName : TffName;
        const Msg : string); virtual; abstract;
end;
```

The `scInitialize`, `scPrepareForShutdown`, `scShutdown`, and `scStartup` methods are instantiated but do nothing. We will use them in the remote plugin engine later. The `WriteError` method, when implemented, writes an error code and error message to the plugin's log component. Because `TffLoggableComponent` is an ancestor of `TffBasePluginEngine`, the plugin already contains the interface for associating it with an instance of `TffBaseLog`.

## Create the real plugin engine

The real plugin engine may be placed in either a server or a client application. It implements the `WriteError` method. The class definition for `TffServerErrorPlugin` is as follows:

```
TffServerErrorPlugin = class(TffBaseErrorPlugin)
protected
public
    procedure WriteError(
        const ClientID : TffClientID; const ErrCode : LongInt;
        const UserName : TffName; const Msg : string); override;
end;
```

The `WriteError` method, shown in the following lines of code, is implemented as described in the previous section.

```
procedure TffServerErrorPlugin.WriteError(
    const ClientID : TffClientID; const ErrCode : LongInt;
    const UserName : TffName; const Msg : string);
const
    errMsgFormat = '%d %s %s [%d]';
    { <clientID> <userName> <message> [error code]
      Since we are using TffEventLog, the entire string is
      prefixed with the value of GetTickCount and the current
      thread ID. }
begin
    FEventLog.WriteStringFmt(
        errMsgFormat, [ClientID, UserName, Msg, ErrCode]);
end;
```

## Create the remote plugin engine

The remote plugin engine is a bit more interesting. Its job is to connect a client application to the real plugin engine which is sitting in a FlashFiler Server somewhere. The declaration of `TffRemoteErrorPlugin` is as follows:

```
TffRemoteErrorPlugin = class(TffBaseErrorPlugin)
protected
    FClientID : TffClientID;
    { The clientID returned when a connection is established. }
    FTransport : TffBaseTransport;
    procedure ceSetTransport(aTransport : TffBaseTransport);
    { Called when setting the Transport property. }
    procedure Notification(
        AComponent : TComponent; Operation : TOperation);
    { Called when FTransport is deleted in the IDE. Allows us
      to nil out FTransport. }
```

```

    procedure scPrepareForShutdown; override;
    procedure scStartup; override;
public
    constructor Create(aOwner : TComponent); override;
    procedure WriteError(
        const ClientID : TffClientID; const ErrCode : LongInt;
        const UserName : TffName; const Msg : string); override;
published
    property Transport : TffBaseTransport
        read FTransport write ceSetTransport;
        {The transport through which this class talks to the
         remote server.}
end;

```

A remote plugin engine must communicate with the real plugin engine via a transport. Therefore, we declare a Transport property. We use the Notification method to identify the case where the transport has been freed (e.g., deleted from the IDE).

The scStartup method establishes a connection with the remote server, as shown in the following example:

```

procedure TffRemoteErrorPlugin.scStartup;
var
    Result : TffResult;
begin
    {Assumption : No login required.}
    Result := FTransport.EstablishConnection(
        '', 0, 5000, FClientID);
    if Result <> DBIERR_NONE then
        raise Exception.Create(
            'Error plugin could not establish connection.');
```

end;

ScStartup is called when the engine's state changes to ffeStarted. Note that we store the client ID in internal variable FClientID. We will use FClientID when sending the error message to the remote server.

ScPrepareForShutdown is called when the engine's state is set to ffeShuttingDown. The scPrepareForShutdown method terminates the connection with the server. The scPrepareForShutdown method is as follows:

```

procedure TffRemoteErrorPlugin.scPrepareForShutdown;
begin
    FTransport.TerminateConnection(FClientID, 5000);
end;

```

Assuming a connection is established, the `WriteError` method translates its input parameters into a `ffnmLogError` request and sends it to the remote server as in the following example:

```

procedure TffRemoteErrorPlugin.WriteError(
    const ClientID : TffClientID; const ErrCode : LongInt;
    const UserName : TffName; const Msg : string);
var
    MsgSize : integer;
    Request : PffnmLogError;
    ReqLen : integer;
begin
    {Requirement : Transport must be assigned.}
    if not assigned(FTransport) then
        raise Exception.Create(
            'Must specify a transport in order to log errors.');
```

8

```

    {Requirement : Plugin must be started.}
    if FState <> ffesStarted then
        raise Exception.Create('Plugin must be started.');
```

{Build the request. We must dynamically allocate memory since the string may be of variable size.}

```

    MsgSize := Length(Msg);
    ReqLen := SizeOf
        (TffnmLogError) + Length(Msg) - SizeOf(TffVarMsgField);
    FFGetMem(Request, ReqLen);
    try
        Request^.UserName := UserName;
        Request^.ErrorCode := ErrCode;
        Request^.MsgSize := MsgSize;
        Move(PChar(Msg)^, Request^.Msg, MsgSize);
        FTransport.Post(
            0, FClientID, ffnmLogError, Request, ReqLen, 1000,
            ffrmNoReplyExpected);
    finally
        FFFreeMem(Request, ReqLen);
    end;
end;
```

Notice that we must dynamically allocate memory for the request. The reason we do so is because the error message being logged can be of any size. After sizing the request, the method fills in the static fields (e.g., `UserName`, `ErrorCode`, `MsgSize`) and then copies the text of the message into the remainder of the request buffer. We then post the request to the transport. Because `Post` does not wait for a reply, execution begins immediately after `FTransport.Post`.

## Create the plugin command handler

At this point, we have a real plugin engine that implements the logging of errors. We also have a remote plugin engine that sends a logging request to the real plugin engine via a transport. We need a plugin command handler to bridge the gap between the remote plugin engine and the real plugin engine. Following is the declaration of `TffErrorPluginCmdHandler`:

```
TffErrorPluginCmdHandler = class(TffBasePluginCommandHandler)
protected
    function epcGetPluginEngine : TffServerErrorPlugin;

    procedure epcSetPluginEngine(
        anEngine : TffServerErrorPlugin);

    procedure nmLogError(
        var Msg : TffDataMessage); message ffnmLogError;
        {Called by Process for ffnmLogError message.}

    procedure scInitialize; override;

    procedure scPrepareForShutdown; override;

    procedure scShutdown; override;

    procedure scStartup; override;

public
    procedure Process(
        Msg : PffDataMessage; var handled : Boolean); override;
        {This method is called by a command handler when it has a
        message that may be processed by a plugin. If the plugin
        handles the message, set handled to True.}
published
    property PluginEngine : TffServerErrorPlugin
        read epcGetPluginEngine write epcSetPluginEngine;

end;
```

The purpose of this command handler is to translate an incoming `ffnmLogErrors` request into a call to `TffServerErrorPlugin.WriteError`. In order to do so, it needs to know which plugin engine it is using. We connect the command handler to the plugin engine via the `PluginEngine` property.

When you create a plugin command handler, you must create a `PluginEngine` property. `TffBasePluginCommandHandler` declares a private `FPluginEngine` variable of type `TffBasePluginEngine`. `TffBasePluginCommandHandler` does not have a `PluginEngine` property because we want to match up plugin command handlers with the right type of plugin engine. We don't want a `TffErrorPluginHandler` connected to a `TffFormatHardDrivePlugin`.



When implementing the `PluginEngine` property, we create read and write methods that cast `FPluginEngine` to the appropriate type. Following is the implementation of `epcGetPluginEngine`, used to read the `PluginEngine` property:

```
function TffErrorPluginCmdHandler.epcGetPluginEngine :  
    TffServerErrorPlugin;  
begin  
    Result := TffServerErrorPlugin(FPluginEngine);  
end;
```

Following is the implementation of `epcSetPluginEngine`, used to set the `PluginEngine` property:

```
procedure TffErrorPluginCmdHandler.epcSetPluginEngine(  
    anEngine : TffServerErrorPlugin);  
begin  
    if assigned(FPluginEngine) then  
        FPluginEngine.RemoveCmdHandler(Self);  
    if assigned(anEngine) then anEngine.AddCmdHandler(Self);  
end;
```

## 8

Notice that two operations are taking place in `epcSetPluginEngine`. First, if the command handler is already connected to a plugin engine then we use `TffBasePluginEngine.RemoveCmdHandler` to de-register the command handler. Second, if the command handler is being connected to a new plugin engine, we use `TffBasePluginEngine.AddCmdHandler` to register the command handler. This allows the plugin engine to maintain a list of command handlers. The plugin engine is then able to set the state of its command handlers when the engine's state changes.

More than one plugin may reside within the server, each with its own plugin command handler. When the server command handler comes across an unrecognizable request, it passes it off to the plugin command handlers. The plugin command handler's `Process` method determines whether it can handle the request. Following is the implementation of the `Process` method:

```
procedure TffErrorPluginCmdHandler.Process(  
    Msg : PffDataMessage; var handled : Boolean);  
begin  
    handled := (Msg^.dmMsg = ffnmLogError);  
    if handled then  
        Dispatch(Msg^);  
        { Note: We don't have to free the message data because  
          a) we were called by TffBaseCommandHandler.Process and  
          b) that particular method frees the data for us. }  
end;
```

This plugin command handler recognizes the `ffnmLogError` message. If it is passed such a message, it then uses `TObject.Dispatch` to call `nmLogError`. We could call `nmLogError` directly but using `Dispatch` allows us to add new messages later with fewer changes.

The `nmLogError` method is as follows:

```
procedure TffErrorPluginCmdHandler.nmLogError(  
    var Msg : TffDataMessage);  
var  
    aMsg : String;  
    aStream : TMemoryStream;  
begin  
    with Msg, PffnmLogError(dmData)^ do begin  
        aStream := TMemoryStream.Create;  
        try  
            aStream.Write(Msg, MsgSize);  
            aStream.Position := 0;  
            SetLength(aMsg, MsgSize);  
            aStream.Read(aMsg[1], MsgSize);  
            TffServerErrorPlugin(FPluginEngine).WriteError(  
                dmClientID, ErrorCode, UserName, aMsg);  
        finally  
            aStream.Free;  
        end;  
    end;  
end;  
end;
```

The remote plugin engine wrote the error message into the request buffer. This method retrieves the error message by reading it into a stream and then writing the stream into a variable of type `String`. It then calls the real plugin engine's `WriteError` method. The request has been translated into a method call.

## Integrate the plugin

We now have three plugin components. Install them using the `EXPLUG` package located in `Examples\Delphi\Plugins`. Our next task is to integrate the real plugin engine and plugin command handler into the FlashFiler Server using the following steps:

1. Open the FlashFiler Server project (declared in `FFSERVER.DPR`).
2. Open the engine manager data module (declared in the `UFFEGMGR` unit).
3. Place an instance of `TffErrorPluginCmdHandler` into the engine manager.
4. Connect the plugin command handler to a server command handler using the `TffErrorPluginCmdHandler.CommandHandler` property. All requests not recognized by the server command handler are passed on to the plugin command handler.

5. Place an instance of `TffServerErrorPlugin` into the engine manager.
6. Connect the plugin command handler to the plugin engine using the `TffErrorPluginCmdHandler.PluginEngine` property. When the plugin command handler receives an `ffnmLogError` request, it now calls the `WriteError` method of the plugin engine.
7. If you want the error messages to be written to the standard `FFServer.log`, connect the plugin engine to the `EventLog` component in the engine manager. Otherwise, place a new `TffEventLog` in the engine manager. Connect the plugin engine to the new event log.
8. Set the plugin engine's `EventLogEnabled` property to `True`. Make sure the event log's `Enabled` property is set to `True`.
9. Save your changes.
10. Recompile and start the FlashFiler Server.

When the FlashFiler Server starts the engine manager, the engine manager in turn starts the server engine and the plugin engine. The FlashFiler Server can now receive error logging requests. Let's use a sample project to test the error logging. Open the project `EXPLUGDM`. It resides in subdirectories `Examples\CBuilder\Plugins` and `Examples\Delphi\Plugins`. The user interface of this example program is shown in Figure 8.13.

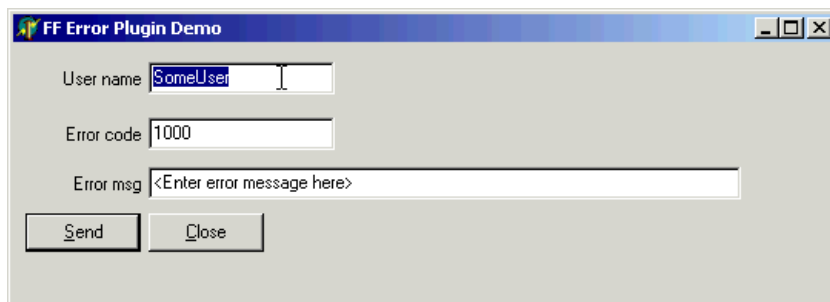


Figure 8.13: The Error Plugin Demo main window.

To log an error with the FlashFiler Server, specify a user name, an error code, and an error message. Next, click the Send button. The error message is written to the file associated with the plugin engine's event log. The following example shows an implementation of Send:

```
procedure TfrmErrPlugMain.pbSendClick(Sender : TObject);
begin
    tpMain.State := ffesStarted;
    try
        errPlugin.State := ffesStarted;
        errPlugin.WriteError(
            0, StrToInt(efErrCode.Text),
            efUserName.Text, efErrMsg.Text);
    finally
        errPlugin.State := ffesInactive;
        tpMain.State := ffesInactive;
        efErrMsg.SetFocus;
    end;
end;
```

It starts the transport and remote error plugin. This method then calls the remote error plugin's WriteError method. Finally, it shuts down the error plugin and transport. Note that you could just as easily add a real plugin engine to this form. If you gave it the same name (i.e., errPlugin) then you would not have to change any code in the pbSendClick method. The only difference is that error messages would be logged to the local hard drive instead of the drive where the FlashFiler Server resides.

In this example, you learned how to create a real plugin engine. You learned how to create a remote plugin engine that sends requests to the real plugin engine. In order for the requests to reach the real plugin engine, you created a plugin command handler that translated the requests into something understandable by the real plugin engine. Finally, you learned how to place the plugin into the FlashFiler Server. In the next example, you will learn how to extend FlashFiler Server's existing functionality.

---

# Extending the Server Engine

Engine monitors and extenders allow you to expand and participate in the functionality of the FlashFiler Server. For example, if you wanted to audit the changes made by personnel to a specific table then you could create a cursor monitor that logs changes as they occur. If you licensed your product by number of concurrent users, you could create a client extender that limits the number of active clients.

It doesn't take much work to extend the server engine. The major steps are as follows:

1. Create an engine monitor.
2. Create an engine extender.
3. Integrate the engine monitor into the FlashFiler Server.

The rest of this section details a cursor monitor that beeps every time a record is inserted, updated, or deleted. The code for the cursor monitor is in Examples\CBuildr\Extend\Cursor and Examples\Delphi\Extend\Cursor in the EXCURMON unit. Also, loading the package EXEXTDRS in subdirectory Examples\CBuildr\Extend or Examples\Delphi\Extend will register the example monitors in the IDE.

8

## Creating an engine monitor

An engine monitor registers interest in one or more server object classes. For example, a monitor that interacts with clients registers interest in the class TffSrClient. For a full discussion of server object classes and other monitor-related topics, see "Event monitoring" on page 441.

The cursor monitor in this example registers interest in TffSrBaseCursor, as shown in the following code:

```
procedure TffCursorMonitor.bemSetServerEngine(anEngine :  
  TffBaseServerEngine);  
begin  
  inherited;  
  AddInterest(TffSrBaseCursor);  
end;
```

BemSetServerEngine is called when a server engine is connected to an engine monitor. At that point, the cursor monitor tells the engine that the monitor is interested in instances of TffSrBaseCursor. Note that a single monitor may register interest in any or all of the legitimate server object classes.

When the server engine creates an instance of `TffSrBaseCursor`, which could be a table cursor or a SQL cursor, it calls the engine monitor's `Interested` method. The following example shows an implementation of the `Interested` method:

```
function TffCursorMonitor.Interested(
    aServerObject : TffObject) : TffBaseEngineExtender;
begin
    Result := nil;
    if (aServerObject is TffSrBaseCursor) then
        Result := TffCursorExtender.Create(Self);
    end;
```

This is the monitor's chance to decide if it is definitely interested in a particular cursor. In this example, the monitor is interested in all cursors. In other situations, it may be more appropriate to check some other attribute, such as the name of the table for which the cursor was opened.

When a monitor decides it is interested in a particular server object, it must create and return an instance of `TffBaseEngineExtender`. If it is not interested, it merely returns `nil`. In this example, the monitor returns an instance of `TffCursorExtender`.

### Creating an engine extender

Engine extenders are attached to the server object passed to the monitor's `Interested` method. The extenders stay with the server object until the server object is destroyed. The declaration of the `TffCursorExtender` class is very simple and is shown in the following lines of code:

```
type
    TffCursorExtender = class(TffBaseEngineExtender)
    protected
    public
        constructor Create(aOwner : TffBaseEngineMonitor); override;
        function Notify(serverObject : TffObject;
            action : TffEngineAction) : TffResult; override;
    end;
```

As certain actions affect the cursor, the cursor or server engine tells the extender what action has occurred. At that point, the extender has the chance to do something. Different extenders are interested in different actions. It would be very wasteful to notify every extender of every action happening to its host object. Therefore, extenders use an `InterestedActions` property to identify those actions about which it should be notified.

Extenders store the set of actions within an internal variable named `FActions`. The `Create` method for this extender initializes the value of the `FActions` as shown in the following example:

```
constructor TffCursorExtender.Create(  
    aOwner : TffBaseEngineMonitor);  
begin  
    inherited;  
    FActions := [  
        ffeaAfterRecDelete, ffeaAfterRecInsert, ffeaAfterRecUpdate];  
end;
```

This particular extender wants to be notified after a record is deleted, inserted, or updated. When one of the actions occurs, the host object or server engine calls the extender's `Notify` method as in the following lines of code:

```
function TffCursorExtender.Notify(  
    serverObject : TffObject; action : TffEngineAction) :  
    TffResult;  
begin  
    Result := DBIERR_NONE;  
    if serverObject is TffSrBaseCursor then  
        case action of  
            ffeaAfterRecDelete :  
                begin  
                    beep;  
                    sleep(100);  
                    beep;  
                    sleep(100);  
                    beep;  
                end;  
            ffeaAfterRecInsert :  
                beep;  
            ffeaAfterRecUpdate :  
                begin  
                    beep;  
                    sleep(200);  
                    beep;  
                end;  
        end; { case }  
    end;  
end;
```

This extender's `Notify` method issues one or more beeps depending upon what has happened to a record. Your own extenders may do whatever is necessary, as long as it is not too time-consuming or disruptive to the host object. A notification may occur within the context of a transaction so it is best to limit the amount of time required to complete the `Notify` method. Be careful that `Notify`'s work does not cause an overall decrease in performance of the server.

For a list of possible actions, see the “Engine notifications” on page 443.

## Integrating into the FlashFiler Server

After the engine monitor and extender have been created, the engine monitor may be registered as a component. Register this example by installing the package `EXEXTDRS` in `Examples\CBuilder\Extend` or `Examples\Delphi\Extend`. Once you have the cursor monitor in your tool palette, add it to the FlashFiler Server as follows:

1. Open the FlashFiler Server project (declared in `FFSERVER.DPR`).
2. Open the engine manager data module (declared in the `UFFEGMGR` unit).
3. Place an instance of `TffCursorMonitor` into the engine manager.
4. Connect the cursor monitor to the server engine via the cursor monitor's `ServerEngine` property.
5. Save your changes.
6. Recompile and start the FlashFiler Server.

Once the FlashFiler Server starts, run FlashFiler Explorer. Open a table and insert a record. As soon as the record is inserted, the server beeps. If you modify the same record, you hear two beeps. If you delete the record, you hear three beeps.

In this section, you learned how to participate in the actions taken by the server engine. If you need further information about engine monitors, engine extenders, or their base classes, see “Chapter 12: Plugins and Extenders” on page 439.





---

## Chapter 9: Transports and Threads

This chapter is for developers using the `TffLegacyTransport` component provided with FlashFiler, for those interested in understanding how transports work, and for those interested in creating custom transports.

In FlashFiler 1, the client and server communications are hidden. You can control the protocol used by a client via a `TffCommsEngine` component and you can control the server's protocol. However, it is very difficult to add a custom protocol or modify an existing protocol.

In FlashFiler 2, communications are handled by transports. Transports serve as a conduit for requests and replies but they do not contain any logic specific to processing requests or replies. Any given transport may be used in an application (i.e., client) and the FlashFiler Server. You can even take FlashFiler Server out of the loop, using transports to connect two of your own applications. A transport is thread-safe and supports one or more connections between client and server.

A client-side transport uses a protocol to communicate with a server-side transport understanding the same protocol. The protocol being used depends upon the type of transport. In FlashFiler 1.x, a `TffCommsEngine` can use the TCP/IP, IPX/SPX, or Single User Protocol (SUP) for communication. In FlashFiler 2, these protocols are now bundled into the `TffLegacyTransport` component.

## Transport modes

A transport has a Mode property. The mode identifies whether the transport is sending requests to a server or processing requests received from a client. A transport sending requests to a server must be in send mode (i.e., `fftmSend`). A transport receiving requests from a client must be in listen mode (i.e., `fftmListen`).

## Transport states

Before a transport can communicate with a like transport, the following requirements must be met:

- You must set the transport's Enabled property to True.
- You must activate the transport by setting its State property to `ffesStarted`.

A transport may be in one of several states as shown in Figure 9.1.

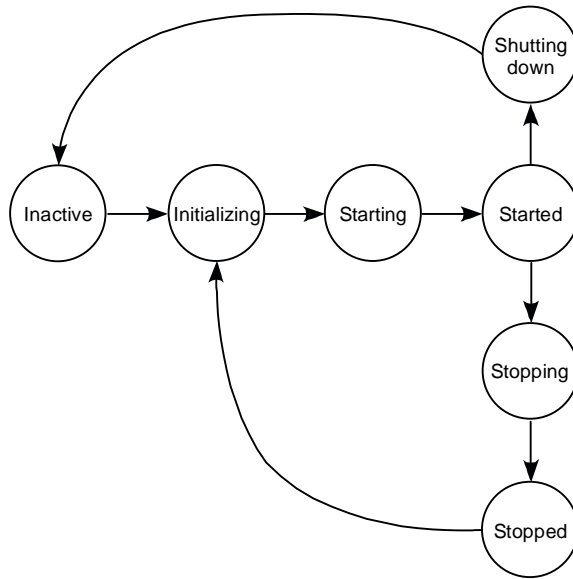


Figure 9.1: Transport State diagram.

The circles represent states. Lines between the circles represent the path taken by the transport's internal state engine as it goes from a start state to a destination state. For example, when a transport is first created its State is set to `ffesInactive`. Setting the State to `ffesStarted` causes the state engine to change the state from `ffesInactive` to `ffesInitializing`, then to `ffesStarting`, and finally to `ffesStarted`.

As a transport progresses from one state to the next, the state engine calls protected methods within the transport. The protected methods allow a transport to perform initialization, start-up, and shutdown actions. See the `State` property of the `TffStateComponent` class for more information. Table 9.1 describes each state.

**Table 9.1:** *Transport States*

State	Meaning
<code>ffesInactive</code>	The transport is not doing any work. This is the default state for all transports.
<code>ffesInitializing</code>	The transport is being prepared for work.
<code>ffesStarting</code>	The transport is starting.
<code>ffesStarted</code>	The transport has successfully started and is ready for work.
<code>ffesShuttingDown</code>	The transport has been told to shutdown.
<code>ffesStopping</code>	The transport does not respond to this state. This state is used by <code>TffEngineManager</code> to temporarily halt a <code>TffBaseServerEngine</code> .
<code>ffesStopped</code>	The transport does not respond to this state. This state indicates a <code>TffBaseServerEngine</code> is temporarily halted.
<code>ffesUnsupported</code>	The operating system and/or drivers on the workstation do not support the transport.
<code>ffesFailed</code>	An error occurred during start-up of the transport. The transport may not be used again until the application is restarted.

## Transport connections

Client transports may establish connections with only one server transport. Server transports host connections from multiple client transports.

Once you have a client-side transport and a server-side transport, the client-side transport must initiate a connection with the server-side transport. Before it can initiate a connection, the client-side transport must know the address of the server-side transport.

One way to obtain the address of all available server transports is to use the `GetServerNames` method of the `TffBaseTransport` class. Depending upon the protocol, `GetServerNames` sends out a broadcast message. All transports using the same protocol and configured to respond to broadcast messages (see the `RespondToBroadcasts` property of the

TffBaseTransport class) will send back a reply. The application controlling the client-side application may then decide with which server-side transport to connect and set the transport's ServerName property appropriately.

If a broadcast for servers does not suit the situation, the client application may store the server's address (which varies depending upon the protocol) in a configuration file, the registry, etc. and set the transport's ServerName property prior to initiating a connection.

Once the server-side transport's address is known, call the EstablishConnection method of the TffBaseTransport class. If the server transport accepts the connection, it issues a unique client ID to the client transport. The client ID must be used in all subsequent communications between the client and server transports.

As mentioned previously, most transports allow you to establish multiple connections between a client transport and a server transport. Each connection has its own unique client ID.

When a client application no longer needs its connection with the server, call the TerminateConnection method of the TffBaseTransport class.

## Requests and replies

Once a connection is established, the client transport typically sends requests to the server transport. If the client transport expects a reply from the server, use the Request method. If no reply is expected, use the Post method.

The server transport hands the requests off to some facility for processing. For example, the TffLegacyTransport component hands off each request to a TffServerCommandHandler which then routes the request to a TffServerEngine.

If a reply is expected, the server application sends a reply to the client via the server transport. An error code within the reply tells the client application whether the request was successfully processed.

FlashFiler 2 uses a standard format for requests and replies. The format, declared as type `TffDataMessage` in the `FFLLCOMM` unit, is shown in Table 9.2.

**Table 9.2:** *The `TffDataMessage` field definitions.*

Field	Request	Reply
<code>dmMsg</code>	The unique identifier for the message. The unique identifier tells the server what kind of request has been received. FlashFiler defines its message identifiers in the <code>FFNETMSG</code> unit.	The reply must contain the message identifier from the original request.
<code>dmClientID</code>	The client identifier assigned to the client transport when it called the <code>EstablishConnection</code> method.	The reply must contain the client identifier from the original request.
<code>dmRequestID</code>	An identifier assigned to the request by the client application. The purpose of this identifier is to allow the client to accurately match a reply with the original request. This identifier is unique to the client but may be duplicated across clients. The client application may generate this identifier in any manner it deems appropriate.	The reply must contain the identifier from the original request.
<code>dmTime</code>	From the server's point of view, this is the time the request was received. It may be set to zero on the client-side. The server is free to use this field as it sees fit. The <code>TffLegacy</code> Transport sets this field to the return value of <code>GetTickCount</code> .	The reply value is set by the server transport when it initially received the request.
<code>dmRetryUntil</code>	The point in time up to which the server may try to fulfill the request. The transports provided with FlashFiler do not use this field. It is provided for your convenience.	The reply value depends upon the transport's implementation.

**Table 9.2:** *The TffDataMessage field definitions. (continued)*

Field	Request	Reply
dmErrorCode	The request should initialize this field to zero or DBIERR_NONE (defined as zero in the FFSRBDE unit). This field indicates whether the request was successfully processed.	If the request was successfully processed, the reply must contain zero or DBIERR_NONE. Otherwise, this field should contain an error code describing the problem.
dmData	The data associated with the request. The format of the request is specific to the client and server applications. If no data is required, set this field to nil.	The data associated with the reply. The format of the reply is specific to the client and server applications. If no data is being returned, set this field to nil.
dmDataLen	The length of the request data. If no data is being sent, set this field to zero.	The length of the reply data. If no data is being sent, set this field to zero.

---

# TffBaseTransport Class

The TffBaseTransport class defines the basic interface for a FlashFiler transport. If you create your own transport, are implementing your own support for multiple threads, or your protocol of choice includes support for multiple threads, inherit from this class. Otherwise, consider inheriting from TffThreadedTransport.

## Hierarchy

TComponent (VCL)	
❶ TffComponent (FLLBASE) .....	78
❷ TffLoggableComponent (FLLCOMP) .....	80
❸ TffStateComponent (FLLCOMP) .....	82
TffBaseTransport (FLLCOMM)	



## Properties

CommandHandler	EventLogOptions	ServerName
Enabled	Mode	❸ State
❷ EventLog	MsgCount	UpdateCount
❷ EventLogEnabled	RespondToBroadcasts	

## Methods

BeginUpdate	EstablishConnection	ResetMsgCount
BtBeginUpdatePrim	❶ FreeInstance	❸ ScInitialize
BtCheckListener	GetName	❸ ScPrepareForShutdown
BtCheckSender	GetServerNames	❸ ScShutdown
BtCheckServerName	IsConnected	❸ ScStartup
BtEndUpdatePrim	❷ LcLog	Sleep
BtInternalReply	❶ NewInstance	Supported
CancelUpdate	Post	TerminateConnection
ConnectionCount	Process	Work
CurrentTransport	Reply	
EndUpdate	Request	

## Events

OnAddClient	OnRemoveClient
OnConnectionLost	❸ OnStateChange

## Reference Section

### BeginUpdate

method

```
procedure BeginUpdate;
```

↳ Notifies the transport that its properties are about to be modified.

Use this method to tell the transport that its properties are about to be modified. All subsequent property modifications are stored in buffer variables. The modifications are applied to the component when you call the EndUpdate method. If an error occurs during the setting of properties, use the CancelUpdate method to discard the property modifications.

The FlashFiler Server uses this scheme to set transport property values stored in server table FFSINFO.

When this method is called, it calls the virtual BtBeginUpdatePrim method so that inherited classes may initialize the buffer variables specific to their implementation. This method then initializes the buffer variables specific to TffBaseTransport.

Calls to BeginUpdate are reference counted. Each call to BeginUpdate must have a corresponding call to EndUpdate. The buffer variables are copied to the transport's properties only after EndUpdate has been called for the last time.

The following example shows how this method may be used:

```
with TCPTransport do begin
  BeginUpdate;
  try
    ServerName := aServerName;
    RespondToBroadcasts := True;
    Mode := ffftmListen;
  except
    CancelUpdate;
  end;
```

See also: BtBeginUpdatePrim, BtEndUpdatePrim, CancelUpdate

```
procedure BtBeginUpdatePrim; virtual;
```

↳ Initializes buffer variables in inherited classes.

This method is called by the `BeginUpdate` method. If you are creating a custom transport, use this method to initialize buffer variables specific to your transport. Each transport property has an associated buffer variable. When `EndUpdate` is called, the value of the buffer variable is assigned to the property. By initializing the buffer variable, you ensure the buffer variable has a valid value even though it was not changed during the update.

See also: `BeginUpdate`, `BtEndUpdatePrim`, `CancelUpdate`, `EndUpdate`

```
procedure BtCheckListener;
```

↳ Verifies the transport's `Mode` property is set to `fftmListen`.

Use this method in your own transports to verify the transport is in listen mode. If the transport is not in listen mode, this method raises an instance of exception `EffServerComponentError` having error code `ffsce_MustBeListening`.

The following example shows how this method may be used:

```
procedure TffTransport.SendReply;
begin
    scCheckStarted;
    btCheckListener;
    ...
end;
```

See also: `BtCheckSender`, `Mode`

```
procedure BtCheckSender;
```

↳ Verifies the transport's Mode property is set to ftfmSend.

Use this method in your own transports to verify the transport is in send mode. If the transport is not in send mode, this method raises an instance of exception `EffServerComponentError` having error code `ffsce_MustBeSender`.

The following example shows how this method may be used:

```
procedure TffTransport.Post;
begin
    scCheckStarted;
    btCheckSender;
    ...
end;
```

See also: `BtCheckListener`, `Mode`

```
procedure BtCheckServerName;
```

↳ Verifies the transport's ServerName property has a value.

Use this method in your own transports to verify the transport's ServerName property has been specified. If the transport requires a server name and the server name is an empty string, this method raises an instance of exception `EffServerComponentError` having error code `ffsce_MustHaveServerName`.

The following example shows how this method may be used:

```
function TffTransport.EstablishConnection : TffResult;
begin
    btCheckSender;
    btCheckServerName;
    scCheckStarted;
    ...
end;
```

See also: `ServerName`

```
procedure BtEndUpdatePrim; virtual;
```

↳ Transfers the buffer variables into an inherited transport's properties.

This method is called by the EndUpdate method. If you are creating a custom transport, override this method. For properties specific to your transport, this method must copy the value of the associated buffer variable into the property. Do not assign the value directly to a property's internal variable. Instead, assign it via the property so that any processing associated with the property's write method may occur.

See also: BeginUpdate, BtBeginUpdatePrim, CancelUpdate, EndUpdate

```
procedure BtInternalReply(msgID : longint; errorCode : TffResult;
    replyData : pointer; replyDataLen : LongInt); virtual;
```

↳ Sends a reply to a client application.

This method is called from the Reply method. The base implementation verifies the transport is in listen mode and is started. If you create your own transport, override this method such that it sends the reply back to the client.

msgID is the unique ID of the message type originally sent by the client.

errorCode must be DBIERR\_NONE (in the FFSRBDE unit) if the request was processed without any problems. Otherwise, set errorCode to the error code associated with the problem. replyData is a pointer to the data that is to be sent to the client. replyDataLen is the length of the data pointed to by replyData.

See also: Reply

```
procedure CancelUpdate;
```

↳ Discards the property modifications applied to a transport.

Use this method to notify the transport that all property modifications made since the first call to BeginUpdate are no longer applicable.

See also: BeginUpdate, BtBeginUpdatePrim, BtEndUpdatePrim, EndUpdate

## CommandHandler

property

```
property CommandHandler : TffBaseCommandHandler
```

↳ Defines the command handler to which a listening transport sends requests.

This property can only be changed if the transport is inactive and is applicable only if the transport is in listen mode. When a transport receives a request from a client, it passes the request to the CommandHandler.

See also: Mode

## ConnectionCount

virtual method

```
function ConnectionCount : longInt; virtual;
```

↳ Returns the number of established connections.

For a transport in send mode, this returns the number of connections established with the remote server. For a transport in listen mode, this returns the number of connections established by remote clients.

See also: EstablishConnection, Mode

## CurrentTransport

method

```
class function CurrentTransport : TffBaseTransport;
```

↳ Returns the transport used by the current thread.

This function is applicable only for transports in listen mode. When a listening transport receives a request, the transport stores itself in thread-local variable `ffitvTransportID`. It then passes the request off to a command handler.

At some point, the command handler must send a reply through the transport back to the client. The command handler doesn't have to know which transport passed it the request. Instead, the command handler calls class function `TffBaseTransport.Reply`. The Reply method uses `CurrentTransport` to determine which transport initiated the request for the current thread.

See also: CommandHandler, Reply

## Enabled

property

property Enabled : Boolean

Default: False

↳ Identifies whether the transport may send or receive requests.

To allow a transport to send or receive requests, set this property to True. A disabled transport will always be in an inactive state. If this property is set to False while the transport is active, the transport's State is set to `ffesInactive`.

See also: Mode, `TffStateComponent.State`

## EndUpdate

method

procedure EndUpdate;

↳ Tells the transport that its property modifications are complete and should be applied.

Use this method to tell the transport that its property modifications are complete. Calls to `BeginUpdate` are reference counted. Therefore, each call to this method must have had a prior call to `BeginUpdate`. When the last call to `EndUpdate` occurs, this method calls `BtEndUpdatePrim` so that inherited classes may transfer their buffer variables to their associated properties. This method then transfers the buffer variables specific to this class.

When transferring buffer variables, the variables should be assigned to their properties instead of to the internal property variables. This allows the property write methods to perform any required processing.

The following example shows how this method may be used:

```
with TCPTransport do begin
  BeginUpdate;
  try
    ServerName := aServerName;
    RespondToBroadcasts := True;
    Mode := fftmListen;
    EndUpdate;
  except
    CancelUpdate;
  end;
```

See also: `BeginUpdate`, `BtBeginUpdatePrim`, `BtEndUpdatePrim`, `CancelUpdate`

```
function EstablishConnection(const aUserName : TffName;  
    aPasswordHash : integer; timeOut : longInt;  
    var aClientID : TffClientID) : TffResult; virtual; abstract;
```

↳ Establishes a connection with a server-side transport.

Use this method to establish a connection with a server-side transport. If you create your own transport, you must override this function. In addition to implementing the connection, you must also define the requirements for a transport to establish a connection (whether the transport must be in listen mode, for example).

aUserName is the logon ID of the person requesting a connection. This parameter may be an empty string if FlashFiler Server security is disabled.

aPasswordHash is the hash value of the password entered by the user logging on. If the FlashFiler Server's security is disabled, this parameter is ignored and may be set to zero. To generate the hash value for a password, use the FFCalcShStrElfHash function in the FFSRHASH unit.

timeOut is the number of milliseconds the transport may wait for a connection to be established with the server-side transport.

If the connection is established, this function returns DBIERR\_NONE and aClientID is filled with the unique client identifier handed back by the server. aClientID must be used in all subsequent calls to the Post and Request methods.

The following example illustrates the use of this method:

```
if TCPTransport.EstablishConnection(  
    'mephisto', FFCalcShStrElfHash(aPassword),  
    5000, newClientID)  
<> DBIERR_NONE then  
    ShowMessage('Connection failed.');
```

See also: ConnectionCount, OnAddClient, OnRemoveClient, TerminateConnection



```
property EventLogOptions : TffTransportLogOptions
TffTransportLogOptions = set of TffTransportLogOption;
TffTransportLogOption = (
    fftpLogErrors, fftpLogRequests, fftpLogReplies);
```

Default: [fftpLogErrors]

↳ Identifies the type of messages that may be written to the EventLog.

This property identifies what type of messages the transport may write to the EventLog. Messages are written to the log only if an EventLog is specified, EventLogEnabled is set to True, and the messages match one of the EventLogOptions shown in Table 9.3.

Value	Meaning
fftpLogErrors	Write all errors to the EventLog.
fftpLogRequests	Write all requests to the EventLog. When the transport is in listen mode, a request is defined as an incoming message. When the transport is in send mode, a request is defined as an outgoing message.
fftpLogReplies	Write all replies to the EventLog. When the transport is in listen mode, a reply is defined as an outgoing message. When the transport is in send mode, a reply is defined as an incoming message.

See also: Mode, TffLoggableComponent.EventLog,  
TffLoggableComponent.EventLogEnabled

## GetName

virtual abstract method

```
function GetName : string; virtual; abstract;
```

↳ Returns a descriptive name for the transport.

If you create your own transport, your implementation of this method should return a descriptive name for the transport. For example, the TffLegacyTransport class returns a name based upon the protocol being used. A TffLegacyTransport using TCP/IP protocol returns the value “TCP/IP (FF)”.

The name is displayed in the FlashFiler Server GUI and should tell the user what kind of transport or protocol is in use.

```
procedure GetServerNames(  
    aList : TStrings; const timeout : longInt); virtual; abstract;
```

↳ Retrieves the names of FlashFile Servers on the network or local machine.

If you create your own transport, you must implement this method. It must broadcast for FlashFile Servers listening via the same type of transport.

aList is an instance of TStrings. GetServerNames fills aList with the name of each server responding to the broadcast. timeout is the number of milliseconds the transport should wait for replies to the broadcast.

The following example shows how this method may be used:

```
Servers := TStringList.Create;  
TCPTransport.GetServerNames(Servers, 5000);  
ShowMessage('Servers responding: ' +  
    IntToStr(Servers.Count));
```

See also: RespondToBroadcasts

```
function IsConnected : Boolean; virtual; abstract;
```

↳ Identifies whether the transport is connected to a server.

If you create your own transport, this method must return True if your transport is connected to a server. Otherwise, it must return False.

See also: ConnectionCount, EstablishConnection, TerminateConnection

## Mode

property

```
property Mode : TffTransportMode  
TffTransportMode = (fftmSend, fftmListen);
```

Default: fftmSend

- Identifies whether the transport's main goal is to send requests to a server or handle requests from clients.

Each transport must be a sender (i.e., mode fftmSend) or a listener (i.e., mode fftmListen). Senders are used on a client application to send requests to a remote server. Listeners are used on a server to receive requests from a client.

The transport must have its State set to ffeInactive before you can change the Mode property.

See also: TffStateComponent.State

## MsgCount

read-only property

```
property MsgCount : longInt
```

- Identifies the number of requests processed by the transport.

Use this property to determine how many requests have been received by the transport. Be aware that MsgCount does not indicate the number of requests successfully processed by the command handler to which the requests are routed.

Use the ResetMsgCount method to reset this property to zero.

**Note:** This property is applicable only to transports in listen mode.

See also: CommandHandler, Mode, ResetMsgCount

```
property OnAddClient : TffAddClientEvent

TffAddClientEvent = procedure(Listener : TffBaseTransport;
    const userID : TffName; const timeout : longInt;
    const clientVersion : longInt;
    var passwordHash : TffWord32; var aClientID : TffClientID;
    var errorCode : TffResult; var isSecure : Boolean;
    var serverVersion : longInt) of object;
```

↳ Defines an event handler that is called when the transport has received a request to add a new client.

Before a transport in send mode can issue requests to a transport in listen mode, the sender asks the listener to establish a client connection. If the listener transport does not manage the creation of a client, it generates this event so that another object can create the necessary client information.

If you create your own transport and base it upon `TffBaseTransport` or `TffThreadedTransport`, you do not have to use this event. It is provided for your convenience. If you create your own transport and base it upon `TffLegacyTransport`, you must assign a handler to this event.

Listener is the transport receiving the request for a client connection.

userID is the user name parameter passed to the `EstablishConnection` method. the `OnAddClient` event handler may use this value to determine if the client has submitted a valid logon ID.

timeout is the number of milliseconds the client is willing to wait for the client connection to be established.

clientVersion is the version number of the client application. The `OnAddClient` event handler may compare this with its own version number to see if the client is compatible.

passwordHash is the hashed version of the user's password stored on the server. The `TffLegacyTransport` class uses this value to encrypt replies back to the client. As a result, once the user has specified their password then the hashed password never needs to be passed from client to server.

aClientID is the client ID assigned by the OnAddClient event handler to the new client connection. This client ID may be passed back to the sender transport. If your transport inherits from TffLegacyTransport, the sender transport expects the listener to return a unique client ID.

errorCode identifies whether the client was successfully added. If the client was successfully added, set this parameter to DBIERR\_NONE (in the FFSRBDE unit); otherwise, set it to an error code describing the problem.

isSecure must be set to True if secure communications are to be used.

serverVersion is the version number of the server application. This version number may be passed back to the client for a compatibility check.

See also: EstablishConnection, OnRemoveClient, TerminateConnection

## OnConnectionLost event

---

```
property OnConnectionLost : TffConnectionLostEvent
TffConnectionLostEvent = procedure(
    Sender : TffBaseTransport; aClientID : TffClientID) of object;
```

↪ Indicates when a transport has unexpectedly lost a connection.

9

If you are creating your own custom transports, raise this event when the transport loses a client connection. Do not raise the event if the client-side transport disconnected on purpose from the remote transport.

Sender is the transport losing the connection. aClientID is the unique ID assigned to the client via the EstablishConnection method.

See also: EstablishConnection

```
property OnRemoveClient : TffRemoveClientEvent  
  
TffRemoveClientEvent = procedure(  
    Listener : TffBaseTransport; const aClientID : TffClientID;  
    var errorCode : TffResult) of object;
```

↳ Defines an event handler that is called when a transport receives a termination request from a client transport.

This event is typically raised when the sender transport's `TerminateConnection` method is called. However, if a listener detects the connection is abruptly terminated, it may opt to generate this event.

If you create your own transport and base it upon `TffBaseTransport` or `TffThreadedTransport`, you do not have to use this event. It is provided for your convenience. If you create your own transport and base it upon `TffLegacyTransport`, you must assign a handler to this event.

Listener is the transport receiving the request for a client connection.

aClientID is the client ID assigned to the sender transport when the connection was first established. The handler for this event may use the client ID to index into its client information.

errorCode indicates whether the operation is successful. If the operation is successful, set this parameter to `DBIERR_NONE` (in the `FFSRBDE` unit); otherwise, set it to an error code describing the problem.

See also: `EstablishConnection`, `OnAddClient`, `TerminateConnection`

```
procedure Post(transportID : longInt; clientID : TffClientID;  
  msgID : longInt; requestData : pointer; requestDataLen : longInt;  
  timeout : longInt; replyMode : TffReplyModeType);  
  virtual; abstract;  
  
TffReplyModeType = (ffrmReplyExpected, ffrmNoReplyExpected,  
  ffrmNoReplyWaitUntilSent);
```

✚ Sends a request to a remote server.

In those cases where a reply is not expected, use this method to send a request to a remote server. If you create your own transport, decide upon the requirements that must be in place for this method to work. For example, does the transport have to be in send mode? Must the transport State be set to `ffesStarted`?

`transportID` is not currently used. Set it to zero. `clientID` is the unique client ID returned by the `EstablishConnection` method. `msgID` is a unique identifier for the type of message being sent.

`requestData` is a pointer to the data being sent to the server transport and is application specific. For example, if the client is inserting a new record into the database then `requestData` points to a buffer containing the new record. If no data is being sent to the remote server, set `requestData` to `nil`.

`requestDataLen` is the length of the data being sent to the remote server. If no data is being sent to the remote server, set `requestDataLen` to zero.

In the case where `replyMode` is set to `ffrmNoReplyWaitUntilSent`, `timeout` is the number of milliseconds the client is willing to wait for the reply to be sent; otherwise, set this parameter to zero.

replyMode identifies whether the transport expects a reply from the server. The values for replyMode are shown in the following table:

Value	Meaning
ffrmReplyExpected	The sender expects a reply from the server. Although you can specify this value for Post, it is treated as though you specified the value ffrmNoReplyExpected.
ffrmNoReplyExpected	The sender does not expect a reply from the server. The client thread calling Post resumes execution as soon as the message is queued for sending.
ffrmNoReplyWaitUntilSent	The sender does not expect a reply from the server. The client thread is blocked until the request has been sent or the number of milliseconds specified in the timeout parameter have elapsed.

The following code illustrates the use of Post:

```
var
  Buffer : TDateTime;
  ClientID : TffClientID

{Establish a connection with the server.}
if TCPTransport.EstablishConnection
  ('gandalf', 0, 5000, ClientID) = DBIERR_NONE then begin
  {Get the current time.}
  Buffer := Now;
  {Set the time on the server.}
  TCPTransport.Post
    (0, ClientID, msgSetServerTime, @Buffer,
     sizeof(Buffer), 1000, ffrmNoReplyExpected);
  TCPTransport.TerminateConnection(ClientID, 1000);
end;
```

See also: EstablishConnection, Mode, Process, Request, TffStateComponent.State



```

procedure Process(Msg : PffDataMessage); virtual;

PffDataMessage = ^TffDataMessage;

TffDataMessage = record
    dmMsg : longint;
    dmClientID : TffClientID;
    dmRequestID : longint;
    dmTime : TffWord32;
    dmRetryUntil : TffWord32;
    dmErrorCode : TffResult;
    dmData : pointer;
    dmDataLen : TffMemSize;

```

↳ Processes a request received from a client transport.

This method is called when the transport is in listen mode and it has received a message from a client transport. The default implementation of this method stores the transport into a thread-local variable (for use by the Reply method of the TffBaseTransport class) and passes the request off to CommandHandler.

Msg is the message received from the remote transport. The following table describes the possible values of the fields in TffDataMessage:

Field	Description
dmMsg	The unique identifier for the message. On the remote transport, this is specified via the Post or Reply methods.
dmClientID	The client ID assigned to the remote transport when it called the EstablishConnection method.
dmRequestID	An identifier assigned to the request. This identifier is unique to the client but may be duplicated across clients. This identifier must be passed back in the reply to the remote transport so that it knows with which request the reply is associated.
dmTime	The time the message was received. TffLegacyTransport fills in this value once the request has been received.
dmRetryUntil	The point in time up to which the server may try to fulfill the request. The transports provided with FlashFiler do not use this field. It is provided for your convenience.

Field	Description
dmErrorCode	When first received, this should be set to DBIERR_NONE. The reply back to the remote client should contain DBIERR_NONE if the request was fulfilled without error. Otherwise it must contain an error code describing the problem.
dmData	The data associated with the request. This data is specified when the remote transport's Post or Request method is called.
dmDataLen	The length of dmData. This value is specified when the remote transport's Post or Request method is called.

See also: CommandHandler, Mode, Reply

## Reply

virtual class method

```
class procedure Reply(msgID : longint; errorCode : TffResult;
    replyData : pointer; replyDataLen : LongInt); virtual;
```

↳ Sends a reply from a Listening transport back to the transport submitting a request.

This method is implemented as a class procedure so that the CommandHandler sending the reply does not have to know from which transport it received the request. Reply locates the transport using thread-local variable ffitvTransportID.

msgID is the message identifier passed by the request.

errorCode must be set to DBIERR\_NONE if the request was completed successfully. Otherwise it must be set to an error code describing the problem.

replyData is a pointer to data that is to be transmitted back to the client. For example, if the client requested a record be retrieved from the database then replyData would point to the buffer containing the record. If no data is to be sent back to the client, set replyData to nil.

replyDataLen is the length of the data pointed to by replyData. If no data is being sent to the client, set replyDataLen to zero.

See also: BtInternalReply, CommandHandler

```
procedure Request(transportID : longInt; clientID : TffClientID;  
  msgID : longInt; timeout : longInt; requestData : pointer;  
  requestDataLen : longInt; replyCallback : TffReplyCallback;  
  replyCookie : longInt); virtual; abstract;  
  
TffReplyCallback = procedure(msgID : longInt; errorCode :  
  TffResult; replyData : pointer; replyDataLen : LongInt;  
  replyCookie : LongInt);
```

↳ Sends a request to a remote server and waits for a reply.

In those cases where a reply is expected, use this method to send a request to a remote server. If you create your own transport, decide upon the requirements that must be in place for this method to work. For example, does the transport have to be in Send mode? Must the transport State be set to `ffesStarted`?

`transportID` is not currently used. Set it to zero.

`clientID` is the unique client ID returned by the `EstablishConnection` method.

`msgID` is a unique identifier for the type of message being sent.

`timeout` is the number of milliseconds the client is willing to wait for a reply from the remote server.

`requestData` is a pointer to the data being sent to the remote server and is application specific. For example, if the client is inserting a new record into the database then `requestData` points to a buffer containing the new record. If no data is being sent to the remote server, set `requestData` to `nil`.

`requestDataLen` is the length of the data being sent to the remote server. If no data is being sent to the remote server, set `requestDataLen` to zero.

`replyCallback` is the method called once the reply is received.

`replyCookie` is an integer value passed to `ReplyCallback`.

When implemented, this method blocks the current thread until a reply is received or the number of milliseconds specified by the timeout parameter has elapsed. As mentioned previously, `replyCallback` is called once the reply is received. The parameters passed to `replyCallback` are defined in the following table:

Parameter	Meaning
<code>msgID</code>	The unique identifier for the message sent to the remote server.
<code>errorCode</code>	The error code returned in the remote server's reply. If the request is carried out successfully then <code>errorCode</code> is set to <code>DBIERR_NONE</code> . Otherwise, it is set to an error code describing the problem.
<code>replyData</code>	A pointer to the data returned in the reply. If no data is sent in the reply, <code>replyData</code> is set to <code>nil</code> .
<code>replyDataLen</code>	The length of the data pointed to by <code>replyData</code> . If no data is sent in the reply, <code>replyDataLen</code> is set to zero.
<code>replyCookie</code>	The value specified in the call to <code>Request</code> .

The following code illustrates the use of `Request`:

```
var
  ClientID : TffClientID

{Establish a connection with the server.}
if TCPTransport.EstablishConnection
  ('gandalf', 0, 5000, ClientID) =
    DBIERR_NONE then begin
    {Get the time from the server.}
    TCPTransport.Request
      (0, ClientID, msgGetServerTime, nil, 0,
       1000, GetServerTimeCallback, 0);
    TCPTransport.TerminateConnection(ClientID, 1000);
  end;
```

See also: `EstablishConnection`, `Mode`, `Post`, `Process`, `Reply`, `TffStateComponent.State`

**ResetMsgCount**

virtual method

```
procedure ResetMsgCount; virtual;
```

↪ Resets the `MsgCount` property to zero.

Use this method to reset the transport's message count property to zero.

See also: `MsgCount`

## RespondToBroadcasts

property

```
property RespondToBroadcasts : Boolean
```

Default: False

- ↳ Indicates whether a Listener transport is to respond to broadcast requests from remote clients.

In general, there are two ways to connect a client transport with a remote server. The first method is to know the name (i.e., address) of the remote server. The second method is to use the `GetServerNames` method to find all available remote servers.

The `TffLegacyTransport` implementation of `GetServerNames` broadcasts a request across the local network. Any remote servers listening via the same protocol and having `RespondToBroadcasts` set to `True` send a response back to the broadcasting transport.

If `RespondToBroadcasts` is set to `False`, the transport does not respond to broadcast requests.

See also: `GetServerNames`

## ServerName

property

```
property ServerName : TffNetAddress
```

Default: Empty string

- ↳ Specifies the name of a remote server to which the transport is to connect.

In general, there are two ways to connect a client transport with a remote server. The first method is to know the name (i.e., address) of the remote server. The second method is to use the `GetServerNames` method to find all available remote servers.

If the name of the remote server is known, assign the name to the `ServerName` property. For example, a `TffLegacyTransport` using TCP/IP may have `ServerName` set to `prodServer@192.16.138.1`.

When the `ServerName` is known, the `EstablishConnection` method may use the `ServerName` to connect with the remote server.

See also: `EstablishConnection`, `GetServerNames`

## Sleep

virtual method

```
function Sleep(const timeOut : longInt) : Boolean; virtual;
```

↳ Forces a Sender transport to temporarily suspend its connection with the remote server.

The default implementation of this method returns False to indicate it is not implemented. If you create your own transport, use this method to have a Listener transport temporarily disconnect from the remote server. Restore the connection when any client-side activity occurs.

You may find this feature useful in situations where the user must disconnect their laptop from the network or the client application is in a period of inactivity.

See also: EstablishConnection, TerminateConnection

## Supported

virtual method

```
function Supported : Boolean; virtual;
```

↳ Indicates whether the operating system and drivers on the workstation support the transport.

The default implementation of this method returns True. If you are creating your own transport, this provides you with a way to check the workstation for compatibility with your transport. Return True if the transport is supported; otherwise, return False.

If the transport is not supported on the workstation, its State is set to ffesUnsupported.

See also: TffStateComponent.State

## TerminateConnection

virtual abstract method

```
procedure TerminateConnection(const aClientID : TffClientID;  
    const timeout : longInt); virtual; abstract;
```

↳ Tells a client transport to terminate its connection with the server transport.

When a client application has finished sending requests to a remote server, use this method to disconnect from the server.

aClientID is the client identifier returned by the EstablishConnection method.

timeout is the amount of time the client application is willing to wait for the disconnection to succeed.

See also: ConnectionCount, EstablishConnection

## UpdateCount

read-only property

property UpdateCount : Integer

↳ Identifies the number of times BeginUpdate has been called.

This property is for internal use only.

See also: BeginUpdate, CancelUpdate, EndUpdate

## Work

virtual abstract method

procedure Work; virtual; abstract;

↳ Tells the transport to perform work based upon its Mode.

If you create your own transport, a transport in Send mode must start processing requests and watching for replies. A transport in Listen mode must start listening for requests.

See also: Mode

---

## TffThreadedTransport Class

The `TffThreadedTransport` class provides multi-thread support for protocols lacking multi-thread support. If you are creating your own transport and your protocol of choice includes support for multiple threads, inherit from `TffBaseTransport` instead of this class.

This class creates one instance of `TffRequest` per request sent to the server transport. When the `Post` or `Request` method is called, this class creates an instance of `TffRequest` and pushes it onto an internal FIFO queue. If a reply is expected, the thread issuing the request waits for the reply to arrive. Classes inheriting from `TffThreadedTransport` must implement the processing of the internal queue and wake up the waiting thread once the reply is received.

An instance of `TffThreadPool` is connected to `TffThreadedTransport`. Classes inheriting from `TffThreadedTransport` may use the thread pool to process requests received by the server-side transport.



# Hierarchy

TComponent (VCL)

① TffComponent (FLLBASE)	78
② TffLoggableComponent (FLLCOMP)	80
③ TffStateComponent (FLLCOMP)	82
④ TffBaseTransport (FLLCOMM)	321
TffThreadedTransport (FLLCOMM)	

# Properties

④ CommandHandler	④ EventLogOptions	④ ServerName
④ Enabled	④ Mode	③ State
② EventLog	④ MsgCount	ThreadPool
② EventLogEnabled	④ RespondToBroadcasts	④ UpdateCount

# Methods

④ BeginUpdate	④ EstablishConnection	Request
④ BtBeginUpdatePrim	① FreeInstance	④ ResetMsgCount
④ BtCheckListener	④ GetName	③ ScInitialize
④ BtCheckSender	④ GetServerNames	③ ScPrepareForShutdown
④ BtCheckServerName	④ IsConnected	③ ScShutdown
④ BtEndUpdatePrim	② LcLog	③ ScStartup
④ BtInternalReply	① NewInstance	④ Sleep
④ CancelUpdate	Post	④ Supported
④ ConnectionCount	④ Process	④ TerminateConnection
④ EndUpdate	④ Reply	④ Work

# Events

④ OnAddClient	④ OnRemoveClient
④ OnConnectionLost	③ OnStateChange

## Reference Section

Post	method
------	--------

```
procedure Post(transportID : longInt; clientID : TffClientID;
  msgID : longInt; requestData : pointer; requestDataLen : longInt;
  timeout : longInt; replyMode : TffReplyModeType); override;

TffReplyModeType = (ffrmReplyExpected, ffrmNoReplyExpected,
  ffrmNoReplyWaitUntilSent);
```

🔗 Sends a request to a server transport.

In those cases where a reply is not expected, use this method to send a request to a remote server. TffThreadedTransport pushes the request onto an internal queue.

transportID is not currently used. Set it to zero.

clientID is the unique client ID returned by the EstablishConnection method.

msgID is a unique identifier for the type of message being sent.

requestData is a pointer to the data being sent to the remote server and is application specific. For example, if the client is inserting a new record into the database then requestData points to a buffer containing the new record. If no data is being sent to the remote server, set requestData to nil. The TffRequest object used to send the request makes a copy of this data.

requestDataLen is the length of the data being sent to the remote server. If no data is being sent to the remote server, set requestDataLen to zero.

In the case where replyMode is set to ffrmNoReplyWaitUntilSent, timeout is the number of milliseconds the client is willing to wait for the reply to be sent. Otherwise, set this parameter to zero.

replyMode identifies whether the transport expects a reply from the server. The following table defines the values for replyMode:

Value	Meaning
ffrmReplyExpected	The Sender expects a reply from the server. Although you can specify this value for Post, it is treated as though you specified the value ffrmNoReplyExpected.
ffrmNoReplyExpected	The Sender does not expect a reply from the server. The client thread calling Post resumes execution as soon as the message is queued for sending.
ffrmNoReplyWaitUntilSent	The Sender does not expect a reply from the server. The client thread is blocked until the request has been sent or timeout milliseconds have elapsed.

The following code illustrates the use of Post:

```
var
  Buffer : TDateTime;
  ClientID : TffClientID
{Establish a connection with the server.}
if TCPTransport.EstablishConnection
  ('gandalf', 0, 5000, ClientID) =
    DBIERR_NONE then begin
  {Get the current time.}
  Buffer := Now;
  {Set the time on the server.}
  TCPTransport.Post
    (0, ClientID, msgSetServerTime, @Buffer,
     sizeof(Buffer), 1000, ffrmNoReplyExpected);
  TCPTransport.TerminateConnection(ClientID, 1000);
end;
```

See also: Request

```
procedure Request(transportID : longInt; clientID : TffClientID;
  msgID : longInt; timeout : longInt; requestData : pointer;
  requestDataLen : longInt; replyCallback : TffReplyCallback;
  replyCookie : longInt); override;

TffReplyCallback = procedure(msgID : longInt;
  errorCode : TffResult; replyData : pointer;
  replyDataLen : LongInt; replyCookie : LongInt);
```

↳ Sends a request to a server transport and waits for a reply.

In those cases where a reply is expected, use this method to send a request to a server transport. TffThreadedTransport pushes the request onto an internal queue. The current thread is blocked until a reply is received or the number of milliseconds specified by the timeout parameter has elapsed.

transportID is not currently used. Set it to zero.

clientID is the unique client ID returned by the EstablishConnection method.

msgID is a unique identifier for the type of message being sent.

timeout is the number of milliseconds the client is willing to wait for a reply from the remote server.

requestData is a pointer to the data being sent to the remote server and is application specific. For example, if the client is inserting a new record into the database then requestData points to a buffer containing the new record. If no data is being sent to the remote server, set requestData to nil. The TffRequest object used to send the request makes a copy of this data.

requestDataLen is the length of the data being sent to the remote server. If no data is being sent to the remote server, set requestDataLen to zero.

replyCallback is the method called once the reply is received.

replyCookie is an integer value passed to ReplyCallback.

As mentioned previously, `replyCallback` is called once the reply is received. The parameters passed to `replyCallback` are as follows:

Parameter	Meaning
<code>msgID</code>	The unique identifier for the message sent to the remote server.
<code>errorCode</code>	The error code returned in the remote server's reply. If the request is carried out successfully then <code>errorCode</code> is set to <code>DBIERR_NONE</code> ; otherwise, it is set to an error code describing the problem.
<code>replyData</code>	A pointer to the data returned in the reply. If no data is sent in the reply, <code>replyData</code> is set to <code>nil</code> .
<code>replyDataLen</code>	The length of the data pointed to by <code>replyData</code> . If no data is sent in the reply, <code>replyDataLen</code> is set to zero.
<code>replyCookie</code>	The value specified in the call to <code>Request</code> .

The following code illustrates the use of `Request`:

```
var
  ClientID : TffClientID

{Establish a connection with the server.}
if TCPTransport.EstablishConnection
  ('gandalf', 0, 5000, ClientID) =
    DBIERR_NONE then begin
    {Get the time from the server.}
    TCPTransport.Request
      (0, ClientID, msgGetServerTime, nil, 0,
       1000, GetServerTimeCallback, 0);
    TCPTransport.TerminateConnection(ClientID, 1000);
  end;
```

See also: `EstablishConnection`, `Mode`, `Post`, `Process`, `Reply`, `TffStateComponent.State`

ThreadPool

property

```
property ThreadPool : TffThreadPool
```

↪ Indicates the thread pool to be used by a server transport when processing requests.

While `TffThreadedTransport` does not implement multi-threaded processing of requests, it does provide this property to facilitate processing. For an example of multi-threaded processing, see the `TffLegacyTransport` class (in the `FLLLCY` unit).

---

## TffLegacyTransport Class

The TffLegacyTransport class enables communication between client and server applications using TCP/IP, IPX/SPX, or Single User Protocol (SUP). The Protocol property controls which protocol the transport uses.

**Note:** FlashFiler no longer provides support for NetBIOS protocol.

TffLegacyTransport is thread-safe. When requests are submitted to a client-side transport via its Post or Request methods, the requests are placed into a FIFO queue. A protocol thread started by TffLegacyTransport removes each request from the queue and sends them to the server transport. If a reply is expected, the transport places the request on a wait list. When the reply is received, the transport places the reply data on the original request and wakes up the thread submitting the request.

When a request is received by a server-side TffLegacyTransport, the transport passes the request off to its associated command handler. If a thread pool is connected to the transport, the handoff is done using one of the pool's threads. Otherwise, it is done using a direct call to the command handler. The command handler or server engine is responsible for replying to the client.

The TffLegacyTransport is bidirectional. The primary job of a server-side transport is to receive requests from clients. A server application may also send a request back to a client using the Post and Request methods of a server-side TffLegacyTransport. You may configure the client-side TffLegacyTransport to receive such requests by attaching a command handler to the transport. See the Chat program in the Examples subdirectory for an example.

The TffLegacyTransport is capable of handling requests and replies too large to fit into one network packet.

# Hierarchy

## TComponent (VCL)

❶ TffComponent (FLLBASE).....	78
❷ TffLoggableComponent (FLLCOMP) .....	80
❸ TffStateComponent (FLLCOMP) .....	82
❹ TffBaseTransport (FLLCOMM) .....	321
❺ TffThreadedTransport (FLLCOMM) .....	345
TffLegacyTransport (FLLLCY)	

# Properties

❹ CommandHandler	❹ Mode	❸ State
❹ Enabled	❹ MsgCount	❺ ThreadPool
❷ EventLog	Protocol	❹ UpdateCount
❷ EventLogEnabled	❹ RespondToBroadcasts	
❹ EventLogOptions	❹ ServerName	

# Methods

❹ BeginUpdate	EstablishConnection	❺ Request
❹ BtBeginUpdatePrim	❶ FreeInstance	❹ ResetMsgCount
❹ BtCheckListener	GetName	❸ ScInitialize
❹ BtCheckSender	GetServerNames	❸ ScPrepareForShutdown
❹ BtCheckServerName	IsConnected	❸ ScShutdown
❹ BtEndUpdatePrim	❷ LcLog	❸ ScStartup
❹ BtInternalReply	❶ NewInstance	❹ Sleep
❹ CancelUpdate	❺ Post	❹ Supported
ConnectionCount	❹ Process	TerminateConnection
❹ EndUpdate	❹ Reply	Work

# Events

❹ OnAddClient	❹ OnRemoveClient
OnConnectionLost	❸ OnStateChange

## Reference Section

### ConnectionCount

method

```
function ConnectionCount : longInt; override;
```

↳ Returns the number of established connections.

For a transport in send mode, this returns the number of connections established with the server-side transport. For a transport in listen mode, this returns the number of connections established by client-side transports.

See also: EstablishConnection, TffBaseTransport.Mode

### EstablishConnection

method

```
function EstablishConnection(const aUserName : TffName;  
    aPasswordHash : integer; timeOut : longInt;  
    var aClientID : TffClientID) : TffResult; override;
```

↳ Establishes a connection with a server-side TffLegacyTransport transport.

Use this method to establish a connection with a server-side TffLegacyTransport transport.

aUserName is the logon ID of the person requesting a connection. This parameter may be an empty string if FlashFiler Server security is disabled.

aPasswordHash is the hash value of the password entered by the user logging on. If the FlashFiler Server's security is disabled, this parameter is ignored and may be set to zero. To generate the hash value for a password, use the FFCalcShStrElfHash function in the FFSRHASH unit.

timeOut is the number of milliseconds the transport to wait for a connection to be established.

If the connection is established, this function returns DBIERR\_NONE and aClientID is filled with the unique client identifier handed back by the server. aClientID must be used in all subsequent calls to the Post and Request methods.



When establishing a connection, the `TffLegacyTransport` does the following:

1. Starts a connection with the server transport using the `TffBaseCommsProtocol.Call` method of the transport's embedded protocol. If successful, the reply from the server includes a temporary `clientID`.
2. Asks the server for permission to attach as a client by sending an `ffnmAttachServer` (see the `FFNETMSG` unit) request. If the user name, password, and client version number are acceptable, the server returns the official `clientID` to the client transport. The client transport replaces the temporary `clientID` with the official `clientID`.
3. Sends a `ffnmCheckSecureComms` request to the server transport. The purpose of this message is to authenticate the client in a secure server situation. If the message is understood by the server transport, the identity of the client is assumed to be correct.

If the last step fails, this method returns error code `DBIERR_INVALIDUSRPASS`.

The following example illustrates the use of this method:

```
if TCPTransport.EstablishConnection(
    'mephisto', FFCalcShStrElfHash(aPassword),
    5000, newClientID) <> DBIERR_NONE then
    ShowMessage('Connection failed.');
```

See also: `ConnectionCount`, `TffBaseTransport.OnAddClient`,  
`TffBaseTransport.OnRemoveClient`, `TerminateConnection`

## **GetName**

**method**

```
function GetName : string; override;
```

↳ Returns a descriptive name for the transport.

`TffLegacyTransport` returns a name based upon the currently selected protocol. For example, if the `Protocol` property is set to `ptTCPIP` then this function returns “TCP/IP (FF)”.

See also: `Protocol`

```
procedure GetServerNames(  
    aList : TStrings; const timeout : longInt); override;
```

↪ Retrieves the names of FlashFiler Servers on the network or local machine.

aList is an instance of TStrings. GetServerNames fills aList with the name of each server responding to the broadcast. timeout is the number of milliseconds the transport should wait for replies to the broadcast.

A server-side TffLegacyTransport responds to this broadcast if the following prerequisites are met:

- The Protocol used by the server-side transport matches the protocol used by the client-side transport.
- The server-side transport's RespondToBroadcasts property is set to True.

Although the TffLegacyTransport's TCP/IP protocol can be used to connect client applications with a FlashFiler Server via the internet, this method will not broadcast to server-side transports across the internet. The TCP/IP protocol broadcasts to the local machine, local area networks, and wide area networks.

The following example shows how this method may be used:

```
Servers := TStringList.Create;  
TCPTransport.GetServerNames(Servers, 5000);  
ShowMessage('Servers responding: ' +  
    IntToStr(Servers.Count));
```

See also: TffBaseTransport.RespondToBroadcasts

```
function IsConnected : Boolean; override;
```

↳ Identifies whether the transport is connected to a server.

This function returns `True` if the transport's `State` is `ffesStarted` and the transport has established at least one connection to a server-side transport; otherwise, it returns `False`.

See also: `ConnectionCount`, `EstablishConnection`, `TerminateConnection`

```
property OnConnectionLost : TffConnectionLostEvent
```

```
TffConnectionLostEvent = procedure(  
    Sender : TffBaseTransport; aClientID : TffClientID) of object;
```

↳ Indicates when a transport has unexpectedly lost a connection.

The legacy transport raises this event when it detects it has lost a connection with the FlashFiler Server. It does not raise this event when it is explicitly told to terminate the connection with the FlashFiler Server via the `TerminateConnection` method.

Sender is the transport losing the connection. `aClientID` is the unique ID assigned to the client via the `EstablishConnection` method.

**Note:** This event is raised once per each lost connection if more than one connection has been established with the same FlashFiler Server.

See also: `EstablishConnection`, `TerminateConnection`

```
property Protocol : TffProtocolType

TffProtocolType = (
    ptSingleUser, ptTCPIP, ptIPXSPX, ptDirect, ptRegistry);

Default: ptRegistry
```

Identifies the protocol to be used by the transport.

The Protocol property tells the transport which protocol to use for sending or receiving requests. The following table explains each protocol value:

Value	Meaning
ptSingleUser	Single User Protocol (SUP). SUP uses Windows messaging to communicate between a client transport and a server transport on the same workstation. It may not be used to communicate across a network. SUP is faster than TCP/IP and IPX/SPX. It is useful for situations where a FlashFile Server must communicate with remote clients but must also communicate with a client on the same workstation.
ptTCPIP	TCP/IP using Winsock. This protocol may be used to communicate between transports on the same workstation or connected via Local Area Network (LAN), Wide Area Network (WAN), or Internet.
ptIPXSPX	IPX/SPX using Winsock.
ptRegistry	Indicates the protocol to be used is stored in the registry at HKEY_LOCAL_MACHINE\Software\TurboPower\FlashFile\2.0\Client Configuration, value Protocol. When the transport is initializing, it reads the value from the registry and instantiates the correct protocol.

**Note:** FlashFile no longer provides support for NetBIOS protocol.

```
procedure TerminateConnection(const aClientID : TffClientID;  
    const timeout : longInt); override;
```

↳ Tells a client transport to terminate its connection with the server transport.

When a client application has finished sending requests to a server transport, use this method to disconnect from the server transport.

aClientID is the client identifier returned by the EstablishConnection method. timeout is the amount of time the client application is willing to wait for the disconnection to succeed.

See also: ConnectionCount, EstablishConnection

```
procedure Work; override;
```

↳ Tells the transport to send and/or receive requests.

TffLegacyTransport is a bidirectional transport. Its behavior reflects its Mode property. However, a server TffLegacyTransport may be directed to send a request to a client TffLegacyTransport.

The implementation of the Work method reflects this capability. The Work method tells the embedded protocol to take time out to listen for requests from connected transport(s). The Work method then processes any unhandled request, if one is available.

The Work method is called repeatedly from the TffLegacyTransport's protocol thread, until the protocol thread is terminated.

See also: TffBaseTransport.Mode, TffStateComponent.State

# TffRequest Class

The TffThreadedTransport and TffLegacyTransport classes track requests using the TffRequest class. They create one instance of TffRequest per request. Each TffRequest sits in a FIFO queue waiting to be sent to the server transport.

If a reply is expected, the thread issuing the request calls the TffRequest.WaitForReply method. This method blocks the thread until a reply is received. Once the request is sent, the TffRequest object is shifted to a “waiting for reply” list. If no reply is expected then the TffRequest is freed and the thread issuing the request is not blocked. See the Post and Request methods of the TffThreadedTransport class on page 345 for additional information.

If a reply is expected and received, the protocol thread finds the TffRequest in the “waiting for reply” list and stores the reply data in the TffRequest. The protocol thread then uses the WakeUpThread method to unblock the thread associated with the TffRequest.

## Hierarchy

TObject (VCL)	
❶ TffObject (FLLBASE) .....	94
TffRequest (FLLREQ)	

## Properties

Aborted	MsgID	RequestData
BytesToGo	ReplyData	RequestDataLen
ClientID	ReplyDataLen	StartOffset
ErrorCode	ReplyMode	Timeout
EventLog	ReplyMsgID	

## Methods

AddToReply	Lock	Unlock
Create	❶ NewInstance	WaitForReply
❶ FreeInstance	SetReply	WakeUpThread

## Reference Section

### Aborted

read-only property

property Aborted : Boolean

↳ Indicates whether the request was aborted.

If Aborted is True, the request has been aborted. This typically happens when the client does not receive a reply from the server transport within the number of milliseconds specified by the Timeout property.

When a request is aborted, the protocol thread (i.e., the thread carrying out the actual sending and receiving of messages) is responsible for freeing the TffRequest instance. This is because the thread issuing the original request does not know if the protocol thread is accessing the TffRequest. It is entirely possible for the protocol thread to be placing the reply data onto the TffRequest when the timeout occurs. Freeing the TffRequest at that moment could cause an access violation.

See also: Timeout

### AddToReply

virtual method

```
procedure AddToReply(  
    replyData : pointer; replyDataLen : longInt); virtual;
```

↳ Appends reply data to the TffRequest's current set of reply data.

If a reply is too large to fit into one packet of information, the protocol thread receives multiple packets of information from the server transport. When the protocol thread receives the first packet of information, it uses the SetReply method to place the reply data onto the TffRequest. When subsequent packets of information are received, the protocol thread appends the additional reply data onto the TffRequest using the AddToReply method.

replyData is a pointer to the data received from the server transport. replyDataLen is the length of the data pointed to by replyData.

See also: SetReply

```
property BytesToGo : longInt
```

Default: 0

↳ Specifies how many bytes of RequestData remain to be sent to the server transport.

When sending a request to the server transport, the protocol thread reads the request data from the TffRequest object. If the request data is too large to fit into one packet of information, the protocol thread only retrieves as many bytes of request data as will fit into a packet. In this situation, the protocol thread uses the BytesToGo property to track how many bytes remain to be sent to the server transport.

See also: RequestData, RequestDataLen, StartOffset

```
property ClientID : TffClientID
```

↳ Specifies the client with which the request is associated.

Each TffRequest is associated with a specific client connection. The ClientID is the unique identifier for that client connection, as returned by the EstablishConnection method of TffBaseTransport.

See also: TffBaseTransport.EstablishConnection

```
constructor Create(clientID : TffClientID; msgID : longint;  
    requestData : pointer; requestDataLen : LongInt; timeout :  
    longInt; const replyMode : TffReplyModeType); virtual;
```

```
TffReplyModeType = (ffrmReplyExpected, ffrmNoReplyExpected,  
    ffrmNoReplyWaitUntilSent);
```

↳ Creates an instance of TffRequest.

The TffThreadedTransport class creates one instance of TffRequest per request sent via its Post and Request methods.



`clientID` is the unique identifier for the client as returned by the `EstablishConnection` method of `TffBaseTransport`. `msgID` is the unique identifier for the type of message being sent to the server transport.

`requestData` is a pointer to the data being sent to the server transport. `TffRequest` makes a copy of this data and is responsible for freeing that copy. If no data is being sent then set this parameter to `nil`.

`requestDataLen` is the length of the data being sent to the server transport. If no data is being sent then set this parameter to zero.

`timeout` is the number of milliseconds the requesting thread is willing to wait for a response from the server transport. If a reply is expected but not received within the specified number of milliseconds then the `TffRequest`'s `Aborted` property is set to `True`, its `ErrorCode` is set to `fferrReplyTimeout`, and the thread associated with the request is unblocked via the `WakeUpThread` method.

`replyMode` indicates whether the thread issuing the request expects a reply. The following table defines the possible values for this parameter:

Value	Meaning
<code>ffrmReplyExpected</code>	The sender expects a reply from the server.
<code>ffrmNoReplyExpected</code>	The sender does not expect a reply from the server. The client thread calling <code>Post</code> resumes execution as soon as the message is queued for sending.
<code>ffrmNoReplyWaitUntilSent</code>	The sender does not expect a reply from the server. The client thread is blocked until the request has been sent or timeout milliseconds have elapsed.

See also: `Aborted`, `ClientID`, `TffBaseTransport.EstablishConnection`, `MsgID`, `TffThreadedTransport.Post`, `ReplyMode`, `TffThreadedTransport.Request`, `RequestData`, `RequestDataLen`, `Timeout`, `WakeUpThread`

## ErrorCode

property

property ErrorCode : TffResult

↳ Indicates whether the request was successfully processed.

The ErrorCode property contains the error code returned in the reply from the server transport. If the request timed out before a reply was received, ErrorCode is set to `fferrReplyTimeout`. If the request was carried out successfully, ErrorCode is set to `DBIERR_NONE`.

See also: `TffBaseTransport.Reply`, `Timeout`

## EventLog

property

property EventLog : TffBaseLog

↳ Specifies the TffBaseLog to which debug and error information may be written.

If logging is enabled, the transport instantiating the request assigns the event log component to the EventLog property. TffRequest contains a protected method, `rqWriteString`, for writing information to the log. The current implementation of TffRequest does not write any debug or error messages to the log.

```
procedure Lock;
```

↳ Ensures that only one thread may access the `TffRequest` until the `Unlock` method is called.

In certain situations, only one thread should access a `TffRequest` instance. The thread calling `Lock` is granted access. Any subsequent threads calling `Lock` are blocked until the first thread calls the `Unlock` method.

For example, it is entirely possible for `TffLegacyTransport` to receive a reply to a request before the method issuing the request has finished. In that situation, it is possible for the `TffRequest` to be freed before the sending method finishes using `TffRequest`, resulting in an access violation. The solution is to call `Lock` in `TffRequest`'s destructor and to call `Lock` in the `TffLegacyTransport` send method. This ensures the request is destroyed after the sending routine has completed.

**Note:** Each call to `Lock` must have a corresponding call to `Unlock`. The best practice is to put the call to `Unlock` in the finally section of a try/finally block, as shown in the following example:

```
aRequest.Lock;
try
  aRequest.AddToReply(@Buffer, length(Buffer));
finally
  aRequest.Unlock;
end;
```

See also: `Unlock`

```
property MsgID : longInt
```

↳ Specifies the type of message being sent to the server transport.

`MsgID` is a unique identifier for the kind of request sent to the server transport. For example, a `GetServerDateTime` request may have `MsgID` set to 100 while a `SetServerDateTime` request may have `MsgID` set to 101. The request's `MsgID` is specified in the `Create` constructor.

See also: `Create`

## ReplyData

property

property ReplyData : pointer

Default: nil

↪ Specifies the data received from the server transport.

If no reply has been received, a timeout has occurred, or the reply does not contain any data, this property is set to nil. Otherwise, ReplyData points to the set of data sent by the server transport.

**Note:** TffRequest is responsible for freeing the memory associated with ReplyData.

See also: AddToReply, ReplyDataLen, SetReply

## ReplyDataLen

property

property ReplyDataLen : longInt

Default: 0

↪ Specifies the length of the data received from the server transport.

If no reply has been received, a timeout has occurred, or the reply does not contain any data, this property is set to zero. Otherwise, ReplyDataLen specifies the length of the data pointed to by ReplyData.

See also: ReplyData

```
property ReplyMode : TffReplyModeType
```

```
TffReplyModeType = (ffrmReplyExpected, ffrmNoReplyExpected,  
    ffrmNoReplyWaitUntilSent);
```

↳ Specifies whether the client expects a reply from the server transport.

The following table defines the possible values for replyMode:

Value	Meaning
ffrmReplyExpected	The client thread expects a reply from the server. The client thread submitting the request is blocked until a timeout occurs or a reply has been received from the server transport.
ffrmNoReplyExpected	The client thread does not expect a reply from the server transport. The client thread resumes execution as soon as the TffRequest is queued for sending.
ffrmNoReplyWaitUntilSent	The client thread does not expect a reply from the server. The client thread is blocked until the request has been sent or timeout milliseconds have elapsed.

See also: Create, TffBaseTransport.Post, TffBaseTransport.Request

```
property ReplyMsgID : longInt
```

↳ Specifies the message identifier returned in the reply.

In general, ReplyMsgID is the same as MsgID. However, in some situations the TffLegacyTransport may send a multi-part reply to the client transport. In that situation, ReplyMsgID is set to ffrmMultiPartMessage (in the FFNETMSG unit). This tells the TffLegacyTransport it must break the reply into its individual messages.

## **RequestData**

**read-only property**

property RequestData : pointer

↳ Specifies the data to be sent to the server transport.

This property points to the data associated with the request. If the type of message being sent does not require any data, set this property to nil.

**Note:** TffRequest is responsible for freeing the data associated with RequestData.

See also: Create, RequestDataLen

## **RequestDataLen**

**read-only property**

property RequestDataLen : longInt

↳ Specifies the length of the request data to be sent to the server transport.

RequestDataLen specifies the length, in bytes, of RequestData. If no data is associated with the request, this property returns zero.

See also: Create, RequestData

```
procedure SetReply(replyMsgID : longInt; errorCode : TffResult;  
    replyData : pointer; totalReplyLen : longInt;  
    replyDataLen : longInt); virtual;
```

↳ Sets the reply information on a TffRequest.

When a reply is received from the server transport, the protocol thread finds the TffRequest associated with the reply and uses SetReply to place the reply information on the TffRequest.

replyMsgID is the message identifier associated with the request. This is typically the same as the message identifier in the original request. However, TffLegacyTransport recognizes ffnmMultiPartMessage (in the FFNETMSG unit) as a multi-part reply.

errorCode indicates whether the request was successfully completed on the server. If errorCode is DBIERR\_NONE (in the FFSRBDE unit) then the request was successfully completed. Otherwise, the errorCode indicates the type of problem that occurred.

replyData is the data associated with the reply. If no data is required or available for the reply, this parameter is set to nil. If the reply is too large to fit into one packet, this points to a subset of the reply data.

totalReplyLen is the total length of the reply. If the reply is too large to fit into one packet, this parameter identifies the total length of the reply data and will be greater than the replyDataLen parameter.

replyDataLen is the length of the data pointed to by replyData. If the reply is too large to fit into one packet, this parameter will be smaller than the totalReplyLen parameter.

If the reply is too large to fit into one packet, the protocol thread calls SetReply when the first part of the reply is received. When receiving subsequent packets, the protocol thread appends the data to the TffRequest using AddToReply.

See also: AddToReply

property StartOffset : longInt

Default: 0

- ↪ Specifies the starting position within RequestData for the next packet being sent to the server transport.

When sending a request to the server transport, the protocol thread reads the request data from the TffRequest object. If the request data is too large to fit into one packet of information, the protocol thread only retrieves as many bytes of request data as will fit into a packet. In this situation, the protocol thread uses the StartOffset property to track its position within RequestData.

See also: BytesToGo, RequestData, RequestDataLen

property Timeout : longInt

- ↪ Indicates the number of milliseconds to wait for a response from the server transport.

When a reply is expected, this property identifies the number of milliseconds the client thread will wait for a response from the server transport. If no reply is received in that time, the TffRequest is aborted (its Aborted property is set to True and its ErrorCode is set to fferrReplyTimeout).

When no reply is expected but the client thread does not want to resume execution until the request has been sent, this property identifies the number of milliseconds the client thread will wait for the request to be sent. If the request is not sent in that time, the TffRequest is aborted.

See also: Aborted, Create, ErrorCode



```
procedure Unlock;
```

↳ Terminates exclusive access to an instance of `TffRequest`.

A thread obtains exclusive access to an instance of `TffRequest` using the `Lock` method. When the thread has finished accessing the `TffRequest`, it must call `Unlock`. Calling `Unlock` allows other threads calling `Lock` to gain exclusive access to the `TffRequest`.

The following example illustrates the use of `Unlock`:

```
aRequest.Lock;
try
  aRequest.AddToReply(@Buffer, length(Buffer));
finally
  aRequest.Unlock;
end;
```

See also: `Lock`

## WaitForReply

virtual method

```
procedure WaitForReply(const timeout : TffWord32); virtual;
```

9

↳ Blocks a thread until a reply is received from the server.

`WaitForReply` is called in the following two situations:

- A client thread calls `TffThreadedTransport.Post` and sets `ReplyMode` to `ffrmNoReplyWaitUntilSent`.
- A client thread calls `TffThreadedTransport.Request`.

`WaitForReply` causes the client thread to be blocked until it is unblocked using the `WakeUpThread` method.

`timeout` is the number of milliseconds the client thread will wait in a blocked state. If no reply is received or the request has not been sent in the allotted time, the `Aborted` property is set to `True` and the client thread is unblocked.

See also: `Aborted`, `TffThreadedTransport.Post`, `TffThreadedTransport.Request`, `ReplyMode`, `WakeUpThread`

```
procedure WakeUpThread; virtual;
```

↳ Unblocks a thread that called WaitForReply.

WakeUpThread is called when a reply has been received from the server transport. It unblocks the client thread calling WaitForReply so that the client may process the reply.

See also: WaitForReply

# TffThreadPool Class

Transports inheriting from TffThreadedTransport prefer to have requests processed in a separate thread. The TffThreadPool class provides these transports with a quick way to obtain and reuse threads. The use of TffThreadPool is not limited to transports. You can take advantage of the thread pool from your own classes as you see fit.

Use the thread pool's InitialCount property to specify how many threads the pool instantiates when the pool is first created. Use the pool's MaxCount property to specify the maximum number of threads the pool may create.

Use the pool's Process method to obtain a thread and have the thread perform some specific task. The thread pool tracks which of its threads are active (i.e., processing a request) and which threads are available for work. If all threads are active and the thread pool has reached its MaxCount, the request is placed into a FIFO queue. The next available thread processes the next pending request from the queue. If no pending requests are available, the thread returns itself to the pool's inactive list.

The threads created by TffThreadPool are of TffPooledThread class. The thread pool does not free already created threads. If you wish to free unused threads, use the Flush method.

## Hierarchy

TComponent (VCL)	
❶ TffComponent (FLLBASE).....	78
TffThreadPool (FLLTHRD)	

## Properties

ActiveCount	InactiveCount	MaxCount
FreeCount	InitialCount	

## Methods

Flush	❶ NewInstance
❶ FreeInstance	ProcessThreaded

## Reference Section

### ActiveCount

read-only property

```
property ActiveCount : integer
```

↳ Identifies the number of threads currently performing a task.

The ActiveCount increases when the thread pool assigns a thread to handle a task submitted via the ProcessThreaded method. The ActiveCount decreases when a thread finishes a task and is returned to the pool of inactive threads.

**Note:** The volatility of this property varies with the frequency with which the ProcessThreaded method is called and with the rate at which the tasks submitted to ProcessThreaded are completed.

See also: InactiveCount, ProcessThreaded

### Flush

method

```
procedure Flush(NumToRetain : integer);
```

↳ Forces the thread pool to remove all but NumToRetain inactive threads.

The thread pool does not free inactive threads. If you wish to reduce the number of inactive threads in the pool, use this method. NumToRetain is the number of inactive threads to keep in the inactive pool. This method does not affect active threads.

See also: ActiveCount, InactiveCount

### FreeCount

read-only property

```
property FreeCount : integer
```

↳ Identifies the number of unfilled thread slots in the pool.

FreeCount is the additional number of threads the pool may create before it reaches its MaxCount. It is calculated as  $\text{MaxCount} - (\text{ActiveCount} + \text{InactiveCount})$ .

See also: ActiveCount, InactiveCount, MaxCount

## InactiveCount

read-only property

```
property InactiveCount : integer
```

↳ Identifies the number of threads that are available to carry out tasks.

The `InactiveCount` value increases when a thread finishes a task and is returned to the pool of inactive threads. It decreases when the thread pool assigns a thread to handle a task submitted via the `ProcessThreaded` method.

**Note:** The volatility of this property varies with the frequency with which the `ProcessThreaded` method is called and with the rate at which the tasks submitted to `ProcessThreaded` are completed.

See also: `ActiveCount`, `ProcessThreaded`

## InitialCount

property

```
property InitialCount : integer
```

Default: 5

↳ Defines the initial number of threads to be created by the thread pool.

Use this property to specify how many threads the pool should instantiate when the pool is first created. This is a time saving feature. The cost of creating the threads is spent up-front instead of when you need the threads.

See also: `MaxCount`

## MaxCount

property

```
property MaxCount : integer
```

Default: 16

↳ Defines the maximum number of threads that may be created by the thread pool.

Use this property to limit the number of threads that may be created by the thread pool. If you decrease the value of this property and the current number of threads in the pool exceed the new value, the thread pool attempts to reduce the number of inactive threads to meet the new threshold.

See also: `InitialCount`

```
procedure ProcessThreaded(aProcessEvent : TffThreadProcessEvent;  
    aProcessCookie: longInt);  
  
TffThreadProcessEvent = procedure(  
    const aProcessCookie : longInt) of object;
```

↳ Acquires a thread from the pool and causes the thread to call the specified event.

Use this method to run a task in a thread provided by the pool. If the pool has at least one inactive thread, the pool assigns the task to the thread and the thread executes the task immediately. If the pool does not have any inactive threads and the pool has not reached its MaxCount, the pool creates a new thread, assigns the task to the thread, and the thread executes the task immediately. If the pool does not have any inactive threads and it has reached its MaxCount, the task is placed into a FIFO queue. When a thread becomes available, it pulls the next task off the queue for immediate processing.

aProcessEvent is the procedure to be called by the thread. aProcessCookie is passed to aProcessEvent. aProcessCookie can be whatever you want it to be. For example, TffLegacyTransport passes a pointer to the message data that is to be processed by the transport's command handler.

See also: ActiveCount, InactiveCount, MaxCount

---

## TffThread Class

The TffThread class overrides the memory allocation performed by TThread and serves as an ancestor to all FlashFiler thread classes. Due to issues with VCL's heap manager, FlashFiler Server uses its own memory management scheme to allocate memory for components. This provides greater reliability for database applications requiring uninterrupted availability.

### Hierarchy

TThread (VCL)

    TffThread (FFLLBASE)

### Methods

FreeInstance

NewInstance

## Reference Section

### FreeInstance

method

```
procedure FreeInstance; override;
```

↪ Deallocates memory allocated by a previous call to the NewInstance method.

This method deallocates memory obtained for an instance of this type. Memory deallocated by this method is returned to the FlashFiler memory pool from which it was obtained. You should never need to call FreeInstance directly.

### NewInstance

method

```
class function NewInstance : TObject; override;
```

↪ Allocates memory for each instance of TffThread.

This method obtains memory from a FlashFiler memory pool for a new instance of this type. The memory is freed by the FreeInstance method. You should never need to call NewInstance directly.



---

# TffPooledThread Class

The TffThreadPool component uses instances of TffPooledThread to process requests. TffPooledThreads are not created in a suspended state. Instead, the thread is started immediately and the thread's Execute method waits for one of two internal events to be signaled.

If the thread pool calls the thread's Process method, the Process method sets internal variables and signals the thread's internal work event. The Execute method then calls the procedure specified in the Process method. When the procedure finishes, the Execute method returns the thread to the pool's inactive list.

If the thread pool needs to terminate the thread, the thread pool calls the thread's DieDieDie method. This method signals the thread's internal termination event, which ends the Execute method.

## Hierarchy

TThread (VCL)	
❶ TffThread (FFLLBASE).....	376
TffPooledThread (FFLLTHRD)	

## Methods

DieDieDie	❶ NewInstance
❶ FreeInstance	Process

## Reference Section

### DieDieDie

method

```
procedure DieDieDie;
```

↳ Tells the thread to terminate.

Pooled threads are kept alive until their parent thread pool is freed or their parent thread pool is told to flush inactive threads. A pooled thread will not terminate until its internal termination event is signaled. Use this method to terminate the thread and signal its internal termination event.

See also: `TffThreadPool.Flush`

### Process

method

```
procedure Process(aProcessEvent : TffThreadProcessEvent;  
  aProcessCookie: longInt);
```

```
TffThreadProcessEvent = procedure(  
  const aProcessCookie: longInt) of object;
```

↳ Tells the pooled thread to call the specified event.

The `Process` method is called from `ProcessThreaded` method of the `TffThreadPool` class. `Process` assigns the parameters to internal variables and then signals its work event. The signaling of the work event causes the thread's `Execute` method to call the procedure specified by `aProcessCookie`. Once the procedure finishes, the thread returns itself to the thread pool's inactive list.

`aProcessEvent` is the procedure to be called by the thread. `aProcessCookie` is passed to the procedure represented by `aProcessEvent`. `aProcessCookie` can be whatever you want it to be. For example, `TffLegacyTransport` passes a pointer to the message data that is to be processed by the transport's command handler.

See also: `TffThreadPool.ProcessThreaded`

---

# TffTimerThread Class

Use TffTimerThread to periodically invoke a procedure. For example, FlashFiler Server uses this class to perform garbage collection.

## Hierarchy

TThread (VCL)	
❶ TffThread (FFLLBASE).....	376
TffTimerThread (FFLLTHRD)	

## Properties

Frequency

## Methods

Create	❶ FreeInstance
DieDieDie	❶ NewInstance

## Reference Section

### Create

### virtual constructor

```
constructor Create(const aFrequency : DWord;  
    aTimerEvent : TffThreadProcessEvent;  
    const aTimerEventCookie : longInt;  
    const createSuspended : Boolean); virtual;
```

```
TffThreadProcessEvent = procedure(  
    const aProcessCookie: longInt) of object;
```

↳ Creates an instance of TffTimerThread.

The Create method specifies the procedure that is to be called and the frequency with which it is called. When started, TffTimerThread waits for the number of milliseconds set on aFrequency before calling aTimerEvent.

aFrequency is the number of milliseconds between each invocation of aTimerEvent. aTimerEvent is the procedure called by the thread. aTimerEventCookie is passed to aTimerEvent. aTimerCookie can be whatever you want it to be. For example, it could be a pointer to a data structure required by aTimerEvent.

createSuspended indicates whether the timer thread is to be started immediately after creation. If createSuspended is False, the thread is started immediately. If createSuspended is True, the thread is placed into a suspended state. To start the thread, call the Resume method of the TThread class.

See also: TThread.Create, TThread.Resume

### DieDieDie

### method

```
procedure DieDieDie;
```

↳ Tells the thread to terminate.

When you no longer need an instance of TffTimerThread, use this method to terminate the thread. This method calls TThread.Terminate and signals the thread's internal termination event. The thread's Execute method exits as soon as the termination event is signaled.

property Frequency : DWord

↳ Specifies the number of milliseconds between each invocation of the timer event.

Set the initial frequency via the Create constructor. Use this property to read the current frequency or to change the frequency. If you change the frequency, the change does not take affect until after the next time the timer procedure is called.

See also: Create

---

# Chapter 10: Command Handlers

This chapter is for developers using the `TffServerCommandHandler` component and for developers interested in creating their own command handlers. The sole purpose of a command handler is to route requests from a transport to a server engine, plugin command handler, or engine manager.

The `TffBaseCommandHandler` base class defines the foundational requirements of all command handlers. If you are creating your own command handler and the command handler does not need to interface with a standard FlashFiler engine, use `TffBaseCommandHandler` as a starting point. If your command handler needs to be aware of standard FlashFiler engines, start with `TffIntermediateCommandHandler` or `TffServerCommandHandler` instead.

## Message routing

When a transport receives a request, it passes the request to the command handler's `Process` method. The `Process` method implements the routing of the request. For example, FlashFiler's standard command handler, `TffServerCommandHandler`, uses the `Dispatch` method of the `TObject` class to route the request to a message handler declared in `TffServerCommandHandler`'s protected section.

**Note:** A command handler's `Process` method is responsible for freeing the memory associated with the message. `TffBaseCommandHandler.BchFreeMsg` may be used for this purpose.

In one sense, a command handler provides a message-oriented interface to a server engine, plugin command handler, or engine manager. The engines do not understand the content of a request; they merely provide a procedural interface to the outside world. The command handler translates the request into a method call of the server engine. As a result, the command handler must also make sure a reply is routed back to the remote transport from which the request originated. The `TffServerCommandHandler`'s protected message handlers accomplish this using `TffBaseTransport.Reply`.

This is best illustrated with an example. The `TransactionStart` method of the `TffServerEngine` component in the `FFSRENG` unit is defined as follows:

```
function TransactionStart
    (const aDatabaseID : TffDatabaseID;
     const aFailSafe   : boolean : TffResult;
```

When a client application starts a transaction, it sends the `ffnmStartTransaction` message (declared in the `FFNETMSG` unit) to the remote server engine. The data structure associated with `ffnmStartTransaction` is `TffnmStartTransactionReq` as shown in the following lines of code:

```
TffnmStartTransaction = packed record;  
    DatabaseID : TffDatabaseID;  
    FailSafe   : Boolean;  
end;
```

The job of the command handler is to map the fields in `TffnmStartTransaction` to the `TransactionStart` function of the `TffServerEngine` component and to send a reply containing the result of `TransactionStart`. The message handler `nmStartTransaction` of the `TffServerCommandHandler` class in the `FFSRCMD` unit accomplishes these tasks as shown in the following code sample:

```
procedure TffServerCommandHandler.nmStartTransaction  
    (var Msg : TffDataMessage);  
var  
    Error : TffResult;  
begin  
    with Msg, PffnmStartTransactionReq(dmData)^ do begin  
        Error := FServerEngine.TransactionStart(DatabaseID, FailSafe);  
        TffBaseTransport.Reply  
            (ffnmStartTransaction, Error, nil, 0);  
    end;  
end;
```

## Command handler states

Like transports and server engines, command handlers contain a state engine. A transport may be in one of several states as shown in the Figure 10.1.

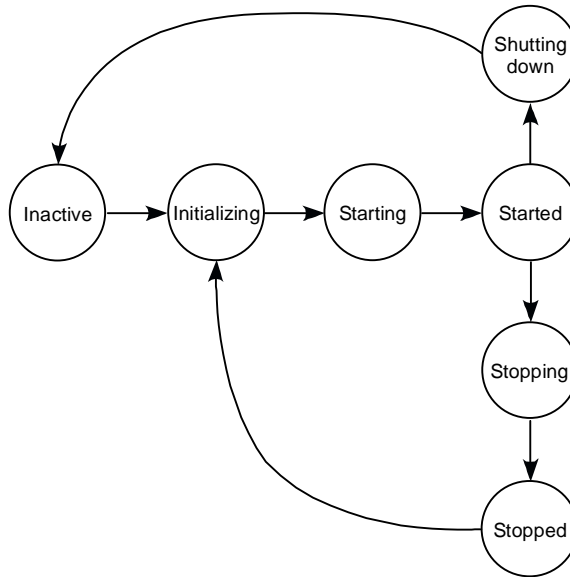


Figure 10.1: Server state diagram.

The circles represent states. Lines between the circles represent the path taken by the command handler's internal state engine as it goes from a start state to a destination state. For example, when a command handler is first created its State is set to `ffesInactive`. Setting the State to `ffesStarted` causes the state engine to change the state from `ffesInactive` to `ffesInitializing`, then to `ffesStarting`, and finally to `ffesStarted`.

As a command handler progresses from one state to the next, the state engine calls protected methods within the command handler. These methods allow a command handler to perform initializations, startup actions, and shut down actions. See the State property of the `TffStateComponent` class for more information.

**Note:** The standard FlashFile command handlers route messages regardless of their state. The reason for this is because there is nothing in their internal makeup that requires the state to be set a certain way.



**Note:** When a transport's `CommandHandler` property is set, the transport registers itself with the command handler. When a command handler's state changes, it sets the state of the registered transports. See “Define the command handler” on page 295 for additional information. Table 10.1 describes each state.

**Table 10.1:** *Command handler states.*

State	Meaning
<code>ffesInactive</code>	The command handler is not doing any work. This is the default state for all command handlers.
<code>ffesInitializing</code>	The command handler is being prepared for work.
<code>ffesStarting</code>	The command handler is starting.
<code>ffesStarted</code>	The command handler has successfully started and is ready for work.
<code>ffesShuttingDown</code>	The command handler has been told to shut down.
<code>ffesStopping</code>	This state does not apply to command handlers.
<code>ffesStopped</code>	This state does not apply to command handlers.
<code>ffesUnsupported</code>	This state does not apply to command handlers.
<code>ffesFailed</code>	This state does not apply to command handlers.

---

# TffBaseCommandHandler Class

The TffBaseCommandHandler class defines the basic interface for a FlashFiler command handler. A command handler routes messages from a transport to a server engine or plugin engine. Also, a command handler may choose to handle certain messages independent of an engine.

If you need to create a command handler unrelated to the standard FlashFiler Server engine, inherit from this class. Otherwise, base your command handler upon TffIntermediateCommandHandler or TffServerCommandHandler.

## Hierarchy

TComponent (VCL)	
❶ TffComponent (FFLLBASE) .....	78
❷ TffLoggableComponent (FFLLCOMP) .....	80
❸ TffStateComponent (FFLLCOMP) .....	82
TffBaseCommandHandler (FFLLCOMM)	

## Properties

EngineManager	❷ EventLogEnabled	TransportCount
❷ EventLog	❸ State	Transports

## Methods

BchFreeMsg	❶ NewInstance	❸ ScPrepareForShutdown
❶ FreeInstance	Process	❸ ScShutdown
❷ LcLog	❸ ScInitialize	❸ ScStartup

## Events

❸ OnStateChange
-----------------

## Reference Section

### BchFreeMsg

virtual method

```
procedure BchFreeMsg(Msg : PffDataMessage);  
  
PffDataMessage = ^TffDataMessage;  
  
TffDataMessage = Record  
    dmMsg : longint;  
    dmClientID : TffClientID;  
    dmRequestID : longint;  
    dmTime : TffWord32;  
    dmRetryUntil : TffWord32;  
    dmErrorCode : TffResult;  
    dmData : pointer;  
    dmDataLen : TffMemSize;  
end;
```

↳ Frees the TffDataMessage structure and message content.

Each command handler is responsible for freeing the memory associated with a message sent to its Process method. The reason for this is that in a multi-threaded situation, the transport has started a new thread to process the request. The transport cannot free the message because it has moved on and is unaware of what the command handler is doing with the message.

This method is provided as a default way to free the message.

See also: Process

---

```
property EngineManager : TffBaseEngineManager
```

↳ Specifies the engine manager to which the command handler routes messages.

When a typical command handler receives a request, it first tries to route it to a server engine. If the server engine does not recognize the request, the command handler passes it on to any plugin command handlers registered with the command handler. If no plugin command handlers accept the request then the command handler passes the request off to the engine manager as a last resort.

The engine manager accepts requests for starting and stopping the server engines and plugin engines. These messages are used infrequently, which explains why the engine manager is the last in line to receive a request. If you expect such a request to flow through a particular transport then you should connect that transport's command handler to the engine manager.

If no engine manager is specified, the command handler sends back a reply containing the error code `fferrUnknownMsg`.

See also: [Process](#)

```
procedure Process(Msg : PffDataMessage); virtual;  
  
PffDataMessage = ^TffDataMessage;  
  
TffDataMessage = record  
    dmMsg : longint;  
    dmClientID : TffClientID;  
    dmRequestID : longint;  
    dmTime : TffWord32;  
    dmRetryUntil : TffWord32;  
    dmErrorCode : TffResult;  
    dmData : pointer;  
    dmDataLen : TffMemSize;
```

↳ Processes a request received from a transport.

When a listening transport receives a request from a client transport, the transport routes the message to a command handler, as specified by the transport's `CommandHandler` property. The default implementation of this method determines if any associated `TffBasePluginCommandHandlers` can handle the request. If none of the `TffBasePluginCommandHandlers` handle the request and a `TffBaseEngineManager` is associated with the command handler, the message is passed off to the `TffBaseEngineManager`. If the `TffBaseEngineManager` does not handle the request, a reply is sent with error code `fferrUnknownMsg`.

Msg is the message received by the transport. The following table describes the possible values of the fields in Msg:

Field	Description
dmMsg	The unique identifier for the message.
dmClientID	The client ID assigned to the remote transport when it called the EstablishConnection method.
dmRequestID	An identifier assigned to the request. This identifier is unique to the client but may be duplicated across clients. This identifier must be passed back in the reply to the remote transport so that it knows with which request the reply is associated.
dmTime	The time the message was received. TffLegacyTransport fills in this value once the request has been received.
dmRetryUntil	The point in time up to which the server may try to fulfill the request. The transports provided with FlashFile do not use this field. It is provided for your convenience.
dmErrorCode	When first received, this should be set to DBIERR_NONE. The reply back to the remote client should contain DBIERR_NONE if the request was fulfilled without error. Otherwise it must contain an error code describing the problem.
dmData	The data associated with the request. This data is specified when the remote transport's Post or Request method is called.
dmDataLen	The length of dmData. This value is specified when the remote transport's Post or Request method is called.

**Note:** If you override this method, your Process method must free the memory associated with the message. One way to do this is to use the BchFreeMsg method.

See also: BchFreeMsg

## TransportCount

read-only property

```
property TransportCount : Longint
```

↳ Identifies the number of transports passing requests to the command handler.

Transports are connected to a command handler via the Transport property of the CommandHandler class. This property tells you how many transports are connected to the command handler. Use this property to determine the upper limit of the Transports property.

See also: TffBaseTransport.CommandHandler, Transports

## Transports

read-only property

```
property Transports[aInx : longInt] : TffBaseTransport
```

↳ Provides access to the transports connected to the command handler.

Transports are connected to a command handler via the Transport.CommandHandler property. Use this property to access the transports. aInx is an index into the list of connected transports and is based zero. Use the TransportCount property to find the upper bound of Transports as shown in the following example:

```
with MyCmdHandler do
  for anIndex := 0 to pred(TransportCount) do
    ShowMessage(Transports[anIndex].GetName);
```

See also: TffBaseTransport.CommandHandler, TransportCount

---

# TffIntermediateCommandHandler Class

The TffIntermediateCommandHandler class exists not because it has any special functionality, but because the classes TffServerCommandHandler and TffServerEngine need to be aware of each other. For example, if command handler A references server engine B, then command handler A must be notified when server engine B is removed from the design-time form.

TffServerCommandHandler and TffServerEngine are declared in separate units. This means they cannot access protected methods normally used to notify components of design-time actions. Our solution was to declare the TffIntermediateServerEngine and TffIntermediateCommandHandler classes in the FFSRINTM unit.

## Hierarchy

TComponent (VCL)	
❶ TffComponent (FLLBASE) .....	78
❷ TffLoggableComponent (FLLCOMP) .....	80
❸ TffStateComponent (FLLCOMP) .....	82
❹ TffBaseCommandHandler (FLLCOMM) .....	387
TffIntermediateCommandHandler (FFSRINTM)	

## Properties

❷ EventLog	ServerEngine	❹ TransportCount
❷ EventLogEnabled	❸ State	❹ Transports

## Methods

❹ BchCheckInactive	❶ NewInstance	❸ ScPrepareForShutdown
❶ FreeInstance	❹ Process	❸ ScShutdown
❷ LcLog	❸ ScInitialize	❸ ScStartup

## Events

❸ OnStateChange
-----------------



## Reference Section

**ServerEngine**

**property**

---

```
property ServerEngine : TffIntermediateServerEngine
```

↳ Specifies the server engine to which this command handler routes client requests.

Note that the server engine must be of the `TffIntermediateServerEngine` class, not `TffBaseServerEngine`.

See also: `TffIntermediateServerEngine`

---

## TffServerCommandHandler Class

The TffServerCommandHandler class routes client requests from an instance of TffBaseTransport to an instance of TffIntermediateEngine. The messages recognized by TffServerCommandHandler are listed in the FFSRNETMSG unit.

**Note:** When a transport is connected to an instance of TffServerCommandHandler, the TffServerCommandHandler replaces any existing handlers for the transport's OnAddClient and OnRemoveClient events with its own. See the schOnAddClient and schOnRemoveClient methods for additional information.

### Message routing

The TffServerCommandHandler class contains one protected method for each message constant declared in the FFNETMSG unit. The protected methods are declared as message handlers and their names are prefixed with “nm”. Each protected message handler corresponds to a public method of TffBaseServerEngine. For example, nmCursorSetToBegin, nmRecordInsert, and nmRecordGetNext correspond to CursorSetToBegin, RecordInsert, and RecordGetNext in TffBaseServerEngine, respectively.

When a TffServerCommandHandler receives a request via its Process method, it treats the request as a message and uses the Dispatch method of the TObject class to route the request to the appropriate message handler. The message handler then performs any logging, if necessary, and calls the corresponding method in the TffBaseServerEngine. The command handler then sends a reply back to the remote transport using TffBaseTransport.Reply.

If TffServerCommandHandler does not have a message handler for the request, the request is routed to the TffBaseCommandHandler.Process method. This method then passes the request on to the plugin command handlers and/or engine manager.

**Note:** A command handler's Process method is responsible for freeing the memory associated with the message. Use the BchFreeMsg method of the TffBaseCommandHandler class for this purpose.

# Hierarchy

TComponent (VCL)

- ❶ TffComponent (FLLBASE) ..... 78
  - ❷ TffLoggableComponent (FLLCOMP) ..... 80
    - ❸ TffStateComponent (FLLCOMP) ..... 82
      - ❹ TffBaseCommandHandler (FLLCOMM)..... 387
        - ❺ TffIntermediateCommandHandler (FFSRINTM) .... 393

TffServerCommandHandler

# Properties

❷ EventLog	❺ ServerEngine	❹ TransportCount
❷ EventLogEnabled	❸ State	❹ Transports

# Methods

❹ BchCheckInactive	❶ NewInstance	❸ ScShutdown
DefaultHandler	Process	❸ ScStartup
❶ FreeInstance	❸ ScInitialize	
❷ LcLog	❸ ScPrepareForShutdown	

# Events

❸ OnStateChange
-----------------

## Reference Section

### DefaultHandler

procedure

```
procedure DefaultHandler(var Message); override;
```

- ↳ Routes an unhandled message to the Process method of the TffBaseCommandHandler class.

As documented in the VCL help file for TObject.Dispatch, this method is called when a message handler is not found for the dispatched Message. This method sends the request to the Process method of the TffBaseCommandHandler class which then passes the request to the plugin command handlers and/or the engine manager.

See also: Process

### Process

procedure

```
procedure Process(Msg : PffDataMessage); override;
```

- ↳ Routes a request to a message handler.

The protected section of TffServerCommandHandler contains one message handler for each message defined in the FFNETMSG unit. The Process method used the Dispatch method of the TObject class to route the message to the appropriate message handler. After the message has been dispatched, it's memory is freed using BchFreeMsg.

If a message handler is not found for the request, then the message is sent to the DefaultHandler method.

**Note:** If you create your own command handler, your Process method must free the memory associated with each message. The BchFreeMsg method of the TffBaseCommandHandler class is provided for this purpose.

See also: DefaultHandler, TffBaseCommandHandler.BchFreeMsg



---

# Chapter 11: Server Engines

This chapter is for developers interested in the server engine components, SQL engine components, and engine manager data module.

A server engine encapsulates the core database functionality used by FlashFiler clients. It can perform such tasks as retrieve records, navigate through records, insert new records, build indexes, and restructure a table. FlashFiler provides an abstract engine class, `TffBaseServerEngine`, which defines the basic interface of a database engine.

As described in Chapter 10, “Command Handlers”, a command handler translates requests from remote clients into something a server engine understands. Because a `TffBaseServerEngine` doesn’t know anything about command handlers, the `TffIntermediateServerEngine` class was created to allow command handlers to connect to a server engine. By separating the classes this way, the FlashFiler architecture gives you more flexibility. You can use command handlers apart from server engines and vice versa.

The `TffServerEngine` class implements the true FlashFiler database engine. The FlashFiler Server and FlashFiler Service, as delivered, each contain one instance of `TffServerEngine`. FlashFiler Server uses an engine manager (see the `TffBaseEngineManager` and the `TffEngineManager` classes in this chapter) containing three transports and a command handler to route all commands to the server engine. As described in Chapter 3, you could just as easily place a `TffServerEngine` in your own application or place multiple server engines within the FlashFiler Server.

`TffServerEngine` does not contain any functionality to process SQL statements. Instead, it routes all SQL-related requests to a `TffSQLEngine` component. By default, FlashFiler Server’s engine manager contains a `TffSQLEngine` component. If you do not want to use SQL in your product, you may remove this component from the engine manager. If you want to create your own FlashFiler compatible SQL engine, use the `TffBaseSQLEngine` class as a starting point.

To have your application connect to a remote server engine, you can explicitly place an instance of `TffRemoteServerEngine` on your form or you can rely upon FlashFiler Client to automatically create an instance behind the scenes. `TffRemoteServerEngine` implements the same interface as `TffServerEngine`. However, the remote server engine uses a transport to send the FlashFiler Client requests to the real server engine. The remote server engine then translates the replies back into an understandable form for the client-side components. See “Following the Flow” on page 270 for more details.

Because `TffServerEngine` and `TffRemoteServerEngine` share the same interface, you can switch your application between an embedded server engine and a remote server engine just by changing component connections. No other code changes are necessary.

You could even embed a `TffServerEngine` and a `TffRemoteServerEngine` in the same application. The `TffServerEngine` within the application could support remote clients and the application could connect to a totally different FlashFiler Server through the `TffRemoteServerEngine`.

If you need to modify products to fit your needs, FlashFiler's open architecture gives you a large measure of flexibility and many options for customization.

---

# TffBaseServerEngine Class

The TffBaseServerEngine class is an abstract class defining the standard interface for a server engine. This class doesn't do much in and of itself. Its descendant classes must implement the server engine functionality.

## Hierarchy

TComponent (VCL)

❶ TffComponent (FLLBASE) .....	78
❷ TffLoggableComponent (FLLCOMP) .....	80
❸ TffStateComponent (FLLCOMP) .....	82
TffBaseServerEngine (FLENG)	

## Properties

❷ EventLog	IsReadOnly	❸ State
❷ EventLogEnabled	NoAutoSaveCfg	

## Methods

❶ FreeInstance	❷ LcLog	❸ ScPrepareForShutdown
GetInterestedMonitors	❶ NewInstance	❸ ScShutdown
GetServerNames	❸ ScInitialize	❸ ScStartup

## Events

❸ OnStateChange
-----------------



## Reference Section

### GetInterestedMonitors

method

```
function GetInterestedMonitors
  (const anObjectClass : TffServerObjectClass) : TffList;

TffServerObjectClass = class of TffObject;
```

↪ Retrieves a list of TffBaseEngineMonitors interested in the specified server object class.

Use this method to determine what engine monitors have registered interest in a specific server object class. See Chapter 12, “Event monitoring” on page 441, for more information concerning engine monitors and a list of classes that may be specified for parameter `anObjectClass`.

If no monitors have registered interested in `anObjectClass`, this method returns `nil`. Otherwise, this method returns a `TffList` containing one or more instances of `TffIntListItem`. A `TffIntListItem` contains a reference to an engine monitor; it is not the engine monitor. To get a handle on the engine monitor you must cast the `TffIntListItem`’s key to type `TffBaseEngineMonitor`, as shown in the following example:

```
MonitorList := anEngine.GetInterestedMonitors(TffSrBaseCursor);
if assigned(MonitorList) then begin
  for anIndex := 0 to pred(MonitorList.Count) do begin
    aMonitor := TffBaseEngineMonitor(
      TffIntListItem(MonitorList[anIndex]).KeyAsInt);
    anExtender := aMonitor.Interested(Self);
    if assigned(anExtender) then
      bcAddExtender(anExtender);
  end;
  MonitorList.Free;
end;
```

Your code is responsible for freeing the returned `TffList`. The instances of `TffIntListItem` within the list are freed but the monitors are untouched.

```
procedure GetServerNames(  
    aList : TStrings; aTimeout : Longint); virtual; abstract;
```

↳ Returns the list of remote servers available through this server engine.

This method applies only to representations of remote server engines such as `TffRemoteServerEngine`. This method should be used to tell FlashFiler Client what servers are available through the transport connected to the server engine. `aList` is an instance of `TStringList`. If any servers are found then `aList` is populated with the server names and addresses (e.g., `elmer@192.68.104.3` for a TCP/IP transport). Otherwise, the list remains empty.

`aTimeout` is the number of milliseconds to wait for servers to respond.

```
property IsReadOnly : Boolean
```

↳ Indicates whether the server engine may write to its log and configuration information.

The value of the `IsReadOnly` property is typically set via a configuration file. In some situations, it is desirable to have the server provide read-only access to information. For example, a CD-ROM based application may provide access to a read-only database. The server engine should not allow data to be written, should not write information to a log file, and should not save configuration information.

This property returns `True` if the server engine is in read-only mode otherwise it returns `False`.

This property overrides the `NoAutoSaveCfg` property.

See also: `NoAutoSaveCfg`

---

property NoAutoSaveCfg : Boolean

↳ Indicates whether the server engine may write its configuration information to disk.

The value of this property is typically set via a configuration file. By default, a server engine usually writes its current configuration information (e.g., aliases, users, network configuration) to disk when the information changes and/or when the server engine shuts down. In some situations, it may be best to disable this behavior. For example, an application developer may deliver his product with pre-defined configuration files and does not want the end user to change the configuration files in any manner.

If this property returns True, the server engine is not saving its configuration to disk. If this property returns False, the server engine is saving configuration changes to disk.

See also: IsReadOnly

# TffIntermediateServerEngine Class

The TffIntermediateServerEngine class exists because the TffServerCommandHandler and TffServerEngine components need to be aware of each other. For example, if command handler A is connected to server engine B then server engine B must be notified when command handler A is removed from the design-time form.

TffServerCommandHandler and TffServerEngine are declared in separate units. This means they cannot access protected methods normally used to notify components of design-time actions. Our solution was to declare the TffIntermediateServerEngine and TffIntermediateCommandHandler classes in the FFSRINTM unit.

## Hierarchy

TComponent (VCL)

❶ TffComponent (FFLLBASE) .....	78
❷ TffLoggableComponent (FFLLCOMP) .....	80
❸ TffStateComponent (FFLLCOMP) .....	82
❹ TffBaseServerEngine (FFLENG) .....	401
TffIntermediateServerEngine (FFRINTM)	

## Properties

CmdHandler	❷ EventLogEnabled	❸ State
CmdHandlerCount	❹ IsReadOnly	
❷ EventLog	❹ NoAutoSaveCfg	

## Methods

❶ FreeInstance	❷ LcLog	❸ ScPrepareForShutdown
❹ GetInterestedMonitors	❶ NewInstance	❸ ScShutdown
❹ GetServerNames	❸ ScInitialize	❸ ScStartup

## Events

❸ OnStateChange
-----------------

## Reference Section

### **CmdHandler**

**read-only property**

---

```
property CmdHandler[aInx : Longint] : TffIntermediateCommandHandler
```

↳ Provides access to the command handlers connected to this server engine.

When you connect a command handler to a server engine, the command handler registers itself with the server engine. The server engine maintains an internal list of the registered command handlers. Use the `CmdHandlerCount` property to determine the number of registered command handlers.

Use this property to access the command handlers. `aInx` is an index into the list of command handlers and is zero-based.

See also: `CmdHandlerCount`

### **CmdHandlerCount**

**read-only property**

---

```
property CmdHandlerCount : Longint
```

↳ Returns the number of command handlers connected to the server engine.

Use this property to determine the number of command handlers connected to the server engine. Use the `CmdHandler` property to access the command handlers.

See also: `CmdHandler`



# TffServerEngine Component

The TffServerEngine component implements the FlashFiler database engine. It contains all functionality related to clients, sessions, databases, tables, and cursors. TffServerEngine does not contain any SQL-specific knowledge. Instead, it forwards SQL requests to the TffBaseSQLEngine identified by its SQLEngine property.

The FlashFiler Server contains one instance of this component. You may place additional instances of this component in FlashFiler Server's engine manager. If you need to implement a single-user application or directly connect to a server engine, you may also place an instance of this component directly into your application. Connecting FlashFiler Client components directly to a server engine gives you much faster performance than talking to a server engine through a network protocol.

## Hierarchy

TComponent (VCL)

❶ TffComponent (FFLLBASE)	78
❷ TffLoggableComponent (FFLLCOMP)	80
❸ TffStateComponent (FFLLCOMP)	82
❹ TffBaseServerEngine (FFLLENG)	401
TffServerEngine (FFSRENG)	

## Properties

BufferManager	❷ EventLogEnabled	SQLEngine
ConfigDir	❹ IsReadOnly	❸ State
Configuration	❹ NoAutoSaveCfg	
❷ EventLog	ServerName	

## Methods

❶ FreeInstance	❷ LcLog	❸ ScPrepareForShutdown
❹ GetInterestedMonitors	❶ NewInstance	❸ ScShutdown
GetServerNames	❸ ScInitialize	❸ ScStartup

## Events

❸ OnStateChange
-----------------

## Reference Section

### BufferManager

read-only property

---

```
property BufferManager : TffBufferManager
```

- ↳ Returns the buffer manager that handles the caching of tables accessed by this server engine.

The most time-consuming operation performed by the server engine is reading from and writing to the hard drive. To optimize performance, the server engine uses a buffer manager to cache file blocks in memory. The file blocks accessed most frequently are kept in memory, greatly decreasing the number of disk reads. File blocks are written to disk only when they have been modified by a transaction and the transaction has committed.

### ConfigDir

property

---

```
property ConfigDir : TffPath
```

Default: The application directory

- ↳ Specifies the directory in which the FlashFiler Server stores its server tables.

The FlashFiler Server stores configuration, alias, and user information in what is referred to as “server tables”. By default, it saves the tables to the current directory of the application executable. If you incorporate more than one server engine into the FlashFiler Server or your own application, use this property to provide each server engine with its own configuration directory. Otherwise, each server engine reads the same set of configuration, alias, and user information.

11

### Configuration

read-only property

---

```
property Configuration : TffServerConfiguration
```

- ↳ Provides access to the server engine’s configuration.

The server engine maintains an internal set of information, including lists of aliases, users, and user-defined indexes. Configuration information is read from configuration files when the server engine starts. Information is written back to the configuration files when the information changes. When the server shuts down, it automatically writes the in-memory alias and user lists to disk.

```
procedure GetServerNames(  
    aList : TStrings; aTimeout : Longint); override;
```

↳ Populates a TStrings descendent with server names.

This method applies only to representations of remote server engines such as TffRemoteServerEngine. Because TffBaseServerEngine declares this method as an abstract method, TffServerEngine must implement the method. Any application calling this method has most likely connected its TffClient component directly to this server engine. Therefore, this method returns the constant string “Direct” which is intended to mean the application is directly calling the server engine.

See also: TffBaseServerEngine.GetServerNames

```
property ServerName : TffNetName  
    TffNetName = string[ffcl_NetNameSize];
```

↳ Returns the server’s name.

In addition to its network address, a server is typically assigned a name. The name, coupled with the address, identifies the server to the outside world. For example, if a client application broadcasts for available servers via TCP/IP, it may see “production@10.0.0.1” and “test@10.0.0.2”. The server’s name is stored in the FFSINFO server table along with the other general information for the server.

```
property SQLEngine : TffBaseSQLEngine
```

↳ Specifies the SQL engine to which this server engine forwards SQL requests.

Use this property to determine which SQL engine handles SQL requests sent to this server engine. Replies from the SQL engine are routed back through the originating server engine.





## TffRemoteServerEngine Component

The TffRemoteServerEngine component represents a server engine hosted on another computer or within another application on the same computer. When coupled with a transport component, you can use TffRemoteServerEngine to connect to a remote server engine.

TffRemoteServerEngine and TffServerEngine both inherit from TffBaseServerEngine, meaning that both components have the same interface. While TffServerEngine implements the real database engine functionality, TffRemoteServerEngine implements a connection to a server engine. It translates the FlashFiler Client's calls to the server engine interface into requests sent via a transport. When it receives a reply from the remote server engine, it translates the reply back into something the FlashFiler Client understands.

Because TffRemoteServerEngine and TffServerEngine both share the same interface, you may easily switch your application between using a remote server engine and an embedded server engine.

### Hierarchy

TComponent (VCL)

- ❶ TffComponent (FFLLBASE) ..... 78
  - ❷ TffLoggableComponent (FFLLCOMP) ..... 80
    - ❸ TffStateComponent (FFLLCOMP) ..... 82
      - ❹ TffBaseServerEngine (FFLENG) ..... 401

TffRemoteServerEngineComponent

# Properties

- ② EventLog
- ② EventLogEnabled
- ④ IsReadOnly
- ④ NoAutoSaveCfg
- ③ State

# Methods

- ① FreeInstance
- ④ GetInterestedMonitors
- GetServerNames
- ② LcLog
- ① NewInstance
- ③ ScInitialize
- ③ ScPrepareForShutdown
- ③ ScShutdown
- ③ ScStartup  
Transport

# Events

- ③ OnStateChange

## Reference Section

### GetServerNames

method

```
procedure GetServerNames(  
    aList : TStringList; aTimeout : Longint); override;
```

↳ Returns the list of remote servers available through this server engine.

The TffClient component uses this method to retrieve the list of servers that may be reached through the remote server engine's transport. For example, if the remote server engine is connected to a TCP/IP transport then this function returns all FlashFiler Servers responding to TCP/IP broadcasts.

aList is a TStringList or similar. If any servers are found then aList is populated with the server names and addresses (e.g., elmer@192.68.104.3 for a TCP/IP transport). Otherwise, the list remains empty.

aTimeout is the number of milliseconds to wait for servers to respond.

### Transport

property

```
property Transport : TffBaseTransport
```

↳ Specifies the transport through which this server engine routes requests.

When you place a TffRemoteServerEngine on your form, you must connect the remote server engine to a transport component. The remote server engine uses the transport to connect with a FlashFiler Server on another computer or within another application on the current computer. Once a connection is established, all requests and replies flow through the transport.

---

## TffBaseSQLEngine Class

TffBaseSQLEngine is an abstract class describing the basic interface for a FlashFiler SQL engine. If you want to create a FlashFiler-compatible SQL engine, use this class as a starting point.

All SQL-related requests are routed through a TffServerEngine. Therefore, a given SQL engine should be capable of supporting any number of server engines. In general, a FlashFiler Client issues the following pattern of calls to a SQL engine in the pattern defined in Table 11.1.

**Table 11.1:** *SQL call patterns.*

Method Called	Action Taken
Alloc	SQL engine allocates resources for a new SQL statement.
Prepare	SQL engine parses and optimizes the SQL statement.
SetParams	Only issued by parameterized queries. Provides the value of each parameter within the SQL statement to the SQL engine. The SQL engine stores these values for execution.
Exec	SQL engine executes the SQL statement.
FreeStmt	SQL engine frees the resources allocated to the SQL statement.

When the ExecDirect method of the TffQuery component is implemented, a FlashFiler Client may issue a call to ExecDirect followed by FreeStmt.

**Note:** SQL statements within FlashFiler are restricted to a specific database. They may operate upon any tables within a specific database. However, any given SQL statement may not cross database boundaries.

# Hierarchy

TComponent (VCL)

- ❶ TffComponent (FFLLBASE) ..... 78
  - ❷ TffLoggableComponent (FFLLCOMP) ..... 80
- TffBaseSQLEngine (FFSQLBAS)

# Properties

- ❷ EventLog
- ❷ EventLogEnabled

# Methods

- |            |                |               |
|------------|----------------|---------------|
| Alloc      | ❶ FreeInstance | ❶ NewInstance |
| Exec       | FreeStmt       | Prepare       |
| ExecDirect | ❷ LcLog        | SetParams     |

## Reference Section

**Alloc**

**virtual abstract method**

```
function Alloc(  
    anEngine : TffBaseServerEngine; aClientID : TffClientID;  
    aDatabaseID : TffDatabaseID;  
    var aStmtID : TffSqlStmtID): TffResult; virtual; abstract;
```

↪ Allocates resources for a new SQL statement.

The implementation of this method must allocate the required data structures for a new SQL statement. It must generate a unique ID for the statement and return it in parameter aStmtID.

anEngine is the server engine forwarding the request. aClientID is the unique ID of the client submitting the request. aDatabaseID is the unique ID of the database upon which the query will operate.

If the SQL statement is allocated successfully, this method must return DBIERR\_NONE (declared in the FFSRBDE unit) otherwise it must return an error code applicable to the situation. The server engine forwarding the request to the SQL engine is responsible for handling the error code.

Because the SQL engine may be receiving SQL requests from multiple server engines, the SQL engine should keep track of the requests on a per engine, per client basis.

See also: FreeStmt

```
function Exec(
    aStmtID: TffSqlStmtID; var aOpenMode: TffOpenMode;
    var aCursorID: TffCursorID; aStream: TStream): TffResult;
    virtual; abstract;
```

```
TffOpenMode = (omReadOnly, omReadWrite);
```

↳ Executes an allocated, prepared SQL statement.

The implementation of this method must execute a SQL statement that has been allocated and prepared via the Alloc and Prepare methods. aStmtID is the unique ID of the SQL statement as returned by the Alloc method. aOpenMode indicates whether the query is intended to be read-only or read/write. FlashFiler Client uses the output value of aOpenMode to determine whether the result set is live. This method must set aOpenMode to a value based upon the following criteria:

Input Value	Query May Be Live	Output Value
omReadOnly	No	omReadOnly
omReadOnly	Yes	omReadOnly
omReadWrite	No	omReadOnly
omReadWrite	Yes	omReadWrite

It is up to the SQL engine to determine whether the query is live or read-only.

If the SQL statement is executed successfully, the implementation of this method must open a cursor for the statement's result set. The FlashFiler Client uses the cursor to retrieve and possibly update the records in the result set.

If the SQL statement is executed successfully, the following must happen:

- The unique ID of the cursor must be returned in parameter aCursorID.
- The TffDataDictionary describing the structure of the result set must be written to aStream.
- This method must return DBIERR\_NONE (declared in the FFSRBDE unit).

If the SQL statement is not executed successfully, this method must return an error code applicable to the situation. It must write the length of the associated error message to aStream followed by the error message itself. The server engine forwarding the request to the SQL engine is responsible for handling the error.

See also: Alloc, ExecDirect, Prepare

```
function ExecDirect(
    anEngine : TffBaseServerEngine; aClientID : TffClientID;
    aDatabaseID : TffDatabaseID; aQueryText : String;
    var aOpenMode : TffOpenMode; var aCursorID : TffCursorID;
    aStream : TStream): TffResult; virtual; abstract;

TffOpenMode = (omReadOnly, omReadWrite);
```

↳ Allocates, prepares, and executes a SQL statement.

The implementation of this method must allocate resources for the SQL statement, prepare the statement for execution, and execute the statement.

**anEngine** is the server engine forwarding the request to the SQL engine. **aClientID** is the unique ID of the client submitting the request. **aDatabaseID** is the unique ID of the database on which the query will operate.

**aQueryText** is the SQL statement to be executed. **aOpenMode** indicates whether the statement is intended to have a read-only or read/write result set. FlashFiler Client uses the output value of **aOpenMode** to determine whether the result set is live. This method must set **aOpenMode** to a value based upon the following criteria:

Input Value	Query May Be Live	Output Value
omReadOnly	No	omReadOnly
omReadOnly	Yes	omReadOnly
omReadWrite	No	omReadOnly
omReadWrite	Yes	omReadWrite

It is up to the SQL engine to determine whether the query is live or read-only.

If the statement is executed successfully and returns a result set, this method must set **aCursorID** to the unique ID of the cursor opened for the result set. This method must then write the instance of **TffDataDictionary** describing the result set structure to **aStream**. If the statement is executed successfully but does not return a result, **aCursorID** must be set to zero and **aStream** must remain empty.

If the statement is executed successfully then this method must return **DBIERR\_NONE** (declared in the **FFSRBDE** unit) otherwise it must return an error code applicable to the situation. It must write the length of the associated error message to **aStream** followed by the error message itself. The server engine forwarding the request to the SQL engine is responsible for handling the error.

See also: **Alloc**, **Prepare**, **Exec**, **FreeStmt**



```
function FreeStmt(  
    aStmtID: TffSqlStmtID): TffResult; virtual; abstract;
```

↳ Frees the resources associated with a SQL statement.

This method is called when the FlashFiler Client has finished using a SQL statement. The implementation of this method must free the resources associated with the SQL statement. If a cursor was opened for the result set, the resources should remain allocated until the cursor is closed. aStmtID is the unique ID of the statement as returned by the Alloc method.

See also: Alloc

```
function Prepare(  
    aStmtID: TffSqlStmtID; aQueryText: String;  
    aStream : TStream): TffResult; virtual; abstract;
```

↳ Prepares a SQL statement for execution.

One way to improve application performance is to have the SQL engine parse and optimize frequently used SQL statements. The work may be done once instead of each time the query is executed. The implementation of this method should parse and optimize the SQL statement.

aStmtID is the unique ID of the statement as returned by method Alloc. aQueryText is the SQL statement to be prepared. If an error occurs during preparation, the SQL engine should generate an informative error message. This method must write the length of the message to aStream followed by the text of the message.

If the statement is prepared successfully then this method must return DBIERR\_NONE (declared in the FFSRBDE unit) otherwise it must return an error code applicable to the situation. The server engine forwarding the request to the SQL engine is responsible for handling the error code.

See also: Alloc

```

function SetParams(
    aStmtID : TffSqlStmtID; aNumParams : Word;
    aParamDescs : PffSqlParamInfoList; aDataBuffer : PffByteArray;
    aStream : TStream): TffResult; virtual; abstract;

PffSqlParamInfoList = ^TffSqlParamInfoList;

TffSqlParamInfoList = array[0..1023] of TffSqlParamInfo;

TffSqlParamInfo = record
    piNum : word;
    piName : string[ffcl_GeneralNameSize];
    piType : TffFieldType;
    piOffset : word;
    piLength : word;
end;

PffByteArray = ^TffByteArray;

TffByteArray = array[0..65531] of byte;

```

↳ Sets the parameters for a prepared SQL statement.

This method is called only for parameterized queries. The purpose of the method is to associate the parameter values with the parameters in the SQL statement.

aStmtID is the unique ID of the statement as returned by the Alloc method. aNumParams is the number of parameters passed to this method. aParamDescs is a pointer to an array of records describing each parameter. The fields within the record are as follows:

Field	Meaning
piNum	Parameter number, one-based.
piName	The name of the parameter as specified in the SQL statement.
piType	The FlashFiler field type (e.g., fftBoolean, fftWideChar).
piOffset	The offset of the parameter's value within aDataBuffer.
piLength	The number of bytes in aDataBuffer occupied by the parameter's value.

aDataBuffer is a buffer containing the parameter values. To find the location of a parameter value within the buffer, use the piOffset and piLength fields in the aParamDescs records.

If an error occurs, this method must generate an informative error message. This method must write the length of the error message to aStream followed by the error message itself. This method must then return an error code appropriate to the situation. The server engine forwarding the request to the SQL engine is responsible for handling the error.

If this method completes successfully, nothing is written to aStream and this method must return DBIERR\_NONE (declared in the FFSRBDE unit).

See also: Alloc



# TffSQLEngine Component

The TffSQLEngine component implements the FlashFiler SQL engine. The current implementation of this engine supports SELECT statements only. See “Syntax Conventions” on page 464 for more details on the supported syntax. The FlashFiler Server’s engine manager contains one instance of this component. All SQL related requests are routed from the server engine to the SQL engine.

See the “TffBaseSQLEngine” class on page 413 for more details.

## Hierarchy

TComponent (VCL)

❶ TffComponent (FFLLBASE) .....	78
❷ TffLoggableComponent (FFLLCOMP) .....	80
TffBaseSQLEngine (FFSQLBAS) .....	413
TffSQLEngine (FFSQLENG)	

## Properties

❷ EventLog	❷ EventLogEnabled
------------	-------------------

## Methods

Alloc	❶ FreeInstance	❶ NewInstance
Exec	FreeStmt	Prepare
ExecDirect	❷ LcLog	SetParams

# Reference Section

Alloc

method

```
function Alloc( anEngine : TffBaseServerEngine;
  aClientID : TffClientID; aDatabaseID : TffDatabaseID;
  var aStmtID : TffSqlStmtID): TffResult; override;
```

↳ Allocates resources for a new SQL statement.

This method allocates the required data structures for a new SQL statement. It generates a unique ID for the statement and returns it in the aStmtID parameter.

anEngine is the server engine forwarding the request. aClientID is the unique ID of the client submitting the request. aDatabaseID is the unique ID of the database upon which the query will operate.

If the SQL statement is allocated successfully, this method returns DBIERR\_NONE (declared in the FFSRBDE unit) otherwise it returns an error code. The server engine forwarding the request to the SQL engine handles the error code.

See also: FreeStmt

Exec

method

```
function Exec( aStmtID: TffSqlStmtID;
  var aOpenMode: TffOpenMode; var aCursorID : TffCursorID;
  aStream: TStream): TffResult; override;

TffOpenMode = (omReadOnly, omReadWrite);
```

↳ Executes an allocated, prepared SQL statement.

This method executes a SQL statement that has been allocated and prepared via the Alloc and Prepare methods. aStmtID is the unique ID of the SQL statement as returned by the Alloc method. aOpenMode indicates whether the query is intended to be read-only or read/write. FlashFiler Client uses the output value of aOpenMode to determine whether the result set is live. This method sets aOpenMode based upon the following criteria:

Input Value	Query May Be Live	Output Value
omReadOnly	No	omReadOnly
omReadOnly	Yes	omReadOnly
omReadWrite	No	omReadOnly
omReadWrite	Yes	omReadWrite

This SQL engine returns a live result set as long as the following conditions are met:

- The SELECT statement retrieves data from only one table.
- The SELECT statement does not contain the DISTINCT keyword or aggregate functions.
- The SELECT statement does not contain a GROUP BY clause.
- The table from which the result set is being gathered has read-write access at the operating system level. A read-only table never returns a modifiable result set.

If the SQL statement is executed successfully, this method opens a cursor for the statement's result set. The FlashFiler Client uses the cursor to retrieve and update the records in the result set.

If the SQL statement is executed successfully, the following occurs:

- The unique ID of the cursor is returned in parameter aCursorID.
- The TffDataDictionary describing the result set structure is written to aStream.
- This method returns DBIERR\_NONE.

If the SQL statement is not executed successfully, this method returns an error code applicable to the situation. It writes the length of the associated error message to aStream followed by the error message itself. The server engine forwarding the request to the SQL engine is responsible for handling the error.

See also: Alloc, ExecDirect, Prepare

## ExecDirect

method

```
function ExecDirect(  
    anEngine : TffBaseServerEngine; aClientID : TffClientID;  
    aDatabaseID : TffDatabaseID; aQueryText : String;  
    var aOpenMode : TffOpenMode; var aCursorID : TffCursorID;  
    aStream : TStream): TffResult; override;
```

```
TffOpenMode = (omReadOnly, omReadWrite);
```

🔗 Allocates, prepares, and executes a SQL statement.

This method allocates resources for a SQL statement, prepares the statement for execution, and executes the statement. For non-parameterized queries or non-query statements, it is a quicker way to execute the statement since there is no need to call Alloc and Prepare.

aEngine is the server engine forwarding the request to the SQL engine. aClientID is the unique ID of the client submitting the request. aDatabaseID is the unique ID of the database on which the query will operate.

aQueryText is the SQL statement to be executed. aOpenMode indicates whether the statement is intended to have a read-only or read/write result set. FlashFile Client uses the output value of aOpenMode to determine whether the result set is live. This method sets aOpenMode based upon the following criteria:

Input Value	Query May Be Live	Output Value
omReadOnly	No	omReadOnly
omReadOnly	Yes	omReadOnly
omReadWrite	No	omReadOnly
omReadWrite	Yes	omReadWrite

This SQL engine returns a live result set as long as the following conditions are met:

- The SELECT statement retrieves data from only one table.
- The SELECT statement does not contain the DISTINCT keyword or aggregate functions.
- The SELECT statement does not contain a GROUP BY clause.
- The table from which the result set is being gathered has read/write access at the operating system level. A read-only table never returns a modifiable result set.

11

If the statement is executed successfully and returns a result set, this method sets aCursorID to the unique ID of the cursor opened for the result set. This method writes the instance of TffDataDictionary describing the result set structure to aStream.

If the statement is executed successfully but does not return a result set then aCursorID is set to Zero and aStream is empty. If the statement is executed successfully, this method returns DBIERR\_NONE (declared in the FFSRBDE unit). Otherwise, it returns an error code applicable to the situation. It writes the length of the associated error message to aStream followed by the error message itself. The server engine forwarding the request to the SQL engine is responsible for handling the error.

See also: Alloc, Prepare, Exec, FreeStmt

```
function FreeStmt(aStmtID: TffSqlStmtID): TffResult; override;
```

↪ Frees the resources associated with a SQL statement.

This method is called when the FlashFiler Client has finished using an SQL statement. This method frees the resources associated with the SQL statement. If a cursor was opened for the statement's result set, the resources remain allocated until the cursor is closed. aStmtID is the unique ID of the statement as returned by the Alloc method.

See also: Alloc

```
function Prepare(aStmtID: TffSqlStmtID;  
    aQueryText : String; aStream : TStream): TffResult; override;
```

↪ Prepares an SQL statement for execution.

This method parses and optimizes the SQL statement. One way to improve application performance is to have the SQL engine parse and optimize frequently used SQL statements. The work is performed once instead of each time the query is executed.

aStmtID is the unique ID of the statement as returned by method Alloc. aQueryText is the SQL statement to be prepared. If an error occurs during preparation, this method returns an error code and generates an informative message. The length of the message, followed by the message itself, is written to aStream.

If the statement is prepared successfully then this method returns DBIERR\_NONE (declared in the FFSRBDE unit).

See also: Alloc



```

function SetParams(
    aStmtID : TffSqlStmtID; aNumParams : Word;
    aParamDescs : PffSqlParamInfoList; aDataBuffer : PffByteArray;
    aStream : TStream): TffResult; override;

PffSqlParamInfoList = ^TffSqlParamInfoList;
TffSqlParamInfoList = array[0..1023] of TffSqlParamInfo;
TffSqlParamInfo = record
    piNum : word;
    piName : string[ffcl_GeneralNameSize];
    piType : TffFieldType;
    piOffset : word;
    piLength : word;
end;

PffByteArray = ^TffByteArray;
TffByteArray = array[0..65531] of byte;

```

↳ Sets the parameters for a prepared SQL statement.

This method is called only for parameterized queries. This method associates the parameter values with the parameters in the SQL statement.

aStmtID is the unique ID of the statement as returned by the Alloc method. aNumParams is the number of parameters passed to this method. aParamDescs is a pointer to an array of records describing each parameter. The fields within the record are as follows:

Field	Meaning
piNum	Parameter number, one-based
piName	Name of the parameter as specified in the SQL statement
piType	FlashFiler field type (e.g., fftBoolean, fftWideChar)
piOffset	Offset of the parameter's value within aDataBuffer
piLength	Number of bytes in aDataBuffer occupied by the parameter's value

aDataBuffer is a buffer containing the parameter values. To find the location of a parameter value within the buffer, use the piOffset and piLength fields in the aParamDescs records.

If an error occurs, this method returns an error code and generates an informative error message. The length of the message, followed by the message itself, is written to aStream. The server engine forwarding the request to the SQL engine is responsible for handling the error.

If this method completes successfully, nothing is written to aStream and this method must return DBIERR\_NONE (declared in the FFSRBDE unit).

See also: Alloc

## TffBaseEngineManager Data Module

The TffBaseEngineManager data module is an abstract class for coordinating the start up and shut down of server engines and plugin engines. It is just like the standard TDDataModule in that it is a place for centralizing non-visual components. In this product, engine managers are used to centralize FlashFiler components.

TffBaseEngineManager's purpose is to serve as an abstract class for your own engine managers. You will not create instances of TffBaseEngineManager. Instead, you will either create instances of TffEngineManager or create your own engine manager that inherits from TffBaseEngineManager. See the “TffEngineManager data module” on page 433 for more information.

## Hierarchy

TDataModule (VCL)

TffBaseEngineManager (FFLLCOMM))

## Properties

## CmdHandler

## CmdHandlerCount

## Methods

## Process

## Shutdown

Stop

Restart

## Startup

## Reference Section

### **CmdHandler**

**read-only property**

```
property CmdHandler[aInx : Longint] : TffBaseCommandHandler
```

↪ Provides access to the command handlers routing commands to this engine manager.

Use this method to access the command handlers routing commands to this engine manager. When remote administration is implemented it will be possible to tell the engine manager to restart and shutdown from a client application. In order for the commands to reach the engine manager, you must connect the engine manager to a command handler via the EngineManager property TffBaseCommandHandler class.

aInx is zero-based and is the command handler you wish to access.

See also: CmdHandlerCount, TffBaseCommandHandler.EngineManager

### **CmdHandlerCount**

**read-only property**

```
property CmdHandlerCount : Longint
```

↪ Returns the number of command handlers routing requests to this engine manager.

Use this property to determine the upper limit of the engine manager's CmdHandler property.

See also: CmdHandler

```

procedure Process(
    msg : PffDataMessage; var handled : Boolean); virtual; abstract;

PffDataMessage = ^TffDataMessage;

TffDataMessage = record
    dmMsg : longint;
    dmClientID : TffClientID;
    dmRequestID : longint;
    dmTime : TffWord32;
    dmRetryUntil : TffWord32;
    dmErrorCode : TffResult;
    dmData : pointer;
    dmDataLen : TffMemSize;
end;

```

↳ Determines whether a client request may be processed.

When a command handler receives a request that is not handled by its server engine or its associated plugins, the command handler passes of the request to the engine manager specified by the `EngineManager` property of the `TffBaseCommandHandler` class.

The engine manager has the opportunity to process the client request. The request is contained in parameter `msg`. The fields in `msg` are shown in the following table:

Field	Description
<code>dmMsg</code>	The unique identifier for the message.
<code>dmClientID</code>	The client ID assigned to the remote transport when it called the <code>EstablishConnection</code> method.
<code>dmRequestID</code>	An identifier assigned to the request. This identifier is unique to the client but may be duplicated across clients. This identifier must be passed back in the reply to the remote transport so that it knows with which request the reply is associated.
<code>dmTime</code>	The time the message was received. <code>TffLegacyTransport</code> fills in this value once the request has been received.
<code>dmRetryUntil</code>	The point in time up to which the server may try to fulfill the request. The transports provided with <code>FlashFiler</code> do not use this field. It is provided for your convenience.

Field	Description
dmErrorCode	When first received, this should be set to DBIERR_NONE. The reply back to the remote client should contain DBIERR_NONE if the request was fulfilled without error. Otherwise it must contain an error code describing the problem.
dmData	The data associated with the request. This data is specified when the remote transport's Post or Request method is called.
dmDataLen	The length of dmData. This value is specified when the remote transport's Post or Request method is called.

If the engine manager recognizes and can handle the message, it must set the handled parameter to True; otherwise, it must set the handled parameter to False.

If the engine manager handles the message, the engine manager is expected to send a reply back to the client. The engine manager must send the reply to the client via the transport identified by class function the CurrentTransport class function of the TffBaseTransport class.

See also: TffBaseCommandHandler.EngineManager, TffBaseTransport.CurrentTransport

## **Restart** virtual abstract method

```
procedure Restart; virtual; abstract;
```

↳ Restarts the server engines and plugin engines contained by the engine manager.

The implementation of this method must restart all server engines and plugin engines contained by the engine manager. One way to do this is to call the Startup and Shutdown methods of each engine. The engines will in turn start up and shut down their command handlers and transports.

See also: Shutdown, TffStateComponent.Shutdown, Startup, TffComponent.Startup, Stop

## **Shutdown** virtual abstract method

```
procedure Shutdown; virtual; abstract;
```

↳ Shuts down all server engines and plugin engines contained by the engine manager.

The implementation of this method must shut down all server engines and plugin engines contained by the engine manager. One way to do this is to call the Shutdown method of each engine. The engines will shut down their command handlers and transports.

See also: Restart, TffStateComponent.Shutdown, Startup, Stop

---

```
procedure Startup; virtual; abstract;
```

↳ Starts all server engines and plugin engines contained by the engine manager.

The implementation of this method must start all server engines and plugin engines contained by the engine manager. One way to do this is to call the `Startup` method of each engine. The engines will start up their command handlers and transports.

See also: `Restart`, `Shutdown`, `TffStateComponent.Startup`, `Stop`

---

```
procedure Stop; virtual; abstract;
```

↳ Stops all server engines and plugin engines contained by the engine manager.

The implementation of this method must stop all server engines and plugin engines.

**Note:** Stopping is different than shutting down. When shutting down, the command handlers and transports associated with the engines are also shut down. This means the server can no longer receive messages from client applications.

When stopping, the engines shut down but their associated command handlers and transports are still active. This leaves open a route from the client application to the engine manager, allowing a client to start up the server after it has been stopped. One way to stop an engine is to call its `Stop` method.

See also: `Restart`, `Shutdown`, `Startup`, `TffStateComponent.Stop`

# TffEngineManager Data Module

The TffEngineManager data module delivered with FlashFiler contains the components required to implement a FlashFiler Server. It contains a server engine, SQL engine, an event log, a command handler, three transports, a thread pool, and a security monitor. The FlashFiler Server UI uses the engine manager to coordinate the start up and shut down of the server engines.

You can alter the contents of the engine manager to suit your needs. You can create a new engine manager using the FlashFiler Engine Manager wizard. To do so, use the following steps:

1. Start Delphi and open the project that is to contain the new engine manager.
2. Select the File | New menu item. The New Items window appears.
3. Click on the Data Modules page of the New Items dialog. One of the data modules on the page is "FlashFiler 2 Engine Manager".
4. Select the item labeled "FlashFiler 2 Engine Manager" and click the OK button. The Select FlashFiler Protocols window appears. This window allows you to specify which transports are to be created in the new engine manager.
5. Select the appropriate transports and then click OK. The engine manager is created.

The current FlashFiler Server UI is tied closely with the engine manager. If you are generating a new engine manager for use by the FlashFiler Server UI, be sure to give the new engine manager the same name as the existing engine manager.

## Hierarchy

TDataModule (VCL)

    ❶ TffBaseEngineManager (FFLLCOMM)

## Properties

❶ CmdHandler

❶ CmdHandlerCount

EventLogEnabled

## Methods

GetServerEngines

Restart

Stop

GetTransports

Shutdown

Process

Startup



## Reference Section

### EventLogEnabled

property

property EventLogEnabled : Boolean

Default: False

↳ Enables or disables the logging of errors by the server engines.

Use this method to control whether the server engines within the engine manager may log errors. The standard FlashFiler engine manager includes an instance of `TffEventLog` and all loggable components within the engine manager are connected to that event log. If you set this property to `True`, the engine manager enables event logging for all components inheriting from class `TffLoggableComponent`. If you set this property to `False`, the engine manager disables event logging for all components inheriting from class `TffLoggableComponent`.

If you read the value of this property, the engine manager returns the value of the first server engine's `EventLogEnabled` property. The engine manager specifically looks for server engines instead of `TffLoggableComponents` because it is possible to control transport logging from the FlashFiler Server UI independent of the server engine logging. In other words, you could enable transport logging through the FlashFiler Server UI and still leave server engine logging disabled.

### GetServerEngines

method

```
procedure GetServerEngines(var aServerList : TffList);
```

11 ↳ Retrieves a list of the server engines contained by the engine manager.

The FlashFiler Server UI uses this method to retrieve a list of the server engines within the engine manager. The UI uses this list to populate its display. `aServerList` is an existing instance of `TffList`. The engine manager adds one instance of `TffIntListItem` to `aServerList` for each server engine. To access a server engine, you must cast the `TffIntListItem`'s key to type `TffBaseServerEngine`, as shown in the following example:

```
anEngineMgr.GetServerEngines(aList);
for anIndex := 0 to pred(aList.Count) do begin
    anEngine :=
    TffBaseServerEngine(TffIntListItem(aList[anIndex]).KeyAsInt);
    {Do something with the engine...}
end;
```

Your code is responsible for freeing `aServerList`. When you free `aServerList`, the instances of `TffIntListItem` are freed but the actual server engines are untouched.

```
procedure GetTransports(
    aServer : TffIntermediateServerEngine; var aTransList : TffList);
```

↪ Retrieves the transports associated with a specific server engine.

The FlashFile Server UI uses this method to retrieve a list of transports associated with the currently selected server engine. The UI uses this list to populate its display. aServer is the server engine for which transports are to be retrieved. aTransList is an existing instance of TffList. The engine manager adds one instance of TffIntListItem to aTransList for each transport associated with aServer. Note that a transport is connected to a command handler that is connected to aServer.

To access a transport, you must cast the TffIntListItem's key to type TffBaseTransport, as shown in the following example:

```
anEngineMgr.GetServerEngines(aList);
for anIndex := 0 to pred(aList.Count) do begin
    anEngine := TffBaseServerEngine(
        TffIntListItem(aList[anIndex]).KeyAsInt);
    aTransList := TffList.Create;
    anEngineMgr.GetTransports(anEngine, aTransList);
    for aTransIndex := 0 to pred(aTransList.Count) do begin
        aTransport :=
            TffBaseTransport(
                TffIntListItem(aTransList[aTransIndex]).KeyAsInt);
        {Do something with the transport...}
    end;
end;
```

Your code is responsible for freeing aTransList. When you free aTransList, the instances of TffIntListItem are freed but the actual transports are untouched.

```
procedure Process(  
    Msg : PffDataMessage; var Handled : Boolean); override;  
  
    PffDataMessage = ^TffDataMessage;  
  
    TffDataMessage = record  
        dmMsg : longint;  
        dmClientID : TffClientID;  
        dmRequestID : longint;  
        dmTime : TffWord32;  
        dmRetryUntil : TffWord32;  
        dmErrorCode : TffResult;  
        dmData : pointer;  
        dmDataLen : TffMemSize;  
    end;
```

↳ Determines whether a client request may be processed.

When a command handler receives a request that is not handled by its server engine or its associated plugins, the command handler passes of the request to the engine manager specified by the `EngineManager` property of the `TffBaseCommandHandler` class.

The engine manager has the opportunity to process the client request. The request is contained in parameter `msg`. The fields in `msg` are shown in the following table:

Field	Description
dmMsg	The unique identifier for the message.
dmClientID	The client ID assigned to the remote transport when it called the <code>EstablishConnection</code> method.
dmRequestID	An identifier assigned to the request. This identifier is unique to the client but may be duplicated across clients. This identifier must be passed back in the reply to the remote transport so that it knows with which request the reply is associated.
dmTime	The time the message was received. <code>TffLegacyTransport</code> fills in this value once the request has been received.
dmRetryUntil	The point in time up to which the server may try to fulfill the request. The transports provided with <code>FlashFiler</code> do not use this field. It is provided for your convenience.

Field	Description
dmErrorCode	When first received, this should be set to DBIERR_NONE. The reply back to the remote client should contain DBIERR_NONE if the request was fulfilled without error. Otherwise it must contain an error code describing the problem.
dmData	The data associated with the request. This data is specified when the remote transport's Post or Request method is called.
dmDataLen	The length of dmData. This value is specified when the remote transport's Post or Request method is called.

The engine manager sets `Handled` to `False` unless one of the following messages is received:

Message	Action Taken
ffnmServerRestart	Sets <code>Handled</code> to <code>True</code> and calls the <code>Restart</code> method.
ffnmServerShutdown	Sets <code>Handled</code> to <code>True</code> and calls the <code>Shutdown</code> method.
ffnmServerStartup	Sets <code>Handled</code> to <code>True</code> and calls the <code>Startup</code> method.
ffnmServerStop	Sets <code>Handled</code> to <code>True</code> and calls the <code>Stop</code> method.

See also: `TffBaseCommandHandler.EngineManager`

<b>Restart</b>	<b>method</b>
----------------	---------------

```
procedure Restart; override;
```

↪ Restarts the server engines and plugin engines contained by the engine manager.

Use this method to restart the state components within the engine manager. The engine manager calls the `Shutdown` method of each server engine and plugin engine. It then calls the `Startup` method of each server engine and plugin engine.

See also: `Shutdown`, `Startup`, `Stop`

## Shutdown

method

```
procedure Start; override;
```

- ↳ Shuts down the server engines and plugin engines contained by the engine manager.

Use this method to shut down the engines within the engine manager. The engine manager calls the Shutdown method of each engine. The engines, in turn, shut down their associated command handlers and transports.

See also: Restart, Startup, Stop

## Startup

method

```
procedure Startup; override;
```

- ↳ Starts the server engines and plugin engines contained by the engine manager.

Use this method to start up the engines within the engine manager. The engine manager calls the Startup method of each engine. The engines, in turn, start their associated command handlers and transports.

See also: Restart, Shutdown, Stop

## Stop

method

```
procedure Stop; override;
```

- ↳ Stops the server engines and plugin engines contained by the engine manager.

Use this method to temporarily stop the engines within the engine manager. Note that stopping is different than shutting down. When shutting down, the command handlers and transports associated with the engines are also shut down. This means the server can no longer receive messages from client applications.

When stopping, the engines shut down but their associated command handlers and transports are still active. This leaves open a route from the client application to the engine manager, allowing a client to start the server after it has been stopped.

See also: Restart, Shutdown, Startup

---

## Chapter 12: Plugins and Extenders

This chapter is a class reference for developers interested in extending the functionality of FlashFiler Server. For a practical explanation of the components and inner workings of the FlashFiler Server, see “Following the Flow” on page 270 and “Modifying the FlashFiler Server” on page 275, both in Chapter 8.

If you wish to add new functionality to FlashFiler Server, create a plugin and plugin command handler. If you wish to monitor and extend the events occurring within a FlashFiler Server, create an engine monitor and engine extender.

### Plugins

Use plugins to add new functionality to a FlashFiler Server. In the same way that a TffServerEngine component may be embedded in a client application or a FlashFiler Server, a plugin engine is a component that may be embedded in a client application or a FlashFiler Server.

The foundational classes for plugins are TffBasePluginEngine and TffBasePluginCommandHandler. Create a descendant of TffBasePluginEngine to implement your required functionality. Plugin engines do not have a pre-defined interface because you decide what interface is needed. When you place your plugin engine into a client application, using the plugin is as simple as inheriting from TffBasePluginEngine and adjusting the client application to call the plugin's methods.

If your plugin can be used on either a FlashFiler client or server, the steps are more involved. Figure 12.1 illustrates how a plugin command is routed from the client to the FlashFiler Server.

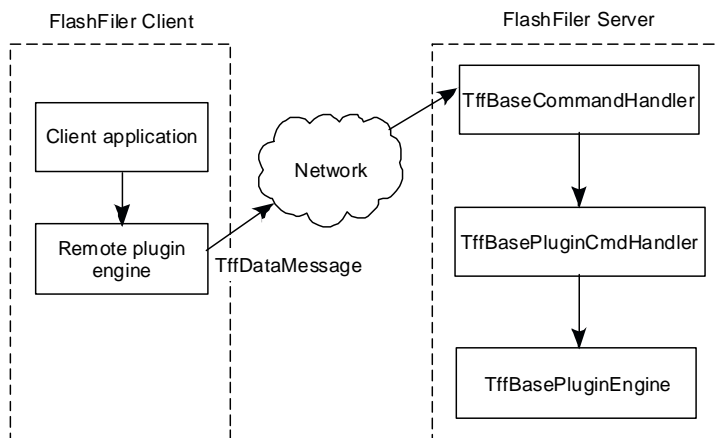


Figure 12.1: Plugin command routing.

A summary of the steps is as follows:

1. Create an abstract plugin engine that inherits from `TffBasePluginEngine`. The abstract class contains virtual abstract methods describing the required functionality. For example, if your plugin records client application errors then create class `TffBaseErrorPlugin`.
2. From the abstract plugin engine, create a real plugin engine that implements the required functionality. Continuing the example, create an instance of the `TffErrorPlugin` class.
3. From the abstract plugin engine, create a remote plugin engine. This is analogous to `TffRemoteServerEngine` which implements the interface of `TffBaseServerEngine`, serving as a conduit to a remote server. Add the property `Transport: TffBaseTransport` to the remote plugin engine. The remote plugin engine class passes commands from the client application through the transport to the remote server. In this example, create an instance of the `TffRemoteErrorPlugin` class.
4. Create a plugin command handler inheriting from `TffBasePluginCommandHandler`. In the FlashFiler Server, the plugin command handler attaches to a `TffBaseServerCommandHandler` property via `TffBasePluginCommandHandler.CommandHandler`. When the server command handler receives a message it does not recognize, it forwards the message to the plugin command handler. If the plugin engine can handle the message, the plugin command handler calls the appropriate method of the plugin engine. In our example, this would be class `TffErrorPluginCommandHandler`.

# Event monitoring

When client applications connect to and interact with the FlashFiler Server, the TffServerEngine creates and uses server-side objects. For example, when a client opens a table, the server creates a cursor. The server uses the cursor to track the client's position within the table. See “Chapter 11: TffServerEngine Component” on page 407 for more details.

You can monitor and participate in the actions taken by server-side objects using the TffBaseEngineMonitor and TffBaseEngineExtender classes. For example, you can track the changes made to records in a particular table. You can log who made the change or you can even cause the change to be rejected if it does not meet certain criteria. This section describes how you can monitor and extend the server-side objects. For a detailed example, see Chapter 8 “Extending the Server Engine” on page 310.

## Server object classes

As mentioned previously, TffServerEngine instantiates various classes in order to carry out its work. In general, the classes used by the server engine correspond to the components you place in FlashFiler client applications. The server object classes are listed in Table 12.1.

**Table 12.1:** *Server object classes.*

Class	Description
TffSrSession	This class corresponds to the TffSession component. One instance of TffSrSession is created per active TffSession in a FlashFiler client application. If you do not explicitly place a TffSession in your application, the FlashFiler client engine creates a TffSession behind the scenes (i.e., the automatic session) resulting in an instance of TffSrSession within the server engine. The server engine does not notify monitors or extenders of any actions relating to TffSrSession.
TffSrDatabase	This class corresponds to TffDatabase. One instance of TffSrDatabase is created per active TffDatabase in a FlashFiler client application.



**Table 12.1:** *Server object classes. (continued)*

Class	Description
TffSrClient	This class corresponds to TffClient and TffCommsEngine. One instance of TffSrClient is created per active TffClient and TffCommsEngine in a FlashFiler client application. If you do not explicitly place one of these components in your application, the FlashFiler client engine creates at least one TffClient behind the scenes (i.e., the automatic client) resulting in the creation of a TffSrClient instance within the server engine.
TffSrBaseCursor	Instances of this class are created when a TffTable or TffQuery is opened in the client application. TffSrBaseCursor is the base class for all cursors created on the server. The server engine does not directly create instances of TffSrBaseCursor but does create instances of its descendant classes. One cursor is created per open table or query.

## Engine monitor

An engine monitor is a non-visual component that tells the server engine what server-side classes are to be monitored. The engine monitor registers interest in a server-side class using its AddInterest method. When the server engine next creates an instance of that class, it calls the monitor's Interested method.

At that point, the monitor decides whether to extend that specific server-side object. If it does decide to extend the server-side object, the monitor's Interested method returns an instance of TffBaseEngineExtender. The server engine attaches the extender to the server-side object and calls the extender's Notify method when something happens to the server-side object.

In order to monitor events within a server engine, do the following:

1. Decide which server-side classes you need to monitor.
2. Create a class that inherits from TffBaseEngineMonitor. This results in a non-visual component that you place in the engine manager and connect to a server engine.
3. Write code that causes the engine monitor to register interest in the appropriate server-side class(es).
4. Implement the monitor's Interested method. It should verify the passed server-side object is of the correct class and return an instance of your engine extender (see next step) as its result. If the decision making within Interested decides it does not need to perform any work related to the server-side object, return nil.

5. Create a class that inherits from `TffBaseEngineExtender`. This is the engine extender returned by `TffBaseEngineMonitor.Interested`.
6. Within the extender, set the value of `FActions` (i.e., the `TffBaseEngineExtender.InterestedActions` property) to the set of actions of which the extender wants to be notified. As shown in the following section, there are a large number of actions that may happen to various objects. The extender improves performance by explicitly stating the actions in which it is interested.
7. Implement the extender's `Notify` method.

## Engine notifications

The server engine notifies extenders of operations affecting server-side objects. It does so by passing an action value from `TffEngineAction` (declared in the `FFLENG` unit) to the extender's `Notify` method. Each action represents an operation and a stage within that operation. There are three stages of server engine notifications listed in Table 12.2.

**Table 12.2:** *Server engine notification stages.*

Stage	Meaning
Before	The operation is about to take place. The engine extender may prohibit the action by returning a non-zero error code. See the <code>TffBaseEngineExtender.Notify</code> method for more details.
After	The operation has concluded successfully.
Failure	The operation has failed.

When notifying extenders of an operation, the server engine sometimes associates a `Failure` action with a `Before` action. If an extender prohibits the operation then the server engine passes the `Failure` action to the preceding extenders. In those cases when there is no relevant `Failure` action, the server engine associates `ffeNoAction` with the `Before` action.

The actions passed to `TffBaseEngineExtender.Notify` depend upon the server-side class and operation. The `TffSrClient` actions are listed in Table 12.3.

**Table 12.3:** *TffSrClient notifications.*

Action	When Notified
<code>ffeaAfterCreateClient</code>	After a new client has been added to the server engine. If the engine extender's <code>Notify</code> method returns a non-zero result, the client connection is not allowed and the client is removed.
<code>ffeaBeforeChgAliasPath</code>	Unused.
<code>ffeaBeforeDBInsert</code>	Before adding an alias.

**Table 12.3:** *TffSrClient* notifications. (continued)

Action	When Notified
ffeaBeforeDBRead	Before the database alias list is retrieved from the server, before a database is opened, and before the database's alias is retrieved from the server.
ffeaBeforeDBUpdate	Unused.
ffeaBeforeRemoveClient	Before a client is removed from the server engine.
ffeaBeforeDBDelete	Before an alias is deleted or modified.
ffeaChgAliasPathFail	Unused.
ffeaDBDeleteFail	After failing to delete or modify an alias.
ffeaDBInsertFail	After failing to add an alias.
ffeaDBUpdateFail	Unused.

The *TffSrDatabase* actions are listed in Table 12.4.

**Table 12.4:** *TffSrDatabase* notifications.

Action	When Notified
ffeaAfterCommit	After a transaction is committed.
ffeaAfterRollback	After a transaction is rolled back.
ffeaAfterStartTrans	After a transaction is started.
ffeaBeforeCommit	Before a transaction is committed.
ffeaBeforeRebuildInx	Before rebuilding an index.
ffeaBeforeRollback	Before a transaction is rolled back.
ffeaBeforeTabDelete	Before deleting a table.
ffeaBeforeTabInsert	Before a table is created.
ffeaBeforeTabPack	Before a table is packed.
ffeaBeforeTabRead	This action occurs when checking the existence of a table, retrieving a list of tables, before exclusively locking a table, before retrieving a table's data dictionary, and before opening a table.
ffeaBeforeTabRestruct	Before a table is restructured.
ffeaBeforeTabUpdate	Before renaming a table.

**Table 12.4:** *TffSrDatabase notifications. (continued)*

Action	When Notified
ffeaTabDeleteFail	After failing to delete a table.
ffeaTabInsertFail	After failing to create a table.
ffeaTabPackFail	After failing to pack a table.
ffeaTabRebuildInxFail	After failing to rebuild an index.
ffeaTabRestructFail	After failing to restructure a table.
ffeaTabUpdateFail	After failing to rename a table.

The TffSrCursor action are listed in Table 12.5.

**Table 12.5:** *TffSrCursor notifications.*

Action	When Notified
ffeaAfterBLOBCreate	After a BLOB is created.
ffeaAfterBLOBDelete	After a BLOB is deleted.
ffeaAfterBLOBFree	After freeing a BLOB stream.
ffeaAfterBLOBTruncate	After truncating a BLOB.
ffeaAfterBLOBWrite	After bytes are written to a BLOB.
ffeaAfterFileBLOBAdd	After adding a file BLOB.
ffeaAfterRecDelete	After a record has been successfully deleted.
ffeaAfterRecInsert	After a single record or record batch has been successfully inserted into the database.
ffeaAfterRecUpdate	After a record has been successfully modified.
ffeaAfterTableLock	After a table lock is successfully acquired.
ffeaBeforeAddInx	Before adding an index to a table.
ffeaBeforeBLOBCreate	Before a BLOB is created.
ffeaBeforeBLOBDelete	Before a BLOB is deleted.
ffeaBeforeBLOBFree	Before freeing a BLOB stream.
ffeaBeforeBLOBGetLength	After a BLOB is created.
ffeaBeforeBLOBRead	Before bytes are read from a BLOB.
ffeaBeforeBLOBTruncate	Before truncating a BLOB.
ffeaBeforeBLOBWrite	Before bytes are written to a BLOB.
ffeaBeforeFileBLOBAdd	Before adding a file BLOB.

**Table 12.5:** *TffSrCursor* notifications. (continued)

Action	When Notified
<code>ffeaBeforeCursorClose</code>	Before a cursor is closed.
<code>ffeaBeforeRecDelete</code>	Before a record or record batch is deleted.
<code>ffeaBeforeRecInsert</code>	Before a single record or record batch is inserted into the database.
<code>ffeaBeforeRecRead</code>	Before a single record or record batch is retrieved from the database.
<code>ffeaBeforeRecUpdate</code>	Before a record is modified.
<code>ffeaBeforeTabDelete</code>	Before deleting a table's index and before emptying a table.
<code>ffeaBeforeTableLock</code>	Before a table lock is acquired.
<code>ffeaBeforeTabRead</code>	Before a bookmarks are compared, when determining if a record is locked, when retrieving a table's record count, and when determining if a table is locked.
<code>ffeaBeforeTabUpdate</code>	When retrieving the next autoincrement value and before resetting the value of a table's autoincrement seed.
<code>ffeaBLOBCreateFail</code>	After failing to create a BLOB.
<code>ffeaBLOBDeleteFail</code>	After failing to delete a BLOB.
<code>ffeaBLOBFreeFail</code>	After failing to free a BLOB stream.
<code>ffeaBLOBTruncateFail</code>	After failing to truncate a BLOB.
<code>ffeaBLOBWriteFail</code>	After failing to write bytes to a BLOB.
<code>ffeaDeleteRecFail</code>	After a record delete has failed.
<code>ffeaFileBLOBAddFail</code>	After failing to add a file BLOB.
<code>ffeaInsertRecFail</code>	After a failure has occurred when inserting a single record or a batch of records.
<code>ffeaTabAddInxFail</code>	After failing to add an index to a table.
<code>ffeaTabDeleteFail</code>	After one of the actions listed for <code>ffeaBeforeTabDelete</code> has failed.
<code>ffeaTableLockFail</code>	After failing to acquire a table lock.
<code>ffeaTabUpdateFail</code>	After one of the actions listed for <code>ffeaBeforeTabUpdate</code> has failed.
<code>ffeaUpdateRecFail</code>	After a record update has failed.

---

# TffBasePluginCommandHandler Class

The TffBasePluginCommandHandler class defines the basic interface for a plugin command handler. A plugin command handler receives messages from an instance of TffBaseCommandHandler and routes the messages to a plugin engine. Also, a plugin command handler may choose to handle certain messages independent of a plugin engine.

## Hierarchy

TComponent (VCL)

❶ TffComponent (FFLLBASE) .....	78
❷ TffLoggableComponent (FFLLCOMP) .....	80
❸ TffStateComponent (FFLLCOMP) .....	82

        TffBasePluginCommandHandler

## Properties

CommandHandler	❷ EventLogEnabled
❷ EventLog	❸ State

## Methods

❶ FreeInstance	Process	❸ ScShutdown
❷ LcLog	❸ ScInitialize	❸ ScStartup
❶ NewInstance	❸ ScPrepareForShutdown	

## Events

❸ OnStateChange
-----------------

## Reference Section

### CommandHandler

property

```
property CommandHandler : TffBaseCommandHandler
```

↳ Specifies which command handler routes messages to the plugin command handler.

In order for the plugin command handler to receive messages, it must be connected to a command handler. Set `CommandHandler` to the `TffBaseCommandHandler` instance that will perform command handling. Instances of `TffServerCommandHandler` route messages to plugin command handlers only when the `TffServerCommandHandler` does not recognize those messages.

See also: `Process`

### Process

method

```
procedure Process(Msg : PffDataMessage; var handled) : Boolean;  
PffDataMessage = ^TffDataMessage;  
  
TffDataMessage = record  
  dmMsg : longint;  
  dmClientID : TffClientID;  
  dmRequestID : longint;  
  dmTime : TffWord32;  
  dmRetryUntil : TffWord32;  
  dmErrorCode : TffResult;  
  dmData : pointer;  
  dmDataLen : TffMemSize;  
end;
```

12

↳ Processes a message received from a command handler.

When an instance of `TffBaseCommandHandler` receives a message that it does not recognize, it routes the message to the first of its registered plugin command handlers. If the plugin command handler recognizes and handles the message, indicated by setting the `handled` parameter to `True`, the plugin command handler sends a reply to the remote client. If the plugin command handler does not recognize the message, the `TffBaseCommandHandler` submits the message to the next registered plugin command handler, and so on.

Msg is the message received by the TffBaseCommandHandler from a transport. The following table defines the values of the fields in Msg:

Value	Meaning
dmClientID	The client ID assigned to the remote transport when it called the EstablishConnection method.
dmRequestID	An identifier assigned to the request. This identifier is unique to the client but may be duplicated across clients. This identifier must be passed back in the reply to the remote transport so that it knows with which request the reply is associated.
dmTime	The time the message was received. TffLegacyTransport fills in this value once the request has been received.
dmRetryUntil	The point in time up to which the server may try to fulfill the request. The transports provided with FlashFiler do not use this field. It is provided for your convenience.
dmErrorCode	When first received, this should be set to DBIERR_NONE. The reply back to the remote client should contain DBIERR_NONE if the request was fulfilled without error. Otherwise it must contain an error code describing the problem.
dmData	The data associated with the request. This data is specified when the remote transport's Post or Request method is called.
dmDataLen	The length of dmData. This value is specified when the remote transport's Post or Request method is called.

If your plugin command handler recognizes the message, set the handled parameter to True.

See also: CommandHandler



## TffBasePluginEngine Class

The `TffBasePluginEngine` class defines the basic interface for your own plugin engine. Use a plugin engine to add new commands to the FlashFiler Server. For more details on creating your own plugin engine, see Chapter 8 Advanced Architecture “Creating Plugins” on page 300 and “Plugins” on page 439 within this chapter.

`TffBasePluginEngine` contains protected methods for managing internal details but it does not expose any public properties, methods, or events. This is because you will create those items as needed for your custom plugin.

### Plugin states

Plugin engines, like server engines, contain a state engine. The FlashFiler engine manager (see “`TffEngineManager` Data Module” on page 433) controls the state of the plugin engine. When the FlashFiler Server is started, the engine manager sets the state of all contained server engines and plugin engines to `ffesStarted`. When the server is shut down, the engine manager sets the state back to `ffesInactive`.

A plugin engine may be in one of several states as shown in Figure 12.1.

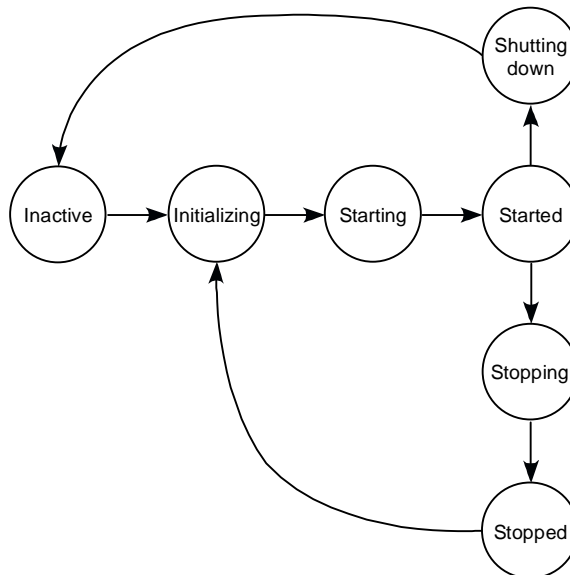


Figure 12.2: Plugin engine state diagram.

The circles represent states. Lines between the circles represent the path taken by the plugin engine's internal state engine as it goes from a start state to a destination state. For example, when a plugin engine is first created its State is set to `ffesInactive`. Setting the State to `Started` causes the state engine to change the state from `Inactive` to `Initializing`, then to `Starting`, and finally to `Started`.

As a plugin engine progresses from one state to the next, the state engine calls protected methods within the plugin engine. See “`TffStateComponent Class`” on page 82 for a full description of the protected state methods. By overriding these methods, you can perform initialization, startup, and shut down actions required by your custom plugin.

**Note:** When the state of a plugin engine changes, it transfers that state to the plugin command handlers that are associated with the plugin engine.

The plugin engine states are described in the Table 12.6.

**Table 12.6:** *Plugin Engine states.*

State	Meaning
<code>ffesInactive</code>	The plugin engine is not doing any work. If the engine reached this state after being in state <code>ffesShuttingDown</code> then the state engine calls method <code>TffStateComponent.ScShutdown</code> .
<code>ffesInitializing</code>	The plugin engine is being prepared for work. The state engine calls method <code>TffStateComponent.ScInitialize</code> .
<code>ffesStarting</code>	The plugin engine is starting. The state engine calls method <code>TffStateComponent.ScStartup</code> .
<code>ffesStarted</code>	The plugin engine has successfully started and is ready for work.
<code>ffesShuttingDown</code>	The plugin engine has been told to shut down. The state engine calls <code>TffStateComponent.ScPrepareForShutdown</code> .
<code>ffesStopping</code>	The plugin engine is in the process of being stopped. The plugin command handlers associated with the engine are left in an active state so that they may handle commands. A handled command may simply mean the plugin command handler notices the plugin engine is inactive and sends a reply to the client stating such.

Table 12.6: *Plugin Engine states. (continued)*

State	Meaning
ffesStopped	The plugin engine has been stopped. The plugin command handlers associated with the engine are left in an active state so that they may handle commands. A handled command may simply mean the plugin command handler notices the plugin engine is inactive and sends a reply to the client stating such.
ffesUnsupported	The plugin engine is not supported on this workstation. This state is not passed on to the associated plugin command handlers.
ffesFailed	An error occurred during startup of the plugin engine. The engine may not be used until the application is restarted. This state is not passed on to the associated plugin command handlers.

Hierarchy

TComponent (VCL)

- ❶ TffComponent (FFLLBASE) .....78
  - ❷ TffLoggableComponent (FFLLCOMP) .....80
    - ❸ TffStateComponent (FFLLCOMP) .....82
      - TffBasePluginEngine (FFLLCOMM)

Properties

❷ EventLog	❷ EventLogEnabled	❸ State
------------	-------------------	---------

Methods

❶ FreeInstance	❸ ScPrepareForShutdown	❸ Startup
❷ LcLog	❸ ScShutdown	❸ Stop
❶ NewInstance	❸ ScStartup	
❸ ScInitialize	❸ Shutdown	

Events

❸ OnStateChange
-----------------

---

# TffBaseEngineExtender Class

The TffBaseEngineExtender class defines the basic interface for an engine extender. As described in the Event Monitoring section, the Interested method of the TffBaseEngineMonitor class returns an engine extender when it wants to monitor and/or extend the actions affecting a server-side object.

The extender's InterestedActions property identifies the actions in which the extender is interested. The server engine calls the extender's Notify method when one an action takes place.

## Hierarchy

TComponent (VCL)	
❶ TffObject (FLLBASE) .....	94
TffBaseEngineExtender (FLLLENG)	

## Properties

InterestedActions

## Methods

❶ FreeInstance	❶ NewInstance	Notify
----------------	---------------	--------

## Reference Section

### InterestedActions

read-only property

```
property InterestedActions : TffInterestedActions
TffInterestedActions = set of TffEngineAction
```

↳ Defines the set of actions that result in a call to the engine extender's Notify method.

The actions seen by an engine extender depend upon the server-side class being monitored. For a list of the actions, organized by server-side class, see “Engine notifications” on page 443. To the outside world, this is a read-only property. The engine extender sets this property using the internal variable FAction.

This property is provided as a performance improvement. If the extender does not set this variable, its Notify method is called for all actions, which may decrease performance. If an extender is notified of an action and a failure occurs during the course of an action, the failure is always reported via the extender's Notify method regardless of InterestedActions.

The following code example shows a cursor extender stating its interest in record inserts, updates, and deletions:

```
Self.FAction :=
  [ffeaBeforeRecInsert, ffeaAfterRecInsert, ffeaBeforeRecUpdate,
   ffeaAfterRecUpdate, ffeaBeforeRecDelete, ffeaAfterRecDelete];
```

See also: Notify

```
function Notify(aServerObject : TffObject;  
    aAction : TffEngineAction) : TffResult;
```

↪ Notifies the extender that an action is about to take place or has taken place.

The Notify method is called when an action is about to be performed or has been performed on aServerObject. The value of aAction depends upon aServerObject's class. For a list of the actions, organized by server-side class, see "Engine notifications" on page 443.

If the action is about to be performed, the extender may prohibit the action by returning a non-zero error code. The server engine notifies engine extenders in the order they were attached to aServerObject. In the event that engine extender D returns a non-zero error code after engine extenders A, B, and C approve the action, the server engine passes a failure action to the Notify method of engine extenders A, B, and C.

If an engine extender returns a non-zero error code, the error code is included in the reply back to the client application.

This method may be called after an action has completed successfully or failed. For example, if the extender is monitoring record inserts then it may receive a ffeaAfterRecInsert or ffeaInsertRecFail notification.

# TffBaseEngineMonitor Class

The TffBaseEngineMonitor class defines the basic interface of a component for monitoring events within a FlashFiler Server. If you want to extend the functionality of the server or monitor events within the server, you must create an engine monitor that inherits from this class. You must also create a class that inherits from TffBaseEngineExtender.

As described in “Server object classes” on page 441, the FlashFiler Server creates various types of objects. For example, when a table or query is opened then the server creates an instance of TffSrBaseCursor. Using an instance of TffBaseEngineMonitor, you may register interest in one or more types of server objects. The server gives the engine monitor an opportunity to monitor the server objects and be notified of events pertaining to those objects. See the “Interested” method on page 458 for more details.

## Hierarchy

TComponent (VCL)	
❶ TffComponent (FFLLBASE)	78
❷ TffLoggableComponent (FFLLCOMP)	80
❸ TffStateComponent (FFLLCOMP)	82
TffBaseEngineMonitor (FFLENG)	

## Properties

❷ EventLog	ServerEngine
❷ EventLogEnabled	❸ State

## Methods

AddInterest	❶ NewInstance	❸ ScPrepareForShutdown
❶ FreeInstance	RemoveAllInterest	❸ ScShutdown
Interested	RemoveInterest	❸ ScStartup
❷ LcLog	❸ ScInitialize	

## Events

❸ OnStateChange
-----------------

## Reference Section

### AddInterest

method

```
procedure AddInterest(anObjectClass : TffServerObjectClass);
```

↳ Registers interest in a specific type of server object.

Use this method to tell the server engine that the engine monitor wants to be alerted when an object of type `TffServerObjectClass` is created. Subsequently, each time the server engine creates an object of that class it calls the monitor's `Interested` method.

`anObjectClass` is the class of a server-side object. See “Server object classes” on page 441 for more details.

When the monitor is no longer interested in the object class, call the `RemoveAllInterest` or `RemoveInterest` method.

The following code example illustrates how the `TffSecurityMonitor` registers interest in various server object classes:

```
procedure TffSecurityMonitor.bemSetServerEngine
  (anEngine : TFFBaseServerEngine);
begin
  inherited bemSetServerEngine(anEngine);
  AddInterest(TffSrClient);
  AddInterest(TffSrDatabase);
  AddInterest(TffSrBaseCursor);
end;
```

**Note:** The `TffSecurityMonitor` registers interest as soon as it is connected with a server engine. It is valid to register interest at other times.

See also: `Interested`, `RemoveAllInterest`, `RemoveInterest`, `ServerEngine`



```
function Interested(
  aServerObject : TffObject) : TffBaseEngineExtender;
virtual; abstract;
```

↳ Queries the engine monitor for interest in a specific server object.

When you create your own engine monitor, you must override this method.

After a server engine has created an instance of a server object class, it determines whether any engine monitors have registered interest in the class. If an engine monitor has registered interest in the class, the server engine calls the monitor's Interested function.

aServerObject is the server object created by the server engine. Its class type will be one of those for which the monitor registered interest. It is the responsibility of the monitor to test the validity of the server object's class type.

If the engine monitor wants to participate in any of the actions undertaken by the server object, the engine monitor must create an engine extender and pass it back as this function's result. The engine extender must be an instance of a subclass of TffBaseEngineExtender.

If the engine monitor does not want to be notified of actions pertaining to the object, set the result to nil.

The following example, from the FFSRSEC unit, illustrates how TffSecurityMonitor determines whether it is interested in a server object:

```
function TffSecurityMonitor.Interested
  (aServerObject : TffObject) : TffBaseEngineExtender;
begin
  {This should always be a TFFSrBaseCursor, TFFSrClient
   or TFFDatabase, but we need to check to be sure}
  if ((aServerObject is TffSrBaseCursor) or
      (aServerObject is TffSrDatabase) or
      (aServerObject is TffSrClient)) then
    Result := TffSecurityExtender.Create(self)
  else
    Result := nil;
end;
```

See also: AddInterest, ServerEngine, TffBaseEngineExtender

## RemoveAllInterest

method

```
procedure RemoveAllInterest;
```

↳ Removes all interest registrations associated with the engine monitor.

Use this method to remove all the engine monitor's interests in server object classes. The associated `ServerEngine` deregisters all classes specified by the monitor's calls to `AddInterest`. Once all interest has been removed, the server engine no longer calls the monitor's `Interest` method.

**Note:** When an engine monitor is freed, the associated server engine automatically removes the monitors registered interests.

See also: `AddInterest`, `RemoveInterest`, `ServerEngine`

## RemoveInterest

method

```
procedure RemoveInterest(anObjectClass : TffServerObjectClass);
```

↳ Removes the monitor's interest in the specified server object class.

Use this method to remove a monitor's interest in a specific server object class. `anObjectClass` is the class of a server-side object, as discussed in "Server object classes" on page 441. Once the interest has been removed, the server engine no longer calls the monitor's `Interest` method when it creates objects of the specified class.

The following example shows how this method may be used:

```
MyCustomMonitor.RemoveInterest(TffSrBaseCursor);
```

**Note:** When an engine monitor is freed, the associated server engine automatically removes the monitor's registered interests.

See also: `AddInterest`, `RemoveAllInterest`, `ServerEngine`

## ServerEngine

property

```
property ServerEngine : TffBaseServerEngine
```

↳ Identifies the server engine being monitored.

In order for an engine monitor to register interest in server object classes and to be notified of instances of those classes, you must connect the monitor to a server engine using the `ServerEngine` property.

If the `ServerEngine` is already specified and you set the property to a different server engine, the monitor automatically removes all interests from the previous server engine.



## TffSecurityMonitor Component

The TffSecurityMonitor component monitors the creation and use of clients, databases, and cursors within the server engine. The FlashFiler Server engine manager contains one instance of TffSecurityMonitor by default. The security monitor attaches an instance of TffSecurityExtender to each instance of TffSrClient, TffSrDatabase, and TffSrBaseCursor.

Before a client, database, or cursor performs an operation, TffSecurityMonitor compares the rights required by the operation to the user's rights. If they match, the operation is allowed to proceed otherwise the error code DBIERR\_NOTSUFFTABLERIGHTS is returned.

If you do not need the security implemented by TffSecurityMonitor and TffSecurityExtender, you are free to remove the instance of TffSecurityMonitor from the engine manager. You are also free to replace it with your own security monitor or third-party security monitor replacement.

# Hierarchy

## TComponent (VCL)

- ❶ TffComponent (FFLLBASE). . . . . 78
  - ❷ TffLoggableComponent (FFLLCOMP) . . . . . 80
    - ❸ TffStateComponent (FFLLCOMP) . . . . . 82
      - ❹ TffBaseEngineMonitor (FFLENG) . . . . . 453
        - TffSecurityMonitor (FFSRSEC)

# Properties

❷ EventLog	❹ ServerEngine
❷ EventLogEnabled	❸ State

# Methods

❹ AddInterest	❶ NewInstance	❸ ScPrepareForShutdown
❶ FreeInstance	❹ RemoveAllInterest	❸ ScShutdown
Interested	❹ RemoveInterest	❸ ScStartup
❷ LcLog	❸ ScInitialize	

# Events

❸ OnStateChange
-----------------

## Reference Section

### Interested

### method

```
function Interested(  
    aServerObject : TffObject) : TffBaseEngineExtender; override;
```

- ↳ Indicates whether the security monitor wishes to monitor the actions associated with the specified server object.

When the security monitor is attached to a server engine, it registers interest with the server engine in instances of `TffSrBaseCursor`, `TffSrDatabase`, and `TffSrClient`. When the server engine creates an instance that belongs to one of those classes, the server engine calls this method.

`aServerObject` is an instance of `TffSrBaseCursor`, `TffSrDatabase`, or `TffSrClient`. This method verifies the instance inherits from one of those classes. If it does, this method returns an instance of `TffSecurityExtender` that is attached to `aServerObject`. Otherwise, this method returns `nil`.

---

## Chapter 13: SQL Reference

This chapter is for developers using the FlashFiler TffQuery component. This chapter describes the Structured Query Language (SQL) statements and functions supported by FlashFiler. This chapter does not teach you how to use SQL. If you are not familiar with SQL, please read one of the many other available books for this purpose. For information about the TffQuery component and parameterized queries, see “Chapter 7: FlashFiler Client” on page 117.

# Syntax Conventions

FlashFiler's SQL engine is case-insensitive. You may write your SQL statements in uppercase, lowercase, or mixed-case. This applies to keywords or table or column names. Table 13.1 describes the format with which supported statements and functions are documented in this chapter.

**Table 13.1:** *SQL syntax conventions.*

Convention	Purpose
UPPERCASE	Denotes reserved keywords. Note that FlashFiler's SQL is case-insensitive. You do not have to type keywords in all uppercase.
<i>italic</i>	Identifies user-supplied parameters, such as table names, that cannot be broken into smaller units.
< <i>italic</i> >	Identifies elements that can be broken into smaller units. For example, an aggregate (i.e., function) may be broken down into the function name and a column name.
[ ]	Denotes optional syntax. Do not type the brackets.
...	Indicates that a clause within brackets may be repeated zero or more times.
	Indicates that one of two or more clauses may be used but not both. This convention occurs only within brackets and braces. Do not type the pipe character.
{ }	Indicates that one of the enclosed clauses must be specified. Do not type the braces.

---

# Naming Conventions

## Tables

Table names must conform to the following requirements:

- They may be no more than 31 characters in length.
- The valid character set for table names is '0'..'9', 'A'..'Z', 'a'..'z', and underscore(\_). Table names may not contain spaces or any other characters outside the aforementioned range.
- The first character in a table name may be any one of the valid characters.
- The table name may not contain a path or file name.

When selecting from a table, you may create an alias for the table as in the following example:

```
SELECT P.Name, I.QtyOnHand FROM Product AS P, Inventory AS I
WHERE I.ProductID = P.ID AND I.QtyOnHand > 0
```

This example creates an alias called P for the product table and an alias called I for the inventory table and then refers to these aliases in the rest of the SQL statement.

## Columns

A column corresponds to a field within a table. Column names may be of any length and may contain numbers, digits, space, and international characters. Punctuation characters are not allowed. When referencing a multiple word column name, surround the column name with double quotes. For example:

```
SELECT OEM, "Average Price", "Last Price"
FROM ProductHistory
```



# Data Types

Table 13.2 lists the FlashFiler data types supported by FlashFiler's SQL engine. The table indicates whether the type is selectable and whether a field of that data type may be used to sort a query's result set. FlashFiler does not support the CAST or CONVERT functions.

**Table 13.2:** *SQL data types supported by FlashFiler.*

Data Type	Can Select?	Can Order By?
fftBoolean	Yes	Yes
fftChar	Yes	Yes
fftWideChar	Yes	Yes
fftByte	Yes	Yes
fftWord16	Yes	Yes
fftWord32	Yes	Yes
fftInt8	Yes	Yes
fftInt16	Yes	Yes
fftInt32	Yes	Yes
fftAutoInc	Yes	Yes
fftSingle	Yes	Yes
fftDouble	Yes	Yes
fftExtended	Yes	Yes
fftComp	Yes	Yes
fftCurrency	Yes	Yes
fftStDate	Yes	Yes
fftStTime	Yes	Yes
fftDateTime	Yes	Yes
fftBLOB	Yes	No
fftBLOBMemo	Yes	No
fftBLOBFmtMemo	Yes	No
fftBLOBOLEObj	Yes	No
fftBLOBGraphic	Yes	No
fftBLOBDBSOLEObj	Yes	No
fftBLOBTypedBin	Yes	No

**Table 13.2:** *SQL data types supported by FlashFiler. (continued)*

Data Type	Can Select?	Can Order By?
fftBLOBFile	Yes	No
fftByteArray	Yes	No
fftShortString	Yes	Yes
fftShortAnsiStr	Yes	Yes
fftNullString	Yes	Yes
fftNullAnsiStr	Yes	Yes
fftWideString	Yes	Yes

**Note:** Selecting any BLOB data type returns a reference number for the BLOB, not the BLOB content. To access the contents of a BLOB, you must create an instance of TffBLOBStream. See TffBLOBStream.Create for more details.

## Supported Statements

FlashFiler SQL engine supports the SELECT SQL statement. FlashFiler does not support other SQL verbs such as CREATE, CONNECT, UPDATE, and DELETE. For syntax and other information, see “Syntax Definitions” on page 472.

# Supported Functions

This section summarizes the various functions supported by FlashFiler's SQL engine. For syntax and other information, see "Syntax Definitions" on page 472.

## Aggregate functions

Table 13.3 lists the aggregates supported by the FlashFiler.

**Table 13.3:** *Aggregate functions.*

Function	Purpose
AVG	Calculate the average of the values in the column.
COUNT	Count the number of rows returned by a query excluding those rows with NULL values.
COUNT(*)	Count all rows in a table including duplicates and those rows with NULL values.
MAX	Find the maximum value of a column in a result set.
MIN	Find the minimum value of a column in a result set.
SUM	Add all scalar values returned for a column in a result set.

## Conversion functions

Table 13.4 lists the conversion functions supported by FlashFiler.

**Table 13.4:** *Conversion functions.*

Function	Purpose
LOWER	Convert all characters to lowercase.
UPPER	Convert all characters to uppercase.

# Date and time functions

Table 13.5 lists the date and time functions supported by FlashFiler. All references to the word server in this section refer to the computer hosting the FlashFiler Server.

**Table 13.5:** *Date and time functions.*

Function	Purpose
CURRENT_DATE	Return the current date of the server at the start of the query.
CURRENT_TIME	Return the current time of the server at the start of the query.
CURRENT_TIMESTAMP	Return the current date and time of the server at the start of the query.
EXTRACT	Extract an individual date or time value from a specified date, time, or datetime.

# String functions

Table 13.6 lists the string functions supported by FlashFiler.

**Table 13.6:** *String functions.*

Function	Purpose
CHARACTER_LENGTH, CHAR_LENGTH	Return the number of characters in a column value.
POSITION	Return the current position of a substring within a string.
SUBSTRING	Extract a substring out of a string.
TRIM	Remove a leading and/or trailing character from a string.

# Other functions

Table 13.7 lists the functions that do not fall into any of the previous categories.

**Table 13.7:** *Miscellaneous functions.*

Function	Purpose
COALESCE	Return NULL if all its operands evaluate to NULL otherwise returns the first (left-most) non-NULL operand.
NULLIF	Return NULL if its operands are equal otherwise returns the first operand.

# Key Words

Table 13.8 lists the words recognized by FlashFiler's SQL engine. Do not use these words for table names, column names, etc. in your SQL syntax.

**Table 13.8:** *SQL reserved words.*

ALL	DATE	LEADING	SOME
AND	DAY	LIKE	SUBSTRING
ANY	DESC	LOWER	SUM
AS	DISTINCT	MATCH	SYSTEM_USER
ASC	ELSE	MAX	THEN
AVG	END	MIN	TIME
BETWEEN	EXISTS	MINUTE	TIMESTAMP
BOTH	EXTRACT	MONTH	TO
BY	FALSE	NOT	TRAILING
CASE	FOR	NULL	TRIM
CHAR_LENGTH	FROM	NULLIF	TRUE
CHARACTER_LENGTH	FULL	OR	UNIQUE
COALESCE	GROUP	ORDER	UNKNOWN
COUNT	HAVING	PARTIAL	UPPER
CURRENT_DATE	HOURL	POSITION	USER
CURRENT_TIME	IN	SECOND	WHEN
CURRENT_TIMESTAMP	INTERVAL	SELECTS	WHERE
CURRENT_USER	IS	SESSION_USER	

# Syntax Definitions

**AVG**

**function**

---

```
AVG([ALL | DISTINCT] <simple expr>)
```

↪ Calculates the average of numeric values in a column or expression.

Use this aggregate function to calculate the average of the values for a specific column or expression. All values in the column or expression must be numeric. By default, all column values are included in the calculation. Use the ALL keyword to explicitly enforce this behavior. Use the DISTINCT keyword to eliminate duplicate values before calculating the average.

NULL values are excluded from the calculation. If a query does not return any rows, the result of AVG is NULL. The column name for the average value in the result set is the same as the AVG expression.

The following example calculates the average salary of all employees:

```
SELECT AVG(salary) FROM Employee
```

See also: COUNT, MIN, MAX, SUM

**CHARACTER\_LENGTH, CHAR\_LENGTH**

**function**

---

```
CHARACTER_LENGTH(<simple expr>)
```

```
CHAR_LENGTH(<simple expr>)
```

↪ Calculates the number of characters returned by a column or simple expression.

Use this string function to calculate the number of characters in a string value returned from a column or simple expression. If the column or result of the simple expression is not a string then the SQL statement will not be executed.

The following example finds all employees whose last name is longer than twenty characters:

```
SELECT ID, FirstName, LastName FROM Contact
WHERE CHAR_LENGTH(LastName) > 20
```

See also: POSITION, SUBSTRING, TRIM

```
COALESCE(<simple expr1> [, <simple exprN> ...])
```

↳ Returns NULL if each of its expressions evaluates to NULL.

This is a shorthand version of the following CASE statement:

```
CASE WHEN a IS NOT NULL THEN a ELSE NULL END
```

Parameters <simple expr1> through <simple exprN> may each be a column, constant, expression, or function. If all parameters evaluate to NULL then this function returns NULL otherwise this function returns <simple expr1>.

See also: NULLIF

## COUNT

## function

```
COUNT({* | [ALL | DISTINCT] <simple expr>})
```

↳ Returns the number of rows that satisfies a query's search condition.

Use this aggregate function to count the number of rows satisfying a query's WHERE clause. If you specify \* then all rows within the result set are counted including those rows with NULL values. The following example counts the number of employees having more than 10 vacation days to take before the end of the year:

```
SELECT COUNT(*) FROM Employee  
WHERE VacationDays > 10
```

If you specify a column name or simple expression, COUNT will include all non-NULL rows by default. This is also the behavior enforced by the ALL keyword. You do not need to specify the ALL keyword. The following example returns the number of employees without any sick days:

```
SELECT COUNT(EmployeeID) FROM Employee  
WHERE SickDays = 0
```

If you specify the DISTINCT keyword, COUNT eliminates all duplicate rows. It does not include NULL values. The following example returns the number of animal types that have been tagged within the nature preserve:

```
SELECT COUNT(DISTINCT AnimalTypeID) FROM TagHistory
```

The column name for the count in the result set is the same as the count expression.

See also: AVG, MAX, MIN, SUM



---

**CURRENT\_DATE**

↳ Returns the current date of the FlashFiler Server.

Use this function to obtain the current date of the computer hosting the FlashFiler Server. This function returns a value of type TDateTime. The time portion is set to zero (i.e., midnight). All references to CURRENT\_DATE within the same statement are guaranteed to have the same value.

The following example finds all orders entered today, with the assumption that the order's Entered field was set to CURRENT\_DATE when the order was originally entered:

```
SELECT OrderID, Status FROM Orders
WHERE Entered = CURRENT_DATE
```

See also: CURRENT\_TIME, CURRENT\_TIMESTAMP, EXTRACT

---

**CURRENT\_TIME**

↳ Returns the current time of the FlashFiler Server.

Use this function to obtain the current time of the computer hosting the FlashFiler Server. This function returns a value of type TDateTime. The date portion is set to zero. All references to CURRENT\_TIME within the same statement are guaranteed to have the same time.

The following example finds all unshipped orders scheduled for shipment before the current time, with the assumption that the order's ScheduledShipTime contains only a time value.

```
SELECT OrderID FROM Orders
WHERE (ScheduledShipTime < CURRENT_TIME) AND
      (NOT Shipped)
```

See also: CURRENT\_DATE, CURRENT\_TIMESTAMP, EXTRACT

CURRENT\_TIMESTAMP

↳ Returns the current date and time of the FlashFiler Server.

Use this function to obtain the current date and time of the computer hosting the FlashFiler Server. This function returns a value of type TDateTime. All references to CURRENT\_TIMESTAMP within the same SQL statement are guaranteed to have the same value.

The following example selects all purchase orders that were sent to the vendor today:

```
SELECT ID, VendorID FROM PurchaseOrders
WHERE Sent BETWEEN CURRENT_DATE AND CURRENT_TIMESTAMP
```

See also: CURRENT\_DATE, CURRENT\_TIME, EXTRACT

---

**EXTRACT****function**

EXTRACT({YEAR | MONTH | DAY | HOUR | MINUTE | SECOND}  
FROM <simple expr>)

↳ Returns the specified portion of a datetime field or an expression returning a TDateTime value.

Use this function to extract the year, month, day, hour, minute, or second from a datetime field or expression returning a datetime value. The following example selects the number of customers placing orders within each month of a specific year:

```
SELECT EXTRACT(MONTH FROM Entered),
COUNT(DISTINCT CustomerID) FROM Orders
WHERE EXTRACT(YEAR FROM Entered) = :Year
GROUP BY EXTRACT(MONTH FROM Entered)
```

See also: CURRENT\_DATE, CURRENT\_TIME, CURRENT\_TIMESTAMP

---

**LOWER****function**

LOWER(<simple expr>)

↳ Converts a string to all lowercase.

Use this function to convert all characters in a string to lowercase. <simple expr> is a column, constant, expression, or function that returns a string type. The following example selects all OEMs from Inventory, fetching the OEM numbers in lowercase:

```
SELECT LOWER(OEMNum), QtyOnHand FROM Inventory
```

See also: UPPER

---

```
MAX([ALL | DISTINCT] <simple expr>)
```

↳ Retrieves the largest value in a column or expression.

Use this aggregate function to determine the largest value in a column or expression. <simple expr> may be a constant, column, expression, or non-aggregate function. If you specify the ALL keyword, MAX considers all non-NULL values. However, this is the default behavior of MAX so ALL is optional. If you specify the DISTINCT keyword, MAX eliminates all duplicate rows before finding the largest value. The column name for the maximum in the result set is the same as the MAX expression.

This function excludes rows having NULL values. If a query does not return any rows, this function returns NULL. The following example selects the largest donor contribution and the average donor contribution:

```
SELECT MAX(Donation), AVG(Donation) FROM Contributions
```

See also: AVG, COUNT, MIN, SUM

---

```
MIN([ALL | DISTINCT] <simple expr>)
```

↳ Retrieves the smallest value in a column or expression.

Use this aggregate function to determine the smallest value in a column or expression. <simple expr> may be a constant, column, expression, or non-aggregate function. If you specify the ALL keyword, MIN considers all non-NULL values. However, this is the default behavior of MIN so you need not specify the ALL keyword. If you specify the DISTINCT keyword, MIN eliminates all duplicate rows before finding the smallest value. The column name for the minimum in the result set is the same as the MIN expression.

This function excludes rows having NULL values. If a query does not contain any rows, this function returns NULL. The following example selects the smallest budget amount ever given to a specific department:

```
SELECT MIN(Budget) FROM Department
WHERE DeptID = :aDepartment
```

See also: AVG, COUNT, MAX, SUM

---

```
NULLIF(<simple expr1>, <simple expr2>)
```

↳ Returns NULL if its operands are equal.

This is a shorthand version of the following CASE statement:

```
CASE WHEN a = b THEN NULL ELSE a END
```

<simple expr1> and <simple expr2> may each be a column, constant, expression, or function returning any type. If <simple expr1> and <simple expr2> are equivalent then this function returns NULL otherwise it returns <simple expr1>.

See also: COALESCE

---

```
POSITION(<simple expr1>, <simple expr2>)
```

↳ Returns the position within string <simple expr1> of substring <simple expr2>.

Use this function to find the position of a substring within another substring. <simple expr1> and <simple expr2> may each be a constant, column, expression, or function that returns a string.

This function is case-sensitive. If the substring is not found within the string, a zero is returned. Otherwise, this function returns the offset of the first character of <simple expr2> within <simple expr1>. The following example finds all parts whose OEM number begins with "HFK":

```
SELECT OEMNum, VendorID FROM Items  
WHERE Position(OEMNum, "HFK") = 1
```

See also: CHAR\_LENGTH, SUBSTRING, TRIM

```

SELECT [ALL | DISTINCT] { * | <selection list> }
    FROM <table list>
    [WHERE <cond expr>]
    [GROUP BY <column list>]
    [HAVING <cond expr>]
    [ORDER BY <order list>][:]

```

↪ Retrieves data from one or more tables within a database.

Use this statement to select a set of data from one or more tables within a database. You may not select data from two or more tables residing in different databases. This statement returns a read-only set.

**Note:** In the following grammar definitions, all occurrences of left and right parentheses means the left or right parenthesis is required. For example, the definition of the EXISTS clause is EXISTS ( <select> ) meaning that you must literally type EXISTS followed by a left parenthesis, followed by a valid SELECT statement, followed by a right parenthesis.

Literal characters in this grammar definition are shown without being quoted, unless the character itself is used as a grammar metacharacter. In that case, the character will be quoted with double quotes, for example “|”.

```

<select> = SELECT [ALL | DISTINCT] { * | <selection list> }
    FROM <table list>
    [WHERE <cond expr>]
    [GROUP BY <column list>]
    [HAVING <cond expr>]
    [ORDER BY <order list>]

```

```

<selection list> = <selection> [, <selection> ...]

```

```

<selection> = <simple expr> [[AS] <column>]

```

```

<column> = <ident>

```

```

<ident> = <letter> [{ <letter> | <digit> } ...]

```

```

<simple expr> = <term> [{ + | - | " | " } <term> ...]

```

```

<term> = <factor> [{ * | / } <factor> ...]

```

```

<factor> = [-] { ( { <cond expr> | <select> } )
    | <field ref>
    | <literal>
    | <param>
    | <aggregate>
    | <scalar func> }

```

```

<cond expr> = <cond term> [OR <cond term>]
<cond term> = <cond factor> [AND <cond factor>]
<cond factor> = [NOT] <cond primary>
<cond primary> = { <exists> | <unique>
                  | <simple expr>
                  | { = | <= | < | > | >= | <> }
                    { <allOrAny> | <simple expr> }
                    | <between> | <like> | <in>
                    | <match>
                    | NOT { <between> | <like> | <in> }
                    | <isTest>
                  }
<exists> = EXISTS ( <select> )
<unique> = UNIQUE ( <select> )
<allOrAny> = { ALL | ANY | SOME } ( <select> )
<between> = BETWEEN <simple expr> AND <simple expr>
<like> = LIKE <simple expr>
<in> = IN ( { <simple expr list> | <select> } )
<simple expr list> = <simple expr> [, <simple expr> ...]
<match> = MATCH [UNIQUE] [{ PARTIAL | FULL }] ( <select> )
<isTest> = IS [NOT] { NULL | TRUE | FALSE | UNKNOWN }
<field ref> = { <ident>
               | " { <ident> | <integer literal> }
                 [{ <ident> | <integer literal> } ...] "
               | . { <ident>
                   | " { <ident> | <integer literal> }
                     [{ <ident> | <integer literal> } ...] "
                   | * }
               ]
<literal> = { <float literal> | <integer literal>
            | <string literal> | <date literal>
            | <time literal> | <timestamp literal>
            | <interval literal> }
<float literal> = <digit> [<digit> ...] . <digit> [<digit> ...]
<integer literal> = <digit> [<digit> ...]

```

**Note:** In *<string literal>*, the double quotes are two single quotes, one after the other. They're used to indicate an single quote within a string.

```

<string literal> = ' [{ <noQuote> | '' } ...] '
<noQuote> = <any character other than single quote and end of line>
<date literal> = DATE <string literal>
<time literal> = TIME <string literal>
<timestamp literal> = TIMESTAMP <string literal>
<interval literal> = INTERVAL <string literal>
                        { YEAR | MONTH | DAY |
                          HOUR | MINUTE | SECOND }
                        [TO { YEAR | MONTH | DAY |
                             HOUR | MINUTE | SECOND}]

<param> = : <ident>

<aggregate> = { COUNT ( { * | [ALL | DISTINCT] <simple expr> } )
                | { MIN | MAX | SUM | AVG }
                ( [ALL | DISTINCT] <simple expr> )
              }

<scalar func> = { CASE <case expr>
                  | { CHARACTER_LENGTH | CHAR_LENGTH }
                    ( <simple expr> )
                  | COALESCE <coalesce expr>
                  | CURRENT_DATE
                  | CURRENT_TIME
                  | CURRENT_TIMESTAMP
                  | LOWER ( <simple expr> )
                  | UPPER ( <simple expr> )
                  | POSITION ( <simple expr> , <simple expr> )
                  | SUBSTRING ( <simple expr>
                                FROM <simple expr>
                                [FOR <simple expr>] )
                  | TRIM ( [LEADING | TRAILING | BOTH]
                           <simple expr>
                           [FROM <simple expr>] )
                  | EXTRACT ( { YEAR | MONTH | DAY
                                HOUR | MINUTE | SECOND }
                              FROM <simple expr> )
                  | NULLIF ( <simple expr> , <simple expr> )
                }

```

```

<case expr> = <when list> [ELSE { NULL | <simple expr> }] END
<when list> = <when> [<when> ...]
<when> = WHEN <cond expr> THEN { NULL | <simple expr> }
<coalesce expr> = ( <simple expr> [, <simple expr> ...] )
<table list> = <table ref> [, <table ref> ...]
<table ref> = { <ident>
                | " { <ident> | <integer literal> }
                  [{ <ident> | <integer literal> } ...] " }
                [[AS] <ident>]
<column list> = <column> [, <column> ...]
<order list> = <order item> [, <order item> ...]
<order item> = { <column> | <integer literal> } [ASC | DESC]

```

A SELECT statement makes it possible to retrieve rows of data from two or more tables. The SELECT grammar is very complex and may contain many clauses. Only the SELECT and FROM clauses are required. The following table describes the purpose of possible clauses in an SQL SELECT statement:

Clause	Purpose
SELECT	Specifies the columns that are to be retrieved. Simply stating * retrieves all columns.
FROM	Specifies the tables from which the columns are to be retrieved. You may assign an alias to each table and use the alias in the other clauses of the SELECT statement.
WHERE	Specifies the criteria for record selection.
GROUP BY	Indicates how rows are to be grouped. Rows are grouped together based upon the values in one or more columns.
HAVING	Allows you to restrict the rows that are grouped based upon the values in the group. This is similar to a WHERE clause but applies to the GROUP BY.
ORDER BY	Indicates how the result set is to be ordered. You may order the result set using one or more columns. For each column, you may specify whether the column is sorted ascending or descending



FlashFiler's SQL engine does not support the following in its SELECT grammar:

bit strings	CONVERT	OVERLAPS
CAST	explicit JOINs	time zones
COLLATE	OCTET_LENGTH	UNION

The following example selects all authors:

```
SELECT * FROM Author
```

The following example selects all authors who live in Indiana:

```
SELECT * FROM Author WHERE State = 'IN'
```

The following example selects the book categories published by each author:

```
SELECT DISTINCT AuthID, Category FROM Books  
ORDER BY AuthID
```

The following example is similar to the previous except that it now includes the author's name:

```
SELECT DISTINCT A.Name, A.AuthID, B.Category  
FROM Author A, Books B  
WHERE B.AuthID = A.AuthID  
ORDER By A.AuthID
```

The following example lists all book categories having more than one book in that category:

```
SELECT Category, Count(*) FROM Books  
GROUP BY Category HAVING Count(*) > 1
```

## SUBSTRING

function

```
SUBSTRING <simple expr1> FROM <simple expr2>  
[FOR <simple expr3>]
```

13

↳ Extracts a substring from a string.

Use this string function to extract a substring from a string. <simple expr1> is the string from which the substring is extracted. <simple expr1> may be a column, constant, expression, or function returning a string. <simple expr2> and <simple expr3> may each be a column, constant, or function returning a numeric value. <simple expr2> is base 1 and is the position at which the extraction begins. <simple expr3> is the number of characters to extract.

If the FOR keyword and <simple expr3> are omitted, the returned substring consists of all characters in <simple expr1> starting with <simple expr2>.

The following example extracts the first three characters from a part number:

```
SELECT SUBSTRING(PartNum FROM 1 FOR 3) FROM Product
```

The following example extracts the last name of a customer:

```
SELECT SUBSTRING(Name FROM POSITION(Name, ' '))  
FROM Customer
```

See also: CHAR\_LENGTH, POSITION, TRIM

## SUM

## function

---

```
SUM([ALL | DISTINCT] <simple expr>)
```

↪ Totals the numeric values in a column or simple expression.

Use SUM to add up one or more numeric values. <simple expr> may be a constant, column, expression, or function that returns a numeric value. If you specify the ALL keyword, SUM totals all non-NULL values. Since this is the default behavior of SUM, you do not need to specify ALL. If you specify the DISTINCT keyword, SUM eliminates duplicate rows before summing the values. The column name for the sum in the result set is the same as the sum expression.

This function ignores NULL values. If the query does not contain any rows, this function returns NULL. The following example totals the salary of all employees within a specific company:

```
SELECT SUM(Salary) FROM Employee WHERE Company = :anID
```

See also: AVG, COUNT, MAX, MIN

```
TRIM([LEADING | TRAILING | BOTH] <simple expr1>
     [FROM <simple expr 2>])
```

↳ Removes leading and/or trailing characters from a string.

Use this function to remove characters from the beginning and/or end of a string. <simple expr1> and <simple expr2> may each be a constant, column, expression, or function that returns a string.

<simple expr1> is the character that is to be trimmed from <simple expr2>. If <simple expr1> is omitted, the trim character is assumed to be a space. If <simple expr1> has a length greater than one then the first character is used as the removal character.

By default, all leading and trailing occurrences of <simple expr1> are removed from <simple expr2>. This is also the behavior of the BOTH keyword. If a trim keyword is not specified and the FROM keyword is not specified then <simple expr1> is assumed to be the string that is to be trimmed and spaces are trimmed from the beginning and ending of the string.

If you specify the LEADING keyword, only leading instances of <simple expr1> are removed from <simple expr2>. If you specify the TRAILING keyword, only trailing instances of <simple expr1> are removed from <simple expr2>.

The following example returns product descriptions minus trailing spaces:

```
SELECT TRIM(TRAILING ' ' FROM Description) FROM Products
```

The following example trims spaces from the beginning and end of employee names:

```
SELECT EmpID, TRIM(Name), HireDate FROM Employee
```

See also: CHAR\_LENGTH, POSITION, SUBSTRING

```
UPPER(<simple expr>)
```

↳ Converts all characters in a string to uppercase.

Use this function to convert all characters in a string to uppercase. <simple expr> is a column, constant, expression, or function that returns a string type. The following example selects all contacts whose last name matches a specific last name:

```
SELECT * FROM Contact WHERE UPPER(LastName) = :LastName
```

See also: LOWER

---

## Chapter 14: Troubleshooting and Performance

Client/server database systems come with their own unique set of considerations. Being familiar with these considerations can help you through the trouble spots and help your applications meet or exceed your performance needs. The two major sections of this chapter explore some of these considerations. The first section, Troubleshooting, focuses on problems users sometimes encounter and the Performance section looks at ways to increase performance.

---

# Troubleshooting

This section provides advice and suggestions for resolving FlashFiler related issues. We've grouped items in this section into categories to help focus on specific areas. The categories and the items within them are alphabetized. Items in quotations indicate specific error messages. If what you're looking for isn't in a specific category, look around, it may be covered in another category.

Our goal for this section is to include common problems you may encounter. Not all situations are covered here. If your situation isn't mentioned here, you can refer to other relevant sections of this manual, search on-line help, or contact us via one of the Technical Support options.

## Compatibility with other components

Generally, FlashFiler is compatible with all components that work with TDataSet descendants. The known exceptions and workarounds are listed in this section.

### Data-aware grids

When using data-aware grids, such as TDBGrid, the vertical scroll bar is not proportionate with the current view of the dataset. Most BDE replacements have this same issue. It's not a bug, but a design issue. We plan on supporting this feature in the future. We'll announce when this feature is available in the FlashFiler newsgroup and through the FlashFiler mailing list.

### ExpressQuantumGrid

When using FlashFiler with Developer's Express ExpressQuantumGrid, you may encounter a problem where your application hangs when using the `egoMultiSelect` option. To work around this problem, set the grid's `UseBookMarks` property to `False`. You also need to enable the `egoLoadAllRecords` option or set the `PartialLoad` property to `True`. As specified in the `SelectRows` topic of the `TCustomdxDBGrid` class, you must use the `SelectNodes` property instead of the `SelectRows` property when trying to access the selected records.

### InfoPower

InfoPower 2000 can be used with FlashFiler as is.

If you own InfoPower 3.x you must also install a special InfoPower-capable FlashFiler table component in Delphi 3. To install this component, click the `Add` button in the `Install Packages` dialog, and select the `F200ID30.DPL` file from FlashFiler's root directory. This will add another component to your FlashFiler tab on the palette called `TffwwTable` (the bitmap has a small IP in the top right hand corner). This package requires `IP30.DPL`, so don't bother doing this if you don't own InfoPower 3.x (note that you must have version 3.01 or later).

If you own InfoPower 4.x you can install a special InfoPower-capable FlashFiler table component in Delphi 3, 4 and C++Builder 3 or 4. To install this component, click the Add button in the Install Packages dialog, and select the F200JDvv.DPL file from your root directory (where vv=30 for Delphi 3, 40 for Delphi 4, 35 for C++Builder 3, and 41 for C++Builder 4). This will add another component to your FlashFiler tab on the palette called TffwwTable (the bitmap has a small IP in the top right hand corner). This package requires IP40XXX.DPL, so don't bother doing this if you don't own InfoPower 4.

Once the InfoPower compatible table has been installed, the InfoPower components can use a TffwwTable through a TwwwDataSource.

TwwwSearchDlg provided with InfoPower 3 and 4 are not compatible with FlashFiler.

### Other standard components

Because of TBatchMove, TDBLookupList, and TDBLookupCombo dependencies on internal parts of the BDE, FlashFiler is not compatible with these components.

## Connectivity

One of the most common problems with developing FlashFiler applications is getting the client and server applications to communicate. Sometimes users have trouble establishing a connection between the client and server applications and other times the connection drops after it's established. Both situations are covered in the following sections.

### Cannot locate or connect to network server.

This message indicates your FlashFiler client application can't find a FlashFiler Server. Ensure the FlashFiler Server is running and the client is using a transport that is enabled on the server.

Try to ping the computer hosting the server from the client computer. If the ping does not receive a response, there is a network or NIC configuration issue.

Networks running across the Internet can also cause a problem since Internet routers typically don't propagate UDP packets. You can resolve this problem by using the FFComms utility (in the FFComms directory) to set the server's address in the registry.

Firewalls can prevent a server from connecting with a client. Clients looking for a server open a port in listen mode and send a message to the server indicating which port it's listening to. If the client is behind a firewall, the server may not be able to respond to the client's message. The client raises an error when it doesn't hear from the server.

If you're using a secure server and a client has three attempts to log on, it will not allow the client to connect after three failed attempts.

## Client loses connection

Clients can lose their connections if the server's Keep Alive settings are set too low. When there is a lack of activity between a client and server for a length of time, the server assumes the client is gone and drops the connection with the client. The Keep Alive settings determine how long the server waits before disconnecting the client.

Lengthy operations, like packing a table, take more time than normal inserting, deleting, and navigating records in a table. Since there isn't any communication during these types of operations, you may have to increase the Keep Alive settings to prevent dropping of the connection.

Heavy operations on the client can also cause the client to lose its connection. When the client performs operations in a tight loop, it's not able to reply to the server's Keep Alive messages. Calling `Application.ProcessMessages` in these types of loops prevents the client from losing its connection in these operations.

See “Operation” on page 33 for more information about setting the Keep Alive options and other server settings.

## Default transport

Once you set a transport in FlashFiler Explorer, it becomes the default transport for all FlashFiler clients on that system. This can cause problems if your application is expecting a different transport. The problem is that FlashFiler stores its default client transport in a single registry key and FlashFiler client applications look to this key for their default transport.

The workaround is to use explicit `TffRemoteServerEngine`, `TffClient`, and `TffLegacyTransport` components in your client application. Set the `TffLegacyTransport`'s `Protocol` property and the `TffClient`'s `IsDefault` property to `True`. Ensure these components are created before any `TffDatabase` or `TffTable` components.

## Table cannot open its database : the database name is not set or doesn't exist [no error code]

This error is raised when the FlashFiler Server doesn't recognize the `DatabaseName` property of a `TffTable` or `TffDatabase`.

If you're not using the FlashFiler Server from the default location, or you're using a singleEXE application, you may need to copy your server tables to the location where the server is being executed. The “SingleEXE Applications” section on page 570 has more information about singleEXE applications and Chapter 3, page 31, has more information about server tables.

If you're using a path for the `TffDatabase`'s `AliasName` property, ensure it is a valid path. Also, ensure `SingeEXE` is not defined if it shouldn't be.

## Crystal reports

The “Crystal Reports Support” section of the Appendix on page 601 describes how to use FlashFiler with Crystal Reports. The important step is renaming the BDE driver (P2BBDE.DLL) so that its name is alphabetically after the FlashFiler driver (P2BFFxxx.DLL). As described in that section, the drivers are checked alphabetically and the BDE driver incorrectly reports that it’s the correct driver for FlashFiler tables.

**Note:** FlashFiler supports 32-bit Crystal Reports versions 4.5 – 7. Crystal Reports 8 and later are not supported.

## Design-time errors

Design-time errors are errors that appear while working in the IDE or while your application is compiling. One thing some users aren’t aware of is that the IDE has an implicit FlashFiler client once the design-time packages are installed. This means that when using the implicit IDE client, you should start your FlashFiler Server before working on a FlashFiler project.

It’s possible to use an explicit IDE client. To do so, place a TffClient component in your application and set its IsDefault property to True.

### FlashFiler: Cannot locate or connect to network server

At design time, this error indicates the IDE can’t find a FlashFiler Server. As previously explained, the IDE has an implicit FlashFiler client that requires a FlashFiler Server when developing a FlashFiler project.

When using the implicit client, if the FlashFiler Server is stopped while working in the IDE, it will not reconnect when the server is restarted. To reconnect the IDE’s implicit client to the server you need to exit the IDE and restart it.

### Index name with spaces

To use an index name that contains spaces, surround the name with brackets as in the following example:

```
MyTable.IndexName := [My Index Name];
```

## Error messages

For the most part, the FlashFiler components present meaningful error messages when things go awry. Since it’s hard to put all relevant information into each error message, we’ve put some of the more common error messages in this Troubleshooting section. You can also look in your FFSERVER.LOG file (located in the same location as your FlashFiler Server or SingleEXE application) for detailed information about the error. If none of the error information helps resolve your problem, please contact Technical Support.



## Installation

Installation should happen automatically as described in “Installation” on page 12 of Chapter 1. If the installation doesn’t occur automatically, the Installation section includes instructions on how to manually install the FlashFiler components into your IDE(s). If you aren’t able to install FlashFiler using the procedures in Chapter 1, please contact Technical Support for further assistance.

### FlashFiler palettes

You should have two FlashFiler palettes in your IDE: FlashFiler Server and FlashFiler Client. If you only have a single palette named FlashFiler, you can either use it as is or take the following steps to separate the two palettes:

1. Remove the FlashFiler package. (Component | Install Package, highlight FlashFiler2 Designtime Package and press the remove button)
2. Ensure the FlashFiler palette is removed. (Tools | Environment Options | Palette)
3. Close your IDE.
4. Delete `\HKEY_CURRENT_USER\Software\Borland\<your IDE>\<your IDE version number>\Palette\FlashFiler.HiddenPage`.

### SingleEXE

Developing a singleEXE application can be a challenge. If you’ve followed the directions and recommendations in the Appendix section “SingleEXE Applications” on page 570, then it should be straight forward. If things still aren’t working properly, this section should help you narrow the problem.

## Run-time errors

Run-time errors are problems that arise once your program has started executing.

One thing that users often don’t realize is that singleEXE applications contain a fully operational FlashFiler Server once they are executed. The only difference between the singleEXE server and the FlashFiler Server application is that the singleEXE server doesn’t have a graphical interface.

**FlashFiler: FF SERVER ERROR: File could not be opened [\$3C01/15361].**

This error message means another application has the file opened. FlashFiler tries to open its files (tables) in exclusive mode. If another application has the table opened, FlashFiler is unable to open it and raises this error.

This problem can also arise if the FlashFiler Server doesn’t have proper operating system rights or permissions to open the file.

## Why can't I open any of the tables in my singleEXE application?

This problem is often due to multiple servers attempting to access the same tables. In other words, the IDE is connected to a separate stand-alone server and has various tables opened in various forms at design time (so that you can see the data). These tables are the same ones that your singleEXE application is trying to open. This is not possible. Our recommendation is to design, code, and test your application with a stand-alone server (i.e., do not use singleEXE mode). This means that the compiler and your application can both connect to the same server and open the same tables. Once everything in your application is working fine, turn on the singleEXE option and compile the application again. Terminate the compiler and the stand-alone server and then run the application (or, conversely, copy the singleEXE application over to a test machine - together with any tables it requires - and run it there). If you wish to debug a singleEXE application then the best way to do this would be to design the application with all of the tables' Active properties set to False (i.e., you cannot see any data at design time). This means that the IDE is not really using the stand-alone server and so can stand having the server shut down. You can then debug your application with the IDE's debugger.

Another difficulty with singleEXE applications is that any stand-alone server that is running in single-user mode on the same machine may act as the server for the client in a singleEXE application. In single user mode (which a singleEXE application is forced to be), the client finds a server by sending a special message to all of the tompmost windows on your system. The window that replies with the correct answer is the server that the client will then use. This happens even in singleEXE mode and it could mean that your singleEXE application is using a stand-alone server (and one that maybe doesn't have the aliases you require, etc.). The answer is to terminate the stand-alone server application.

## FlashFiler Server is too slow

FlashFiler is a very fast database engine, especially for a client/server database engine. If you notice it being slow there is probably something wrong.

The most common thing that causes slow performance is the debug log. Ensure the debug log is turned off during normal operations. Look to the Performance section on page 493 for more information.

## Invalid field values when posting new record

This can appear to happen when a range is set and the new record doesn't fall within the range. Since the new record isn't in the range, it doesn't become the active record when posted and you don't have access to it's field values. To resolve these types of errors you may want to cancel your range before inserting new records.

## SysTools date field returns invalid date

After setting a field as a SysTools Date data type, it should always be treated as if it were a TDateTime. Retrieve the date using the AsDateTime property of the TField class. The main difference between DateTime and SysTools Date fields is the range of dates available and the number of bytes required to store each type. See “FlashFiler Data Types” on page 524 for more information.

## TffTable.RecNo always returns -1

FlashFiler does not support the TDataSet.RecNo property at this time. This is another design issue and is related to the issue with data-aware grids. We plan to add support for this feature in the future. We will announce the addition of this feature in the FlashFiler newsgroup and via the FlashFiler newsletter once available.

## Searching

### Using ranges with null fields

Any records containing null fields will not be included in a range that uses an index containing that field. If you need null values included in a range, you will need to initialize them to an empty value. This initialization can be automatically accomplished using default field values.

---

# Performance

Users moving from the Borland Database Engine to FlashFiler may notice some database operations are slower. Regardless of how you use FlashFiler, whether it's in singleEXE, singleUser, or network mode, at its roots FlashFiler is a client/server database engine. Developing a client/server database application requires a basic knowledge of how the back-end server works and attention to details that might be overlooked in traditional Paradox, dBase, or Microsoft Access style databases.

In this section we focus on those details and FlashFiler Server basics that you need to know to write high performance FlashFiler applications.

## Reduce messages between client and server

One potential bottleneck with any client/server database is the time it takes to send requests to the server and replies back to the client. For performance, you should minimize this traffic. This section contains suggestions to help you minimize message traffic.

### Embed the server engine

If multi-user support is not required, place a TffServerEngine component in the application. This allows the FlashFiler Client to make direct calls into the server engine instead of going through a communications layer. Note that a TffDatabase, TffSession, and TffClient are also required. The following lines of source code illustrate how the components are connected together in order to access the server engine:

```
aServerEngine := TffServerEngine.Create(nil);  
aClient := TffClient.Create(nil);  
aClient.ClientName := 'FFClient';  
aClient.ServerEngine := aServerEngine;  
  
aSession := TffSession.Create(nil);  
aSession.SessionName := 'FFSess';  
aSession.ClientName := aClient.ClientName;  
  
aTable := aTable.Create(nil);  
aTable.SessionName := aSession.SessionName;  
aTable.DatabaseName := 'MyAlias';
```

If multi-user support is required and there is a time-consuming process that needs to work as fast as possible, consider creating a custom FlashFiler Server. The server can support other clients as well as perform whatever processing is needed. The data processing has direct access to the server engine, giving it the best performance. See “Creating Your Own Server” on page 277 for an example.

## Disable and enable data-aware controls

When looping through records in a table, the VCL may be doing work to update data-aware controls connected to your dataset. When performing loops, consider adding calls to the `TDataSet` methods `DisableControls` and `EnableControls` to keep this interaction to a minimum. Look at the following example:

```
MyTable.DisableControls;
try
  while not MyTable.EOF do begin
    //Do Something
    MyTable.Next;
    Application.ProcessMessages;
  end;
finally
  MyTable.EnableControls;
end;
```

## Manually disable DBGrids

When iterating through records in a table that is connected to a data-aware grid, the grid still does some caching, even if you call `DisableControls` prior to executing the loop. The only way to keep this from happening is to manually remove the grid's reference to the data source. Here's an example:

```
var
  DataSource : TDataSource;
begin
  MyTable.DisableControls;
  DataSource := MyDBGrid.DataSource;
  MyDBGrid.DataSource := nil;
  try
    while not MyTable.EOF do begin
      //Do Something
      MyTable.Next;
      Application.ProcessMessages;
    end;
  finally
    MyDBGrid.DataSource := DataSource;
    MyTable.EnableControls;
  end;
end;
```

If you try this example, you'll notice that immediately after setting `DataSource` to `nil`, the grid redraws itself as an empty box. On the surface, there doesn't seem to be a simple way to keep the grid from repainting itself. However, you can send a `WM_SETREDRAW` message to the grid to prevent it from painting. Here's an improved version of the previous example:

```
var
    DataSource : TDataSource;
begin
    MyTable.DisableControls;
    DataSource := MyDBGrid.DataSource;
    {Turn off redraw}
    MyDBGrid.Perform(WM_SETREDRAW, 0, 0);
    MyDBGrid.DataSource := nil;
    try
        while not MyTable.EOF do begin
            //Do Something
            MyTable.Next;
            Application.ProcessMessages;
        end;
    finally
        MyDBGrid.DataSource := DataSource;
        {Turn on redraw}
        MyDBGrid.Perform(WM_SETREDRAW, 1, 0);
        {repaint the entire control}
        MyDBGrid.Invalidate;
        MyTable.EnableControls;
    end;
end;
```

## Monitor message traffic

FlashFiler Server displays statistics about message traffic for each transport. These statistics can be a great tool while working to improve performance.

The Messages column tells you how many messages have been processed by the transport since it was started or reset. You can use this value to compare how many messages are required for similar operations.

The Messages/Sec column helps you determine throughput. For instance, there are limits to how fast messages can be sent between client and server, especially across a network. You can use the Messages/Sec value to help realize these limits. Being aware of these limits helps you decide how much effort to put into reducing message traffic.

**Note:** The statistics can be reset using the FlashFiler Server's main menu Debug option.

For detailed information about the messages being passed, you can turn on the FlashFiler Server's debug log. Be aware, though, that debug logging dramatically reduces performance. Only use the debug log for debugging and profiling.

## Use batch methods

FlashFiler client includes many methods that allow you to process groups of records simultaneously. These routines allow the client to retrieve, insert, or delete many records with a single call to the server. These routines include GetRecordBatch, GetRecordBatchEx, and InsertRecordBatch. Each of these routines is described in "Batched Record Routines" on page 567.

**Note:** FlashFiler is not compatible with the VCL's TBatchMove component.

## Reduce disk activity

Reading and writing data from disk slows down any application and FlashFiler is no exception. Waiting for data to be read or written from disk is a potential bottleneck. This section contains suggestions to help reduce disk activity.

### Set Maximum RAM appropriately

Increasing the server's Maximum RAM setting to a value large enough to hold your most commonly used tables reduces disk activity because it stores data in memory. Since the data is in memory, it doesn't have to be retrieved from disk.

Be careful not to set the Maximum RAM to a value greater than your available RAM. Setting the value too high can cause repeated swaps with virtual RAM.

See "Operation" on page 33 for more information about setting a Maximum RAM value.

### Use explicit transactions

All FlashFiler operations are wrapped inside of a transaction. If you don't explicitly start a transaction, each operation is wrapped in its own implicit transaction. The performance issue is that each transaction is written to disk when it's complete. If you're updating numerous records using implicit transactions, you will have a lot of disk activity. Wrapping that same series of operations in an explicit transaction reduces disk activity and may dramatically increase performance. See "Transaction" on page 27 and "Transactions" on page 181 for more information about transactions.

## Other performance issues

Although the biggest performance gains come from reducing disk activity and message traffic, there are some other performance issues. This section focuses on performance issues that don't fall within the previous sections.

### Avoid calling RecordCount with an active filter

Reading the RecordCount property with an active filter inhibits performance. Record counts are normally very fast operations, however, if you have a filter active the application must cycle through all the records to make sure they match the filter criteria. Take this code, for example:

```
while MyTable.RecordCount > 0 do begin
    MyTable.Delete;
    Application.ProcessMessages;
end;
```

The problem with the previous code is that MyTable.RecordCount is read for each loop iteration. Operations that loop through a dataset should instead be written like this:

```
MyTable.First;
while not MyTable.EOF do begin
    MyTable.Delete;
    Application.ProcessMessages;
end;
```

### Extenders, monitors, user-indexes, fail-safe transactions, and custom encryption

Be aware of the impact of using extenders, monitors, user-defined indexes, fail-safe transactions, and custom encryption. Although we have made FlashFiler very efficient, there is a cost for using these features. The routines used in these operations may be called thousands of times per second and can dramatically degrade server performance in a large application.

### Limit calls to TffTable.Open and TffTable.Close methods

Postings in our support newsgroup suggest that some FlashFiler users make unnecessary calls to close tables and databases. The effect on performance is not on the close operation, but the subsequent open operation. Open operations can take some time, especially if the server does not already have the table open for another client. Connected grids, and other data-aware controls make the operation even more costly.

If you are closing a table just to select a different index or refresh the data you are closing the table needlessly. Simply change the IndexName directly or call Refresh; don't close the table.



## Lookup fields

You may have noticed that lookup fields have a tendency to slow database operations. By default, lookup fields do a lot of work to make sure they're up-to-date. Lookup fields are most often used to reference static information. If your lookup fields reference static information, be sure to set the field's `LookupCache` property to `True`. This loads the lookup values once when the dataset is first opened.

When the information in your lookup cache becomes invalid, a simple call to the field's `RefreshLookupList` method will update the cache.

## Use ranges and server-side filters

Setting a range before searching reduces the number of records that must be examined. Using filters isn't nearly as fast as using ranges, but `FlashFiler` allows you to reduce the time required to filter records by using server-side filters. Using a combination of ranges and server-side filters can produce amazingly fast searches.

## Use a single-user transport locally

Typically, there is no need to use a network transport like `IPX/SPX` or `TCP/IP` when client and server applications reside on the same machine. `FlashFiler`'s `Single User Transport` reduces much of the overhead required by the network transports.

---

## Chapter 15: Data Conversion

Due to new features in FlashFiler 2, such as support of files larger than 4 gigabytes and improved BLOB granularity, the FlashFiler table format has changed. In order to use FlashFiler 2 with existing applications, you must migrate your FlashFiler 1 tables to the new table format. There are three ways for you to convert your existing tables.

First, the FFCnvrt utility provides a graphical user interface for converting tables. If you simply need to convert your FlashFiler 1 tables so that you can start using FlashFiler 2, or if you do not have any need for automated conversions, use this tool.

Second, FFCnvrtC is a command line conversion utility. Use this tool to automate conversions or integrate conversions with your installation and patching programs.

Finally, the TffDataConverter class (in the FFCnvrt unit) provides direct access to the core logic of the data conversion. FFCnvrt and FFCnvrtC are simply interfaces that wrap around TffDataConverter. If you want to develop your own conversion interface, provide new options, or perform additional maintenance during conversion, use this class directly.

This chapter describes each tool and provides examples for both FFCnvrt and FFCnvrtC. The conversion executables are located in the Bin subdirectory of your FlashFiler installation. The source code for each utility and the TffDataConverter class is located in the Convert subdirectory.

---

## Conversion Utilities

Use the FFCnvrt and FFCnvrtC utilities to convert your FlashFiler 1 tables to FlashFiler 2 tables. FFCnvrt provides a graphical user interface while FFCnvrtC is a command line utility.

Both utilities work with all FlashFiler 1 tables and field types. The only exception is encrypted tables. The conversion utilities handle tables encrypted using the standard FlashFiler 1 server. If you customized the table encryption, you must recompile a DLL used by the conversion utilities. See Chapter 3 page “Installing New Encryption Routines” on page 52 for information on table encryption. See the “FlashFiler 1 Dynamic Link Library” on page 514 for details about recompiling the conversion DLL.

Both utilities require you to specify source and destination directories. The source directory is the location of your FlashFiler 1 tables. The destination directory is the location where the converted tables are to be placed. In any one run of a conversion utility, you can convert multiple tables but they must originate from the same source directory and be converted to the same destination directory.

# FFCnvrt: GUI conversion utility

FFCnvrt provides a graphical user interface for data conversion. It provides an intuitive way to convert your tables. Select the tables you want to convert, tell it where to place the converted tables, and click Convert. Watch the status information as the program progresses through your tables.

As shown in Figure 15.1, FFCnvrt's interface contains three sections: Source, Destination, and Status.

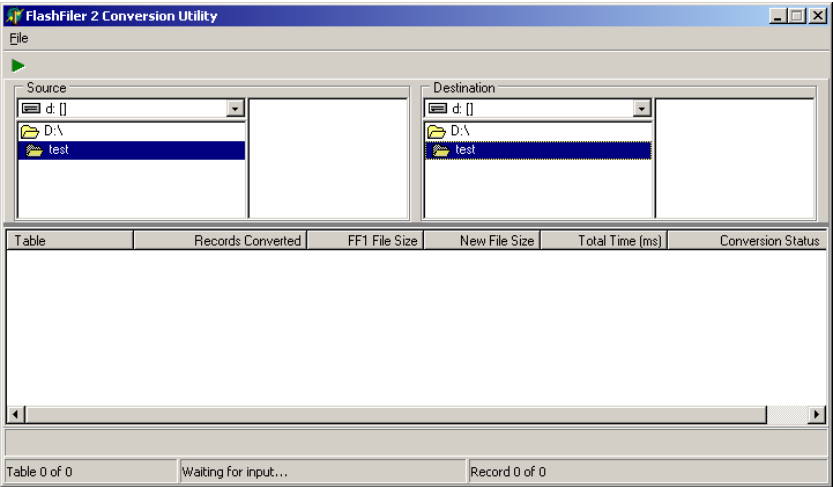


Figure 15.1: The conversion utility main window.

## Specify the source tables

Use the Source section to tell the utility which tables you want to convert. The section is located in the upper left-hand portion of the screen and is labeled Source. It contains three components: Drive List, Directory List, and Table List.

Use the Drive List in the upper left-hand portion of the screen to select the drive containing the tables.

Use the Directory List below the Drive List to choose the directory containing the tables.

After you choose the source drive and directory, the FlashFiler tables located in that directory appear in the Table List. Select one or more tables to convert.

## Specify the destination tables

Use the Destination section to specify the location of the converted tables. Use the Drive List and Directory List controls just as you did in the Source section.

**Note:** The Destination Tables List is read-only. The tables in the destination directory are listed so that you can see if you're about to overwrite existing FlashFiler tables. There's no problem with overwriting tables; we just want to ensure that's what you intend. The interface confirms that you want to overwrite tables before it does so.

## Watch the conversion status

Everything below the Source and Destination sections displays the status of the conversion. The Status section is comprised of three parts: the Status View, the Progress Bar, and the Status Bar.

The Status View shows information about each table being converted. This information includes the table name, number of records converted, original file size, converted file size, length of time to convert, and conversion status.

The Progress Bar immediately below the Status View indicates the progress of each table. The progress bar is empty until a conversion starts.

The Status Bar at the bottom of the screen contains three panels. The left panel, shows which table in the Status View is being converted. The middle panel summarizes the current stage of the conversion. The right panel shows how many records in the current table have been converted.

## Try an example

Since you're probably anxious to get FlashFiler 2 running, we'll convert your FlashFiler 1 server tables as an example. If you remember, the FlashFiler Server creates several tables in its directory in order to store alias, user, and configuration information. These tables are named FFSInfo.FF2, FFSAlias.FF2, FFSUser.FF2, and FFSIndex.FF2.

**Note:** FlashFiler Server creates FFSIndex.FF2 only if you take advantage of User Defined Indexes (see "User-Defined Indexes" on page 543). Do not be alarmed if you do not find an FFSIndex table.

The FlashFiler Server automatically creates these tables in the directory containing the FlashFiler Server executable. In addition, applications compiled as a FlashFiler SingleEXE also write server tables to the directory containing the executable. Assuming you locate a suitable set of server tables and have FlashFiler 2 installed, let's step through a conversion:

1. Start the FFCnvrt utility using the shortcut on your FlashFiler menu. Optionally, start FFCnvrt from the Bin subdirectory of your FlashFiler installation.
2. Once FFCnvrt starts, select the drive containing the FlashFiler 1 tables (see [2] in Figure 15.2).
3. Select the directory containing the FlashFiler 1 tables (see [3] in Figure 15.2). Once you select the directory, the FlashFiler 1 server tables appear in the Table List (see [4]).

4. Select the tables to convert (see [5]).
5. Select the destination drive (see [6]).
6. Select the destination directory (see [7]).
7. Push the Convert button (see [8]). Now, you will see the selected tables listed in the Status section. As the tables are converted, the progress and status of each table are updated. During the conversion the Convert button changes to a Cancel button. When the conversion is complete, the Cancel button is labeled Convert once again.
8. Close FFCnvrt and load your FlashFiler 2 server. FlashFiler 2 server options have changed some, so you need to verify them.

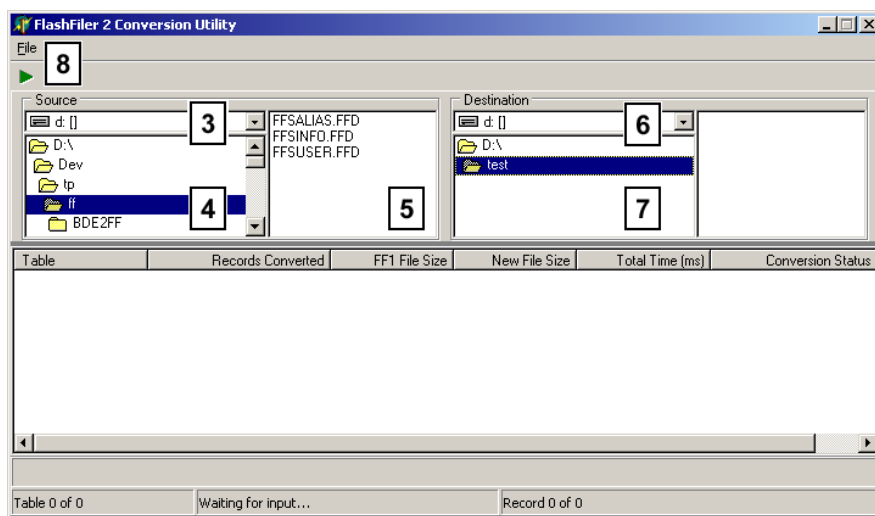


Figure 15.2: The steps for using the conversion utility main window.

**Note:** NetBIOS is not supported in FlashFiler 2. When converting a FFSInfo.FF2 server table, a protocol options screen will appear if your previous server was set to use the NetBIOS protocol. You need to set at least one new protocol. See Figure 15.3 for the Network Configuration dialog box. See “Config | Network configuration...” on page 38 for additional information about setting up your protocols.

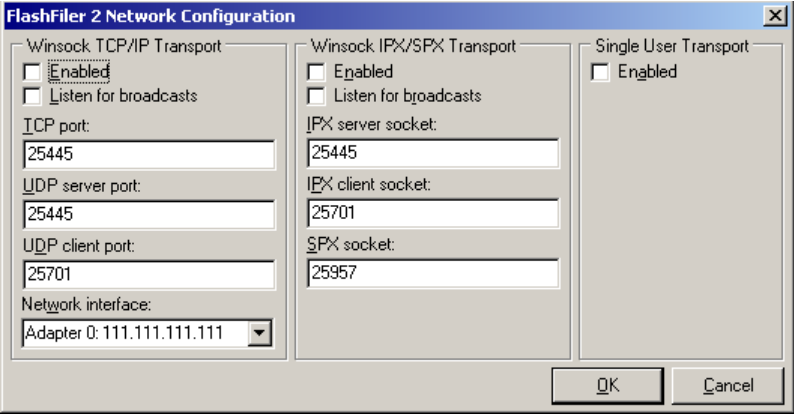


Figure 15.3: The Network Configuration dialog box.

## FFCnvtC: Command line conversion utility

FFCnvtC provides the same functionality as FFCnvt without the user interface. Use FFCnvtC to automate conversions or to accomplish “unmanned” conversions. Unmanned conversions prevent your users from having to perform the conversions. You simply call FFCnvtC from a batch file or application with the correct parameters. The conversion is accomplished without any interaction from your users.

### Parameters

FFCnvtC recognizes the types of parameters as listed in Table 15.1.

**Table 15.1:** *Conversion utility command line arguments .*

Parameters	Purpose
-s	Source directory: location of files to convert
-t	Tables to convert (no path or extension)
-d	Destination directory: location to create converted tables

There are a few things to remember about these parameters:

- You can have multiple table parameters but only one source and destination.

- If you don't list any table (-t) parameters, all tables in the source directory are converted.
- Ensure the destination doesn't contain any tables with the same name as a source table.

## FFCnvrtC examples

Let's look at a few quick examples and then we'll step through how to convert your server tables as we did for the graphical conversion.

### Valid calls to FFCnvrtC

```
FFCnvrtC -sC:\FF -tFFSUser -tFFSAlias -dC:\FF2
```

```
FFCnvrtC -dD:\NewApp -sD:\OldApp -tOldData
```

### Invalid calls to FFCnvrtC

```
FFCnvrtC -sC:\FF1 -tMyTable -sC:\FF2 -tOtherTable -dC:\NewFF
```

(Only one source is allowed.)

```
FFCnvrtC -sC:\FF -tFFSInfo -tFFSUser -tFFSAlias
```

(No destination given)

### Converting server tables

Now we'll step through converting your server tables using FFCnvrtC. If you remember, the FlashFiler Server creates several tables in its directory in order to store alias, user, and configuration information. These tables are named FFSInfo.FF2, FFSAlias.FF2, FFSUser.FF2, and FFSIndex.FF2

**Note:** FlashFiler Server creates FFSIndex.FF2 only if you take advantage of User Defined Indexes (See "User-Defined Indexes" on page 543.). Do not be alarmed if you do not find an FFSIndex table.

Before we get started you need to know where you execute your FlashFiler 1 server and see which system tables you have to convert. You also need to know where you'll be executing your FlashFiler 2 server. Both servers are placed in the Server subdirectory by default, but you may execute yours from a different location.

For the example, my FlashFiler 1 server is installed in C:\FF and my FlashFiler 2 server is installed in D:\FF2. Both servers are executed from their default location.

1. Start up a command session. The easiest way to get to a command session is to click the Start button, click Run, type "command", and hit the Enter key.
2. Change to the location where the FFCnvrtC executable is located. It's installed in the Bin subdirectory of your FlashFiler installation.



3. Type in “FFCnvrtC” at the command line. Add your source parameter with the location of your FlashFiler 1 tables. Add a table parameter for each of your server tables. Then add a destination parameter with the location of your FlashFiler 2 server. Following is an example of FFCnvrtC:

```
FF2Cnvrt -sC:\FF\Server -tFFSInfo  
-tFFSAlias -tFFSUser -tFFSIndex -dD:\FF2\Server
```

4. Press the Enter key and the conversion begins. FFCnvrtC displays a message when it's complete. If the message is “Conversion successful”, you're ready to start the FlashFiler 2 server and verify your settings. If you receive an error message, see “Status” on page 506 and try the conversion again.

## Status

FFCnvrtC reports the status of the conversion in two ways: it displays a message to the screen and it sets an exit code. If executed from a batch file or application, the batch file or execution should check the exit code and attempt to handle any error that occurs.

The way you check the error code depends on how you execute FFCnvrtC. If you're executing it from a Delphi or C++ Builder application, call the `GetExitCodeProcess` Win32 API to retrieve the exit code. If calling it from a batch file, check the `ErrorLevel` value.

Table 15.2 lists the exit codes, associated messages, and their meanings.

**Table 15.2:** *Command-line conversion utility exit codes.*

Exit Code	Message	Meaning
0	Conversion successful	No problems were encountered
1	ERROR: No destination parameter	FFCnvrtC must know where to put the newly created tables
2	ERROR: No source parameter	FFCnvrtC needs to know the location of the files to convert
3	ERROR: Too many source parameters	Only one source parameter is allowed
4	ERROR: No tables to convert	No table parameters and none in the source directory
5	ERROR: <table name> doesn't exist	Table doesn't exist in the source directory
6	ERROR: Source and destination in same location	Source and destination directories must be in different locations

**Table 15.2:** *Command-line conversion utility exit codes. (continued)*

Exit Code	Message	Meaning
7	ERROR: <table name> already in destination	A table with this name is already in your destination (See the Conversion Utilities introduction for more information.)
8	ERROR: <source path> doesn't exist	Source directory does not exist
9	ERROR: <destination path> doesn't exist	Destination directory does not exist
10	ERROR: Conversion of <table name> failed	Error while retrieving records from the table
11	ERROR: (No message)	No parameters given (A short help screen is displayed instead of an error message.)
12	ERROR: Unknown failure	Conversion not successful - unable to determine the reason

---

## TffDataConverter Class

The TffDataConverter class provides the functionality needed to convert FlashFiler 1 tables to FlashFiler 2 tables. It is designed to handle all field types including BLOBs and file BLOBs. It will also convert FlashFiler Server tables, but be aware that the server options have changed and need to be verified when first used.

It is not possible to have FlashFiler 1 and FlashFiler 2 executing in the same application. To get around this limitation, the class depends on the FlashFiler 1 server to be executed from a DLL named FF1Intfc.DLL. This DLL provides the functionality needed to access and retrieve data from a FlashFiler 1 table. If you're encrypting FlashFiler 1 tables with your own key, you will need to recompile the FF1Intfc project with your key and with both SecureServer and SingleEXE defined.

The FlashFiler 2 server used for the conversion is passed in from TffDataConverter's constructor. Since you create this server before passing it in, you have the opportunity to set any options you need. You may even want to compile it with an encryption key and SecureServer defined so you can create encrypted tables during the conversion. See Chapter 3 "Installing New Encryption Routines" on page 52 for information about how to create encrypted tables.

Source code for both of the conversion utilities is included in the Convert subdirectory. The source code for these projects serve as examples of how to use the TffDataConverter class and the FlashFiler 1 DLL.

# Hierarchy

TObject (VCL)

    TffDataConverter (FFConvrt)

# Properties

Canceled	ProgressFrequency	Source
BufferSize	RecordsProcessed	TotalRecords
Destination	ServerEngine	

# Methods

Cancel	Convert	Create
--------	---------	--------

# Events

AfterConvert	OnComplete
BeforeConvert	OnNetBios
OnCancel	OnProgress

## Reference Section

### AfterConvert

event

```
property AfterConvert : TffDataConverterEvent  
  
TffDataConverterEvent = procedure(  
    aSender : TffDataConverter) of object;
```

✚ Defines an event handler that is called when all records are converted.

The AfterConvert event is triggered immediately after all records in a FlashFiler 1 table have been converted into a FlashFiler 2 table. This event is not triggered if the conversion is canceled.

See also: BeforeConvert, OnComplete

### BeforeConvert

event

```
property BeforeConvert : TffDataConverterEvent  
  
TffDataConverterEvent = procedure(  
    aSender : TffDataConverter) of object;
```

✚ Defines an event handler that is called before records are converted.

See also: AfterConvert, OnComplete

### BufferSize

property

```
property Buffersize : TffWord32  
  
Default: 1,048,576
```

✚ Determines how many bytes the converter used to buffer outstanding transactions.

The converter uses explicit transactions when inserting records into the new FlashFiler 2 table. The BufferSize property helps to determine when to commit transactions. The calculation is  $\text{BufferSize} \div \text{RecordLength} = \text{CommitFrequency}$ .

Let's look at an example. Using a 1,048,576 byte (1megabyte) buffer with a record length of 66 bytes. 1,048,576 divided by 66 equals 15887. This means each transaction in this conversion will be committed after every 15887 records are inserted into the new table.

See also: RecordsProcessed

## Cancel

method

```
procedure Cancel;
```

↪ Cancels the conversion process.

Interrupts the conversion. The new FlashFiler 2 table is deleted from the hard drive.

See also: Canceled, OnCancel

## Canceled

read-only property

```
property Canceled : Boolean
```

↪ Identifies whether the current conversion has been canceled.

Canceled is True only after the Cancel method is called.

See also: Cancel, OnCancel

## Convert

method

```
procedure Convert(const aSource : string; const aDest : string);
```

↪ Initiates the conversion of a FlashFiler 1 table to a FlashFiler 2 table.

aSource is the complete path and filename to the FlashFiler 1 table. aDest is the location (drive and directory) where you want the new FlashFiler 2 table to be created. The FlashFiler 2 table will be given the same name as the FlashFiler 1 table. The destination directory cannot be the same location as the source table. The following line of source code illustrates a call to this method:

```
MyConverter.Convert('C:\OldDir\FFTable.FF2', 'D:\NewDir');
```

If there are problems opening the source table or creating the new table, a `EffConverterException` exception is raised.

`EffConverterException`

See also: Destination, OnNetBios, Source, RecordsProcessed

## Create

## constructor

```
constructor Create(aServerEngine : TffBaseServerEngine);
```

↳ Creates an instance of TffDataConverter.

aServerEngine is the server engine used to perform the FlashFiler 2 part of the conversion. aServerEngine can be an embedded or remote server engine. Be sure to set any server options you need before creating the TffDataConverter.

If you want your converter to create encrypted tables during the conversion, compile aServerEngine with your encryption key. See Chapter 3 “Installing New Encryption Routines” on page 52 to learn about custom encryption.

See also: ServerEngine

## Destination

## read-only property

```
property Destination : string
```

↳ Returns the location where the converted table is created.

See also: Convert, Source

## OnCancel

## event

```
property OnCancel : TffDataConverterEvent
```

```
TffDataConverterEvent = procedure(  
    aSender : TffDataConverter) of object;
```

↳ Defines an event handler that is called when a table conversion is canceled.

See also: Cancel, Canceled

## OnComplete

## event

```
property OnComplete : TffDataConverterEvent
```

```
TffDataConverterEvent = procedure(  
    aSender : TffDataConverter) of object;
```

↳ Defines an event handler that is called when the conversion of a table completes.

This event is fired after the AfterConvert event has fired and is the last event to be fired in the conversion process. It is not triggered if the conversion is canceled.

See also: AfterConvert, BeforeConvert

```
property OnNetBios : TffDCNetBiosEvent

TffDCNetBiosEvent = procedure(aSender : TffDataConverter;
    var aCanceled : Boolean;
    var aOptions : TffProtOptions) of object;
```

- ↳ Defines an event handler that is called when converting a server table set to use the NetBIOS protocol.

Among other things, the FFSINFO.FF2 server table identifies the FlashFiler Server's protocol. If the protocol is set to NETBIOS this event is raised to allow setting a new protocol.

---

**OnProgress****event**

```
property OnProgress : TffDataConverterEvent

TffDataConverterEvent = procedure(
    aSender : TffDataConverter) of object;
```

- ↳ Defines an event handler that is called every time a progress point is reached.

The progress points are determined by the ProgressFrequency property.

If there are fewer records in the table than the ProgressFrequency value, this event will be triggered after the conversion is complete. This event is also generated at the end of the conversion if TotalRecords is not a multiple of ProgressFrequency in order to show all records were successfully processed. All OnProgress events are fired before the AfterConvert event.



The following are examples of possible firing points. All examples use the default ProgressFrequency of 100:

**Example 1:**

Converting a table with 89 records. The OnProgress event is only called once because the table has fewer records than the ProgressFrequency value.

**Example 2:**

Converting a table with 500 records. The OnProgress event is called 5 times; once for every 100 records.

**Example 3:**

Converting a table with 555 records. The OnProgress event is called six times; once for every 100 records and once for the last 55 records.

FFCnvrt and FFCnvrtC use this event to update the Status View, Progress Bar, and Status Bar.

See also: AfterConvert, BeforeConvert, ProgressFrequency, RecordsProcessed, TotalRecords

<b>ProgressFrequency</b>	<b>property</b>
--------------------------	-----------------

property ProgressFrequency : TffWord32

Default: 100

↪ Determines when the OnProgress event is triggered.

Set ProgressFrequency to cause the OnProgress event to fire at specific intervals during the conversion. A ProgressFrequency value of 100 means the OnProgress event will fire after every 100 records are converted. See OnProgress for examples of when the OnProgress event is generated.

Setting ProgressFrequency to 0 will prevent any OnProgress events from being generated.

See also: AfterConvert, BeforeConvert, OnProgress, RecordsProcessed, TotalRecords

<b>RecordsProcessed</b>	<b>read-only property</b>
-------------------------	---------------------------

property RecordsProcessed : TffWord32

↪ Reports how many records have been converted.

See also: BufferSize, OnProgress, ProgressFrequency, TotalRecords

## **ServerEngine**

**read-only property**

property ServerEngine : TffBaseServerEngine

↳ Returns the FlashFiler 2 server used to instantiate the TffDataConverter.

See also: Create

## **Source**

**read-only property**

property Source : string

↳ Returns the complete path and filename of the FlashFiler 1 table being converted.

See also: Convert, Destination

## **TotalRecords**

**read-only property**

property TotalRecords : TffWord32

↳ Retrieves the total number of records in the source table.

This property has a value of zero until the conversion process retrieves it from the FlashFiler 1 table.

See also: ProgressFrequency, RecordsProcessed

---

# FlashFiler 1 Dynamic Link Library

The FlashFiler 1 DLL retrieves data from FlashFiler 1 tables. Since it's not possible to have FlashFiler 1 and FlashFiler 2 in the same application, this DLL allows the FFDataConverter class to retrieve data from a FlashFiler 1 table. We named the FlashFiler 1 DLL FF1Intfc.DLL and it is located in the FlashFiler 2 Bin subdirectory.

If you encrypt your FlashFiler 1 tables with your own key, you must recompile the FlashFiler 1 DLL. When you recompile, ensure you include your key and have both SecureServer and SingleEXE defined in FFDEFINE.INC. See Chapter 3 “Installing New Encryption Routines” on page 52 for more information about using encrypted tables.

## Calling DLL routines

The calls to FF1Intfc are much like the calls from any other FlashFiler application. First, call FF1DirOpen to tell the FlashFiler 1 server where to look for its tables. FF1TableOpen is called to open a table in the database. Calling FF1TableDataDictionary gets a copy of the FlashFiler 1 table's data dictionary. Once the table is opened you can call FF1IsFileBLOB, FF1TableEOF, FF1TableFieldValue, FF1TableFirst, FF1TableNext, and FF1TableRecordCount as needed. When you're finished with a table, call FF1TableClose.

The FF1GetMem, FF1FreeMem, and FF1ReallocMem are not used for table operations. We use them to prevent problems with sharing long strings between an application and a DLL. These routines are described in the FlashFiler Memory Manager section of this chapter.

## Functions and Procedures

FF1DirOpen	FF1TableClose	FF1TableNext
FF1FreeMem	FF1TableDataDictionary	FF1TableOpen
FF1GetMem	FF1TableEOF	FF1TableRecordCount
FF1IsFileBLOB	FF1TableFieldValue	
FF1ReallocMem	FF1TableFirst	

## Reference Section

### **FF1DirOpen**

**procedure**

```
procedure FF1DirOpen(aPath : PChar); stdcall;
```

- ↳ Establishes a source directory for FlashFiler 1 tables.

aPath is the directory containing the tables.

### **FF1FreeMem**

**procedure**

```
procedure FF1FreeMem(P : Pointer);
```

- ↳ Frees memory that was allocated using the DLL's memory manager.

P is the address of the chunk of memory to be deallocated. See the “Conversion Memory Manager” on page 520 for more information.

See also: FF1GetMem, FF1ReallocMem

### **FF1GetMem**

**procedure**

```
procedure FF1GetMem(var P : Pointer; aSize : Integer);
```

- ↳ Allocates memory using the DLL's memory manager.

P is any type of pointer. aSize is the size of the memory block to allocate. If the call to FF1GetMem is successful, P contains the address to the new memory block. See “Conversion Memory Manager” on page 520 for more information.

See also: FF1FreeMem, FF1ReallocMem

### **FF1IsFileBLOB**

**function**

```
function FF1IsFileBLOB(aFieldNo : Integer;  
    var aBuffer : Array of Byte) : Boolean;
```

- ↳ Determines if a FlashFiler 1 BLOB field of the active record points to a file BLOB.

If the field contains a file BLOB, this function returns True and places the complete path and filename of the file in aBuffer. The first byte of aBuffer contains the length of the path and filename.

aFieldNo is the field number of the BLOB field to check in the active record.

## **FF1ReallocMem**

**method**

```
procedure FF1ReallocMem(var P : Pointer; aSize : Integer);
```

- ↳ Resizes memory allocated using the DLL's memory manager. P is the address of the block of memory to be resized. aSize is the new size.

See the “Conversion Memory Manager” on page 520 for more information.

See also: FF1GetMem, FF1FreeMem

## **FF1TableClose**

**method**

```
procedure FF1TableClose;
```

- ↳ Closes a FlashFiler 1 table.

Call FF1TableClose when finished with a FlashFiler 1 table.

See also: FF1TableOpen

## **FF1TableDataDictionary**

**method**

```
procedure FF1TableDataDictionary(var aDict : TStream);
```

- ↳ Retrieves the data dictionary of a FlashFiler 1 table.

The structure of the data dictionary has not changed from FlashFiler 1 to FlashFiler 2 so it is used to create the new FlashFiler 2 table. The dictionary is returned in aDict.

## **FF1TableEOF**

**method**

```
function FF1TableEOF : Boolean;
```

- ↳ Determines if a FlashFiler 1 table is positioned at the end of the table.

## **FF1TableFieldValue**

**method**

```
function FF1TableFieldValue(aFieldNo : Integer) : Variant;
```

- ↳ Retrieves the value of a field for the current record.

aFieldNo is the field number being retrieved. The value of the field is returned as a Variant type.

If this method is called for a BLOB field that contains a file BLOB, it will return the BLOB. Use FF1IsFileBLOB to retrieve the filename of a file BLOB.

See also: FF1IsFileBLOB

## **FF1TableFirst** method

---

```
procedure FF1TableFirst;
```

↪ Moves the cursor to the first record of a FlashFiler 1 table.

See also: FF1TableNext

## **FF1TableNext** method

---

```
procedure FF1TableNext;
```

↪ Moves the cursor to the next record of a FlashFiler 1 table.

This is a wrapper for TffTable.Next.

See also: FF1TableEOF

## **FF1TableOpen** method

---

```
function FF1TableOpen(aTableName : PChar) : Integer;
```

↪ Opens a FlashFiler 1 table and prepares the FlashFiler 1 server.

aTableName is the filename (no path or extension) of the table to open.

This function returns zero if it successfully opens the table and -1 if the table couldn't be opened.

## **FF1TableRecordCount** method

---

```
function FF1TableRecordCount : Integer;
```

↪ Returns the number of records in the opened FlashFiler 1 table.

---

# Conversion Memory Manager

The Conversion Memory Manager is used to prevent problems when passing long strings from the FlashFiler 1 DLL and the conversion utilities. There are inherent problems with passing long strings between an application and a DLL. The VCL provides the ShareMem unit to avoid these problems, but it requires a hefty DLL called BORLANDMM.DLL to be deployed with any applications that uses this unit.

To circumvent these problems the FlashFiler 1 DLL uses its own memory manager called FFMemMgr. Like ShareMem, FFMemMgr is listed first in the using project's USES section. During initialization, the memory manager redirects the calls from the conversion application's memory manager to that of the DLL's memory manager.

For those who wish to develop their own means of converting tables for use with FlashFiler 2, we provide the FFMemMgr unit for your use.

## Functions

FFMMFreeMem	FFMMReallocMem
FFMMGetMem	LoadFF1DLL

## Reference Section

---

<b>FFMMFreeMem</b>	<b>function</b>
--------------------	-----------------

---

```
function FFMMFreeMem(P : Pointer) : Integer;
```

↪ Frees memory allocated via FFMMGetMem.

P is the address of the chunk of memory to dispose of. 0 is returned if the call is successful.

---

<b>FFMMGetMem</b>	<b>function</b>
-------------------	-----------------

---

```
function FFMMGetMem(Size : Integer) : Pointer;
```

↪ Allocates memory using the FlashFiler 1 DLL's memory manager.

Size is the size of the chunk of memory requested. If successful, the address of the new block of memory is returned. If not successful, an EOutOfMemory exception will be raised.

EOutOfMemory

---

<b>FFMMReallocMem</b>	<b>function</b>
-----------------------	-----------------

---

```
function FFMMReallocMem(P : Pointer; Size : Integer) : Pointer;
```

↪ Resizes memory previously allocated by FFMMGetMem.

P is the block of memory to resize. Size is the new size. If successful, a pointer with the same address as P will be returned.

---

<b>LoadFF1DLL</b>	<b>function</b>
-------------------	-----------------

---

```
function LoadFF1DLL : Boolean;
```

↪ Loads the FlashFiler 1 DLL.

Returns True if successful or False if the DLL cannot be loaded.





---

# Chapter 16: Appendices

This chapter provides information on a set of miscellaneous topics.

- “FlashFiler Data Types” describes the data types FlashFiler supports as field types in its tables.
- “Converting Data Types” describes the data types that FlashFiler Server can convert between when restructuring a table.
- “Importing Data into FlashFiler Tables,” describes the FlashFiler import facility. It is used to import data into FlashFiler format from ASCII files, typed binary files, and B-Tree File files.
- “User-Defined Indexes,” explains what a user-defined index is, how they work, and shows you how to define your own index types.
- “Client Configuration” introduces the FlashFiler Client Communications Utility and lists some routines to update FlashFiler’s registry settings.
- “FlashFiler-Specific Routines” discusses some BDE-style routines that FlashFiler has implemented to cover the gaps between the VCL and FlashFiler.
- “Batched Record Routines” describes routines to speed up inserting and retrieving multiple records.
- “SingleEXE Applications” describes what a singleEXE application is, how they differ from client/server applications, and how to implement a singleEXE application.
- “TffDataDictionary Class” is a reference section devoted to the data dictionary used by FlashFiler for creating tables.
- “Crystal Reports Support” shows you how to use Crystal Reports with FlashFiler.
- “Example Programs” describes the example programs shipped with FlashFiler.
- “Modifying the FlashFiler Source Code” lists some cautions, tells where to find a list of known bugs and interim bug fixes, and contains instructions for recompiling the FlashFiler code.

# FlashFiler Data Types

There are two kinds of data types in FlashFiler: physical and logical.

Physical data types are defined within FlashFiler, and are used to describe how a field's value is physically stored in memory and on disk.

Logical data types are defined generically within the BDE. FlashFiler translates from the physical type to the nearest logical type as necessary. The VCL uses the logical type to determine the type of TField descendant required for editing or displaying the data type. This means that your programs, components, or routines do not need to know about every type of data used by all of the database engines. Understanding the logical types should suffice.

FlashFiler supports two field types specific to TurboPower Software's SysTools product: `ffstDate` and `ffstTime`. `ffstDate` and `ffstTime` correspond to the SysTool's `TStDate` and `TStTime` types. The only difference between `TDateTime` (i.e., field type `ffstDateTime`) and these types is the manner in which the data is stored. The `ffstDate` and `ffstTime` types require only 4 bytes each, compared to the 8 bytes required to store a `TDateTime`. However, there is no difference in the way the client application sees `TDateTime`, `ffstDate`, and `ffstTime`. FlashFiler Client converts types `ffstDate` and `ffstTime` into `TDateTime` values. Note that TurboPower's SysTools product is not required in order to use field types `ffstDate` and `ffstTime`.

Table 16.1 lists the FlashFiler physical data types, a short description, and the storage requirements for each type.

Table 16.1: *Physical data types.*

Data Type	Description	Storage Requirements
<code>ffstBoolean</code>	Field having 1 of 2 states; usually True or False	1 byte
<code>ffstChar</code>	Any upper- or lowercase letter or number	1 byte
<code>ffstWideChar</code>	a single character from a wide-character set, such as Unicode	2 bytes
<code>ffstByte</code>	8-bit unsigned integer with a range of 0..255	1 byte
<code>ffstWord8</code>	8-bit unsigned integer with a range of 0..255	1 byte

**Table 16.1: Physical data types. (continued)**

<b>Data Type</b>	<b>Description</b>	<b>Storage Requirements</b>
fftWord16	16-bit unsigned integer with a range of 0..65535	2 bytes
fftWord32	32-bit unsigned integer with a range of 0..4294967295	4 bytes
fftInt8	8-bit signed integer with a range of -128..127	1 byte
fftInt16	16-bit signed integer with a range of -32768..32767	2 bytes
fftInt32	32-bit signed integer with a range of -2147483648..2147483647	4 bytes
fftAutoInc	Value that starts at 1 and automatically increments by 1 as each record is inserted )See "SetTableAutoIncValue" on page 229)	4 bytes
fftSingle	Floating-point number with a range of $1.5 \times 10^{-45}$ .. $3.4 \times 10^{38}$	4 bytes
fftDouble	Floating-point number with a range of $5.0 \times 10^{-324}$ .. $1.7 \times 10^{308}$	8 bytes
fftExtended	Floating-point number with a range of $3.6 \times 10^{-4951}$ .. $1.1 \times 10^{4932}$	10 bytes
fftComp	64-bit signed number with a range of $-2^{63}+1$ .. $2^{63}-1$	8 bytes
fftCurrency	Currency with a defined 4 decimal places having a range of -922337203685477.5808..922337203685477.5807	8 bytes

**Table 16.1: Physical data types. (continued)**

<b>Data Type</b>	<b>Description</b>	<b>Storage Requirements</b>
fftStDate	TurboPower's SysTools date with a range of Jan 1, 1600 - Dec 31, 3999	4 bytes
fftStTime	TurboPower's SysTools time (HH:MM:SS) that can represent any time of the day to the second	4 bytes
fftDateTime	VCL's TDateTime	8 bytes
fftBLOB	Binary Large Object	Actual size of BLOB plus some overhead <sup>1</sup>
fftBLOBMemo	Memo field	Actual size of BLOB plus some overhead <sup>1</sup>
fftBLOBFmtMemo	Rich edit memo field	Actual size of BLOB plus some overhead <sup>1</sup>
fftBLOBOLEObj	OLE Object	Actual size of BLOB plus some overhead <sup>1</sup>
fftBLOBGraphic	Graphic data such as a jpeg or bmp file	Actual size of BLOB plus some overhead <sup>1</sup>
fftBLOBDBSOLEObj	dBase OLE object	Actual size of BLOB plus some overhead <sup>1</sup>
fftBLOBTypedBin	Typed binary data	Actual size of BLOB plus some overhead <sup>1</sup>
fftByteArray	Fixed size array of raw bytes	1 byte per unit
fftShortString <sup>3</sup>	Short character string	1 byte per unit plus 1 byte <sup>2</sup>
fftShortAnsiString <sup>3</sup>	Short character string	1 byte per unit plus 1 byte <sup>2</sup>

**Table 16.1:** *Physical data types. (continued)*

Data Type	Description	Storage Requirements
fftNullString <sup>3</sup>	Null-terminated string	1 byte per unit plus 1 byte <sup>2</sup>
fftNullAnsiString <sup>3</sup>	Null-terminated string	1 byte per unit plus 1 byte <sup>2</sup>
fftWideString	String of Unicode characters	2 bytes per unit plus 2 bytes <sup>2</sup>

<sup>1</sup>FlashFiler’s method of storing BLOBs is described on page 19, “Binary Large Objects.” BLOB fields can’t be included in user-defined indexes, so the size information doesn’t apply to creating user-defined indexes. It should also be noted that FlashFiler doesn’t distinguish between the different BLOB types; this is only for the developer’s benefit.

<sup>2</sup>Units refers to the units field on FlashFiler Explorer’s Create Table, View Definition, or Redefine menus. Units also refers to the aUnits parameter of the Add Field method of the TffDataDictionary class and the InsertField method of the TffDataDictionary class. The units can be retrieved using the FieldUnits property of the TffDataDictionary class.

<sup>3</sup>The difference in fftShortString and fftShortAnsiString and between fftNullString and fftNullAnsiString is how they’re sorted. fftShortString and fftNullString are sorted using a straight byte-by-byte character comparison without regard to non-alphanumeric characters or locale. fftShortAnsiString and fftNullAnsiString types are sorted with the Win32 SDK’s CompareStringA function using the current user’s locale and the “word sort” technique.

Table 16.2 demonstrates the difference between the ANSI and non-ANSI types.

**Table 16.2:** *An example of ANSI vs. Non-ANSI sort results.*

ANSI Sort	Non-ANSI Sort
Smith	Smith
Smithes	Smith’s
Smith-Jones	Smith-Jones
Smith’s	Smith-Smithe
Smithsaint	Smithes
Smith-Smithe	Smithsaint
Smithson	Smithson

Now that you have a good understanding of FlashFiler data types, Table 16.3 should clarify the differences in physical and logical data types and how FlashFiler, the BDE, and the VCL look at data types differently.

**Table 16.3:** *Data type comparison.*

<b>From FlashFiler Physical Type</b>	<b>To BDE Logical Type</b>	<b>To VCL TField Type</b>
fftBoolean	fldBOOL	TBooleanField
fftChar	fldZSTRING	TStringField
fftWideChar	fldZSTRING	TStringField
fftByte	fldUINT16	TWordField
fftWord16	fldUINT16	TWordField
fftWord32	fldUINT32	TIntegerField
fftInt8	fldINT16	TSmallIntField
fftInt16	fldINT16	TSmallIntField
fftInt32	fldINT32	TIntegerField
fftAutoInc	fldINT32/fldstAUTOINC	TAutoIncField
fftSingle	fldFLOAT	TFloatField
fftDouble	fldFLOAT	TFloatField
fftExtended	fldFLOAT	TFloatField
fftComp	fldFLOAT	TFloatField
fftCurrency	fldFLOAT/fldstMONEY	TCurrencyField
fftStDate	fldDATE	TDateField
fftStTime	fldTIME	TTimeField
fftDateTime	fldTIMESTAMP	TDateTimeField
fftBLOB	fldBLOB/fldstBINARY	TBlobField
fftBLOBMemo	fldBLOB/fldstMEMO	TMemoField
fftBLOBFmtMemo	fldBLOB/fldstFMTCMEMO	TBlobField
fftBLOBOLEObj	fldBLOB/fldstOLEOBJ	TBlobField
fftBLOBGraphic	fldBLOB/fldstGRAPHIC	TGraphicField
fftBLOBDBSOLEObj	fldBLOB/fldstDBSOLEOBJ	TBlobField
fftBLOBTypedBin	fldBLOB/fldstTYPEDBINARY	TBlobField
fftByteArray	fldBYTES	TBytesField

**Table 16.3:** *Data type comparison. (continued)*

From FlashFiler Physical Type	To BDE Logical Type	To VCL TField Type
fftShortString	fldZSTRING	TStringField
fftShortAnsiString	fldZSTRING	TStringField
fftNullString	fldZSTRING	TStringField
fftNullAnsiStr	fldZSTRING	TStringField
fftWideString	fldZSTRING	TStringField

### Client applications always use logical data types

This differentiation between physical and logical types is most likely to trip developers up when they are using date fields, although other field types can also pose similar problems.

FlashFiler supports the SysTools date type (TStDate) and the TDateTime date type. There are two main differences between the two varieties:

- Size, or the number of bytes in the raw physical record.
- Range, or the number of dates that can be represented.

The SysTools date type is 4 bytes long (a longint) and has a date range of Jan 1, 1600 to Dec 31, 3999. The TDateTime date type is 8 bytes long (a double) and has a date range of Jan 1, 0001 to Dec 31, 9999. (In fact, to be totally accurate, FlashFiler uses the Delphi 1 TDateTime physical representation rather than the 32-bit TDateTime representation. The difference being the base date used to represent the date value.)

When you are designing a table, you should determine the range in which you want your dates to be restricted. If it were a business application, in general you would only be using dates in the 20<sup>th</sup> and 21<sup>st</sup> centuries, so the SysTools date type would be ideal. Furthermore, by using that type, you would be saving 4 bytes per date field per record over the TDateTime field type. In general, we recommend that you use the SysTools date type rather than the TDateTime variety.



Once you have designed the record layout for the table and fixed the types for each of your date fields, you should forget all about it. Put it out of your mind whether you are using SysTools dates or TDateTime dates, it makes absolutely no difference on the client side. Both dates appear as TDateTime dates to the client application.

In the client, the date field objects are created by the VCL to access date fields in the current record of your table. A date field object is an instance of the TDateField class. This class defines a property called AsDateTime to retrieve and set the value of the underlying record's date field. The type of this property is always TDateTime. To read the value in a date field, you would read the AsDateTime property into a TDateTime variable. To change the value in a date field you would write a TDateTime value to the AsDateTime property. In other words, you would completely ignore how dates were physically stored and would always assume that they're TDateTime values, and just use that type in your client application. Behind the scenes, the FlashFiler client code will convert the physical representation of a date into a (logical) TDateTime value, and vice versa, so you don't have to worry about it. As far as your client code is concerned, dates are always TDateTime values.

The problems programmers run into with date fields especially, is to assume that if the physical field type is a SysTools date, the client code should use a SysTools date variable. This is not so. You should always use TDateTime variables and values with a TDateField object. Unfortunately, this is a common mistake.

Let's look at an example that demonstrates the point. StartDate and EndDate are both defined as SysTools Dates in MyTable.

```
{Convert edtStart.Text and edtEnd.Text to TDateTimes and
 assign to their corresponding fields in MyTable.}
MyTable.Insert;
MyTable.FieldName('StartDate').AsDateTime :=
    StrToDateTime(edtStart.Text);
MyTable.FieldName('EndDate').AsDateTime  :=
    StrToDateTime(edtEnd.Text);
MyTable.Post;

{Retrieve StartDate and EndDate from MyTable and display
 in their corresponding labels.}
lblStStart.Text :=
    DateToStr(MyTable.FieldName('StartDate').AsDateTime);
lblStEnd.Text   :=
    DateToStr(MyTable.FieldName('EndDate').AsDateTime);
```

# Converting Data Types

When you redefine a FlashFile table, you can choose to save the existing data in the table. When this is done, you must provide a field map indicating which old fields should be mapped to which new fields. However, not all physical data types can be converted to all other data types. Table 16.4 lists the valid physical data type conversions.

**Table 16.4:** *Data type conversions.*

Data Type	Can Be Converted To
fftByte	fftWord16, fftWord32, fftInt8, fftInt16, fftInt32, fftSingle, fftDouble, fftExtended, fftComp, fftCurrency, fftAutoInc
fftWord16	fftWord32, fftInt32, fftSingle, fftDouble, fftExtended, fftComp, fftCurrency, fftAutoInc
fftWord32	fftSingle, fftDouble, fftExtended, fftComp, fftCurrency, fftAutoInc
fftInt8	fftInt16, fftInt32, fftSingle, fftDouble, fftExtended, fftComp, fftCurrency
fftInt16	fftInt32, fftSingle, fftDouble, fftExtended, fftComp, fftCurrency
fftInt32	fftSingle, fftDouble, fftExtended, fftComp, fftCurrency
fftSingle	fftDouble, fftExtended, fftCurrency
fftDouble	fftExtended, fftCurrency
fftExtended	fftCurrency
fftComp	Cannot convert to any other datatype
fftChar	fftShortString, fftShortAnsiString, fftNullString, fftNullAnsiString, fftWideChar, fftWideString
fftShortString	fftChar, fftShortAnsiString, fftNullString, fftNullAnsiString, fftWideChar, fftWideString
fftShortAnsiString	fftChar, fftShortString, fftNullString, fftNullAnsiString, fftWideChar, fftWideString
fftNullString	fftChar, fftShortString, fftShortAnsiString, fftNullAnsiString, fftWideChar, fftWideString
fftNullAnsiString	fftChar, fftShortString, fftShortAnsiString, fftNullString, fftWideChar, fftWideString

**Table 16.4:** *Data type conversions. (continued)*

<b>Data Type</b>	<b>Can Be Converted To</b>
<code>fftWideChar</code>	<code>fftChar</code> , <code>fftShortString</code> , <code>fftShortAnsiString</code> , <code>fftNullString</code> , <code>fftNullAnsiString</code> , <code>fftWideString</code>
<code>fftWideString</code>	<code>fftChar</code> , <code>fftShortString</code> , <code>fftShortAnsiString</code> , <code>fftNullString</code> , <code>fftNullAnsiString</code> , <code>fftWideChar</code>
<code>fftDateTime</code>	<code>fftStDate</code> , <code>fftStTime</code>
<code>fftStDate</code>	<code>fftDateTime</code>
<code>fftStTime</code>	<code>fftDateTime</code>
<code>fftAutoInc</code>	<code>fftWord32</code> , <code>fftSingle</code> , <code>fftDouble</code> , <code>fftExtended</code> , <code>fftComp</code> , <code>fftCurrency</code>
<code>fftBoolean</code>	<code>fftByte</code> , <code>fftWord16</code> , <code>fftWord32</code> , <code>fftInt8</code> , <code>fftInt16</code> , <code>fftInt32</code> (the result is 1 for True or 0 for False)  <code>fftChar</code> , <code>fftShortString</code> , <code>fftShortAnsiString</code> , <code>fftNullString</code> , <code>fftNullAnsiString</code> , <code>fftWideChar</code> , <code>fftWideString</code> (the result is "Y" for True or "N" for False)
<code>fftByteArray</code>	<code>fftBLOB</code> , <code>fftBLOBMemo</code> , <code>fftBLOBTypedBin</code> , <code>fftBLOBOLEObj</code> , <code>fftBLOBFmtMemo</code> , <code>fftBLOBDBSOLEObj</code> , <code>fftBLOBGraphic</code>
<code>fftCurrency</code>	<code>fftExtended</code>
<code>fftBLOB</code>	<code>fftBLOB</code> , <code>fftBLOBMemo</code> , <code>fftBLOBTypedBin</code> , <code>fftBLOBOLEObj</code> , <code>fftBLOBFmtMemo</code> , <code>fftBLOBDBSOLEObj</code> , <code>fftBLOBGraphic</code>
<code>fftBLOBMemo</code>	<code>fftBLOB</code> , <code>fftBLOBMemo</code> , <code>fftBLOBTypedBin</code> , <code>fftBLOBOLEObj</code> , <code>fftBLOBFmtMemo</code> , <code>fftBLOBDBSOLEObj</code> , <code>fftBLOBGraphic</code>
<code>fftBLOBTypedBin</code>	<code>fftBLOB</code> , <code>fftBLOBMemo</code> , <code>fftBLOBTypedBin</code> , <code>fftBLOBOLEObj</code> , <code>fftBLOBFmtMemo</code> , <code>fftBLOBDBSOLEObj</code> , <code>fftBLOBGraphic</code>
<code>fftBLOBFmtMemo</code>	<code>fftBLOB</code> , <code>fftBLOBMemo</code> , <code>fftBLOBTypedBin</code> , <code>fftBLOBOLEObj</code> , <code>fftBLOBFmtMemo</code> , <code>fftBLOBDBSOLEObj</code> , <code>fftBLOBGraphic</code>

**Table 16.4:** *Data type conversions. (continued)*

<b>Data Type</b>	<b>Can Be Converted To</b>
<code>fftBLOBOLEObj</code>	<code>fftBLOB</code> , <code>fftBLOBMemo</code> , <code>fftBLOBTypedBin</code> , <code>fftBLOBOLEObj</code> , <code>fftBLOBFmtMemo</code> , <code>fftBLOBDBSOLEObj</code> , <code>fftBLOBGraphic</code>
<code>fftBLOBDBSOLEObj</code>	<code>fftBLOB</code> , <code>fftBLOBMemo</code> , <code>fftBLOBTypedBin</code> , <code>fftBLOBOLEObj</code> , <code>fftBLOBFmtMemo</code> , <code>fftBLOBDBSOLEObj</code> , <code>fftBLOBGraphic</code>
<code>fftBLOBGraphic</code>	<code>fftBLOB</code> , <code>fftBLOBMemo</code> , <code>fftBLOBTypedBin</code> , <code>fftBLOBOLEObj</code> , <code>fftBLOBFmtMemo</code> , <code>fftBLOBDBSOLEObj</code> , <code>fftBLOBGraphic</code>

---

# Importing Data into FlashFiler Tables

The FlashFiler Explorer import function (see “Importing Data into FlashFiler Tables” on page 534) allows you to import data from fixed ASCII, typed binary, or B-Tree Filer files into FlashFiler tables. You can import into an existing FlashFiler table or have FlashFiler automatically create a new table based on the source file’s structure. A schema file is required so that FlashFiler knows how to interpret the layout of the data file being imported. Schema files must have the same name as the import data file, but with the extension SCH.

Details of data conversion errors encountered or records skipped are written to a log file. The log file is created in the same directory as the import file, with the same filename, and with the extension LOG.

If you have an existing BDE table and need to produce a fixed ASCII file suitable for import, use BDE Export to ASCII (BETA). BETA is a data export utility supplied with FlashFiler. It allows you to create a fixed ASCII data file from any supported BDE table. BETA also automatically generates the schema file to accompany the exported data. BETA is located in FlashFiler’s BETA subdirectory.

FlashFiler also includes a complete conversion program to convert any BDE table to a FlashFiler table. The utility, for which source is provided, is called BDE2FF (short for “BDE to FlashFiler”), can be found in FlashFiler’s BDE2FF folder.

If you need to import data from a FlashFiler 1 table, please refer to Chapter 15, “FlashFiler Data Conversion.”

## The schema file

To import a data file into FlashFiler, you must provide a schema file, which defines the layout of the data file. A schema file is a plain text file similar to a Windows INI file.

FlashFiler schema files are based on the schema files originally defined by Delphi to support ASCII text files as tables. The Delphi schema files are documented in the Borland Database Engine Online Help as ASCII.DRV.TXT. FlashFiler adds additional keywords and qualifiers to this basic schema file concept. Delphi and C++Builder schema files support data files with variable length records (delimited ASCII). However, FlashFiler supports only data files with fixed length records.

Following is an example schema file:

```
[employee]
FILETYPE=ASCII
DATEMASK=M/D/Y H:M:S T
FIELD1=EMP_NO, NUMBER, 6, 0, 0
FIELD2=FIRST_NAME, CHAR, 15, 0, 6
FIELD3=LAST_NAME, CHAR, 20, 0, 21
FIELD4=PHONE_EXT, CHAR, 4, 0, 41
FIELD5=HIRE_DATE, TIMESTAMP, 30, 0, 45
FIELD6=BIRTH_DATE, DATE, 6, 0, 75, YYMMDD
FIELD7=DEPT_NO, CHAR, 3, 0, 81
FIELD8=JOB_CODE, CHAR, 5, 0, 84
FIELD9=JOB_GRADE, NUMBER, 6, 0, 89
FIELD10=JOB_COUNTRY, CHAR, 15, 0, 95
FIELD11=SALARY, FLOAT, 20, 2, 110
FIELD12=FULL_NAME, CHAR, 37, 0, 130
```

The first line contains the name of the data file (without extension) in brackets. This value is required. It is followed by one or more “keyword = value” pairs, each on a separate line. The valid keywords are FILETYPE, RECLENGTH, DATEMASK, and FIELD. The RECLENGTH and DATEMASK keywords are FlashFiler extensions to the Delphi schema file format.

### The FILETYPE keyword

The FILETYPE keyword indicates which type of file will be imported. Use FILETYPE=ASCII for files of fixed-length ASCII records, FILETYPE=BINARY for files of fixed-length typed binary records, or FILETYPE=BTF for files of fixed-length B-Tree Filer records.

All fields in an ASCII import file contain a string of characters denoting the field's value. Typed binary files can contain strings for character data, but can also contain binary encoded values for fields such as numeric, dates, and currency. Delphi files declared as FILE OF RECORD are considered typed binary files, but any file with fixed-length records and data values that can be translated into FlashFiler data types can be used.

## The RECLENGTH keyword

RECLENGTH is the total number of bytes in the record. It is required for typed binary import files, optional for ASCII import files, and ignored for BTree Filer import files (the record length is determined automatically from the B-Tree Filer header).

If RECLENGTH is not defined for an ASCII import file, the record size is computed from the last field descriptor and a terminating CRLF is assumed. RECLENGTH is required if records are not terminated by CRLE, or if the last field descriptor does not extend to the end of the record (i.e., some fields are being skipped).

The RECLENGTH keyword is a FlashFiler extension to the Delphi schema file format.

## The DATEMASK keyword

Since date and time values stored as ASCII text can be formatted a number of different ways, you must specify the format. The DATEMASK keyword provides a “picture mask” that describes the format of all date and time string fields. Table 16.5 illustrates the result of sample date masks.

**Table 16.5:** *Formatting date and time values.*

Input Data	Date Mask
3/11/97	M/D/Y
970311	YYMMDD
11.3.97	D.M.Y
3/11/97 6:04:22 PM	M/D/Y h:m:s t
3-11-1997 18:04:22.552	M-D-Y h:m:s
11-3-1997 18.04.22	D-M-Y h.m.s

The default date mask is M/D/Y h:m:s t.

Date masks consist of formatting characters and delimiters. Formatting characters, such as ‘Y’, ‘M’, and ‘D’, indicate where to find each element of the field (year, month, or day). Delimiters, like ‘/’, ‘:’, or ‘-’, are used as separators. Delimiter characters are used exactly as specified in the date mask; there is no conversion to international settings on the host machine, since what matters is how they are stored in the import file.

All date formatting characters are uppercase and all time formatting characters are lower case. Table 16.6 defines the characters that can be used as formatting characters in a date mask.

**Table 16.6:** *Date mask characters.*

Char	Marks Location Of
D	Day
M	Month
Y	Year
h	Hours
m	Minutes
s	Seconds
t	AM or PM, for example

In most cases, only a single formatting character of each type is necessary. For example, 'M/D/Y' is sufficient. 'MM/DD/YYYY' is not necessary, but doesn't hurt. All non-white space characters between the delimiters are accepted as data for that element of the field. The number of formatting characters is crucial only when there are no delimiters. For example, in a date mask of YYMMDD, the number of digits for each element of the date must be determined by the number of formatting characters in the date mask.

If a time value contains fractional seconds, for example 18:04:22.552, the fraction is dropped and only the whole number of seconds is saved when importing.

The date mask applies to all date and time fields in the import file. If a date mask contains both date and time components, and a field in the import file contains just a date or just a time, only the relevant part of the date mask is used for that field. If a field contains date or time data formatted differently than specified by DATEMASK, then a date mask that is valid only for that field can be added in its field descriptor (see "The FIELD Keyword" on page 538). The DATEMASK keyword is a FlashFiler extension to the Delphi schema file format.



## The FIELD keyword

The FIELD keyword provides a field descriptor for each field to import. When FlashFiler automatically creates a table (i.e., the data is not imported into an existing table), the fields are defined in sequence based on the FIELD keywords. A field in an import file can be skipped by omitting the field descriptor for it, or by providing a field name that does not match any field in the target FlashFiler table.

Field descriptors have the following format:

```
FIELDn=<field name>, <datatype>, <field width>,  
      <decimal places>, <offset> [,<date mask>]
```

where n is a unique number identifying the field descriptor.

Field name is the name as it appears in the FlashFiler table. Fields in the import file are matched to fields in the FlashFiler table by field name. Case is ignored when this match is performed. If the FlashFiler table is being automatically created, the field names are defined in the order and case in which they are specified in the schema file.

Data type is the data type of the field, as it will be imported into FlashFiler. The valid data types are described in the next section.

Field width is the number of bytes this field occupies in the import file.

Decimal places is the number of digits after the decimal point (for floating-point data types only).

Offset is the number of bytes from the beginning of the import record to the beginning of the field (zero-based).

Date mask is for DATE, TIME, or TIMESTAMP fields only and defines the format of the data. This date mask is required only if the format of the data differs from the format specified by the DATEMASK keyword (or the default date mask).

## Data types supported for import files

Table 16.7 shows the data types that are supported for ASCII import files.

**Table 16.7:** *Data types for ASCII import files.*

Keyword	Description	Compatible FlashFiler Data Types
CHAR	Character/string	fftChar, fftShortString, fftShortAnsiString, fftNullString, fftNullAnsiString, fftWideChar, fftWideString
FLOAT	Floating point (a string of characters denoting a floating-point value)	fftSingle, fftDouble, fftExtended, fftCurrency
NUMBER	Integer	fftByte, fftWord16, fftWord32, fftInt8, fftInt16, fftInt32, fftAutoInc
LONGINT	Long integer	fftWord32, fftInt32, fftComp, fftAutoInc
BOOL	Boolean (can be T/F, Y/N, 1/0)	fftBoolean
DATE	Date (format specified by DATEMASK)	fftDateTime, fftStDate
TIME	Time (format specified by DATEMASK)	fftDateTime, fftStTime
TIMESTAMP	Date and time (format specified by DATEMASK)	fftDateTime, fftStDate, fftStTime

Table 16.8 shows the data types that are supported for typed binary and B-Tree Filer import files.

**Table 16.8:** *Data types for binary/B-Tree Filer import files.*

Keyword	Description	Compatible FlashFiler Data Types
CHAR	Character (array of characters)	fftChar, fftShortString, fftShortAnsiString, fftNullString, fftNullAnsiString, fftWideString
FLOAT	Floating-point <sup>2</sup> (a binary-encoded floating point value)	fftSingle, fftDouble, fftExtended, fftCurrency
BOOL	Boolean (zero=False, nonzero=True)	fftBoolean
DATE	Date character string (format specified by DATEMASK)	fftDateTime, fftStDate
TIME	Time character string (format specified by DATEMASK)	fftDateTime, fftStTime
TIMESTAMP	Date and time character string (format specified by DATEMASK)	fftDateTime, fftStDate, fftStTime
STRING <sup>1</sup>	Length-byte string	fftShortString, fftShortAnsiString, fftNullString, fftNullAnsiString, fftWideString
ASCIIIZ <sup>1</sup>	Null-terminated string	fftShortString, fftShortAnsiString, fftNullString, fftNullAnsiString, fftWideString
UNICODE <sup>1</sup>	UNICODE (16-bit) char (if source length is > 1, then a null-terminated string of WideChars is assumed)	fftWideChar, fftWideString
INTEGER <sup>1</sup>	Signed integer <sup>2</sup>	fftInt8, fftInt16, fftInt32, fftSingle, fftDouble, fftExtended, fftComp, fftCurrency

**Table 16.8:** *Data types for binary/B-Tree Filer import files. (continued)*

Keyword	Description	Compatible FlashFiler Data Types
UINTINTEGER <sup>1</sup>	Unsigned integer <sup>2</sup>	fftByte, fftWord16, fftWord32, fftInt8, fftInt16, fftInt32, fftSingle, fftDouble, fftExtended, fftComp, fftCurrency, fftAutoInc
CURRENCY	Delphi currency field	fftExtended, fftCurrency
DATETIME1 <sup>1</sup>	Delphi TDateTime field (Delphi 1)	fftDateTime, fftStDate, fftStTime
DATETIME2 <sup>1</sup>	Delphi TDateTime field (Delphi 2 or later)	fftDateTime, fftStDate, fftStTime
STDATE <sup>1</sup>	SysTools date <sup>3</sup>	fftStDate, fftDateTime
STTIME <sup>1</sup>	SysTools date <sup>3</sup>	fftStTime, fftDateTime
BINARY <sup>1</sup>	Arbitrary binary data (no translation)	fftByteArray, any BLOB type

<sup>1</sup>These data types are FlashFiler extensions to the Delphi schema data types.

<sup>2</sup>The size of the source field determines the actual source data type. For example, a six- byte FLOAT field is interpreted as a Delphi Real data type.

<sup>3</sup>Date or time field from another TurboPower product. It could be a TStDate or TStTime from SysTools or a WsDate or WsTime from Win/Sys Library or Data Entry Workshop.

## File import example

The following example shows a B-Tree Filer record definition and its corresponding schema definition. Notice that B-Tree Filer's "delete flag" field is denoted in the schema file. If the first field in the record is defined as a 4-byte integer called "DELFLAG", then it is assumed to be a B-Tree Filer delete flag field and all records with a nonzero value in this field are skipped. In addition, a field is not created for the delete flag if a new FlashFiler database is created.

```
TMyRecord = record
  DelFlag      : Longint;
  Name         : string[25];
  CustNum      : Longint;
  OrderDate    : array[1..6] of Char; { YYMMDD }
  OrderTotal   : Double;
  OrderNum     : DWORD;
  DetailItems  : Word;
  Comments     : array[0..500] of Char; { null-terminated string }
end;

[MyTable]
FILETYPE=BTF
DATEMASK=YYMMDD
FIELD1=DelFlag, INTEGER, 4, 0, 0
FIELD2=Name, STRING, 26, 0, 4
FIELD3=CustNum, INTEGER, 4, 0, 30
FIELD4=OrderDate, DATE, 6, 0, 34
FIELD5=OrderTotal, FLOAT, 8, 2, 40
FIELD6=OrderNum, UINTEGER, 4, 0, 48
FIELD7=DetailItems, UINTEGER, 2, 0, 52
FIELD8=Comments, ASCIIZ, 501, 0, 54
```

---

## User-Defined Indexes

FlashFiler allows you to extend the standard database-style indexes, where keys are built on one or more fields that exist in a record. User-defined indexes provide you with the opportunity to reach into the FlashFiler indexing mechanism and alter it for your own purposes.

A classic case that lends itself to the use of a user-defined index is that of a Soundex key. Soundex is an algorithm for generating a phonetic string that represents an English name. Assume that you have a Soundex algorithm. To implement a Soundex key, you could expand the record layout to include a Soundex value field, along with the name field from which it was calculated. You could easily build an index on this new field. However, there are drawbacks to this scheme. Even though the record size increased, no more useful information was added to the record (in fact, it is more of a duplication of information). Over several thousand, or tens of thousands of records, that is a large amount of wasted space. The second drawback is that you must ensure that the Soundex field is recalculated every time the name field changes.

The alternative is a user-defined index. It provides a way for you to write your own key creation and key comparison routines, register them with FlashFiler, and have them used for a specified table and index. In this example, you would write a routine to calculate the Soundex value as a key and a routine to compare two Soundex values. FlashFiler takes care of the rest. When FlashFiler needs the key for a record, it calls your build key routine. When FlashFiler needs to compare two keys, it calls your key comparison routine. This creates an index that requires no extra space in the record and one which does not require you to ensure that special values are consistently recalculated.

Your two routines must be compiled into a DLL, and the DLL must be registered with FlashFiler via the Config | User-defined indexes menu option. See “Config | User-Defined Indexes...” on page 41 for a description of this menu option.

The standard FlashFiler interface unit for user-defined indexes is FFSRINTF. This unit declares a set of standard comparison routines for you to use, and also the type declarations for the build and compare key routines. Here is that declaration:

```
TffKeyCompareFunc = function(const Key1, Key2;  
    aData : PfffCompareData) : Integer
```

Key1 and Key2 are the two keys to be compared. aData is the pointer to TffCompareData structure that defines how the comparison between the two keys is made. The result is less than zero if Key1 is less than Key2, zero if Key1 equals Key2, or greater than zero if Key1 is greater than Key2. The declaration of TffCompareData is as follows:

```
type
  PffCompareData = ^TffCompareData;
  TffCompareData = packed record {Data for comparison operations}
    cdKeyLen   : Longint;           {..max length of key to compare}
    cdDict     : Pointer;           {..dictionary (to be typecast)}
    cdIndex    : Longint;           {..index number}
    cdFldCnt   : Longint;           {..field count (partial
                                    searches)}
    cdPartLen  : Longint;           {..partial length (partial
                                    searches)}
    cdAscend   : Boolean;           {..True if keys are to be
                                    compared in ascending order}
  end;
```

cdKeyLen is the length of the key that was defined when the table was built. cdIndex is the index number of the table for which the comparison is made. You can arrange your code so that one routine performs comparisons for many user-defined indexes for the same table. However, there is no provision for finding out which table it is, so don't use the comparison routine for different tables. If cdPartLen is zero, you can compare the keys using all of the bytes in the keys. If cdPartLen is non-zero, you must compare the keys using only the first cdPartLen bytes of the keys. cdAscend is True if the comparison is to be made so as to ensure an ascending key, False to provide a descending key. The other fields are only used internally. The declaration for the build routine looks like this:

```
TffKeyBuildFunc = function(Index : Integer;
  DataRec : PffByteArray;
  var Key; KeyLen : Integer) : Boolean
```

Index is the index number of the table for which the key is generated. You can arrange your code so that one routine builds keys for many user-defined indexes for the same table. However, there is no provision for finding out which table it is, so don't use the same build routine for different tables. DataRec is a pointer to the physical record. You must know the record layout (it can't be determined at run time). Key is where the key should be generated and KeyLen is the key length of the index. Return True if a key was generated, or False otherwise.

**Note:** For a given record, your build key routine must generate only one possible key. If your routine can sometimes generate a different key, FlashFiler Server cannot use the user-defined index properly, resulting in missing records or even system crashes.

The following lines of source code provide an example of a User-Defined Index DLL:

```

library UserIdx;

uses
    FFLBase,
    FFSrIntf;

function CompareUppercase(const Key1, Key2;
aData : PffCompareData) : Integer; stdcall;

var
    SS1 : TffShStr absolute Key1;
    SS2 : TffShStr absolute Key2;

begin
    with aData^ do begin
        {if a full field comparison is needed...}
        if (cdPartLen = 0) then

            {compare the two keys as short strings, up to
             cdKeyLen characters}
            Result := FFCheckDescend(aData^.cdAscend,
                                     FFCmpShStr(SS1,SS2,aData^.cdKeyLen))
        {otherwise a partial field comparison is required...}
        else {compare the two keys as short strings, up to cdPartLen
             characters}
            Result := FFCheckDescend(aData^.cdAscend,
                                     FFCmpShStr(SS1,
                                                  SS2,
                                                  aData^.cdPartLen));

        end;
    end;

function BuildUppercase(Index : Integer; DataRec : PffByteArray;
    var Key; KeyLen : Integer) : Boolean; stdcall;

var
    KeyStr : TffShStr absolute Key;

begin
    {we are working with the second field in the record which begins
     at offset 4 (zero based) of the data record)}
    KeyStr := FFShStrUpper(PffShStr(@DataRec[4])^);
    Result := True;
end;

exports
    CompareUppercase,
    BuildUppercase;

begin
end.

```



This index provides a case-insensitive ordering of records, based on the first field of the record. The first field is assumed to be a 32-byte short string. The two routines that fully describe this user-defined index are `BuildUppercase` and `CompareUppercase`. `BuildUppercase` builds the key for a record by returning the field, uppercased. `CompareUppercase` compares two keys by doing a standard non-case-sensitive string compare.

☛ **Caution:** The two routines must be declared with the `stdcall` identifier. Otherwise, system crashes may occur at run time.

---

# Client Configuration

FlashFiler clients need to know certain configuration information, like protocol and server name, in order to connect to a FlashFiler Server. FlashFiler stores these settings in the registry. The values are stored in \\HKEY\_LOCAL\_MACHINE\Software\TurboPower\FlashFiler2.01\Client Configuration. This section describes the FFCLCFG unit which includes routines for retrieving and storing these values. FlashFiler also includes a utility called FFCOMMS to ease the setting of the default protocol and, optionally, the default server name.

Two string values may be queried during the initialization of a client application: “Protocol” and “Server Name”. The Protocol string contains a string representation of the selected protocol. Legitimate values are “Single User”, “TCP/IP”, and “IPX/SPX”.

The server name string contains the exact server name as it appears in the FlashFiler Server main window. This value is optional, however, it is useful when multiple FlashFiler servers are available on a network, or when the use of a router prevents a server from responding to broadcasts by a client.

**Note:** All methods begin with “FF”.

## Methods

...ClientConfigGetProtocolName	...ClientConfigReadProtocol	...ClientConfigWriteProtocolName
...ClientConfigGetProtocolNames	...ClientConfigReadProtocolClass	...ClientConfigWriteServerName
...ClientConfigOverrideProtocol	...ClientConfigReadServerName	
...ClientConfigOverrideServerName	...ClientConfigWriteProtocolClass	

# Reference Section

FFClientConfigGetProtocolName

function

```
function FFClientConfigGetProtocolName(  
    aProtocol : TffCommsProtocolClass) : TffShStr;
```

↳ Returns the name for the given protocol.

This routine returns the string representation of the TffCommsProtocolClass specified in aProtocol. The following table lists the protocol classes and their string representations:

TffCommsProtocolClass	Protocol Name
TffSingleUserProtocol	Single User
TffTCPIPProtocol	TCP/IP
TffIPXProtocol	IPX/SPX

The TffCommsProtocolClass type is defined in FLLProt.

See also: FFClientConfigGetProtocolNames, FFClientConfigReadProtocol, FFClientConfigReadProtocolClass, FFClientConfigWriteProtocolClass, FFClientConfigWriteProtocolName

FFClientConfigGetProtocolNames

procedure

```
procedure FFClientConfigGetProtocolNames(aNames : TStrings);
```

↳ Returns a list of protocol names.

This routine fills a TStrings descendent with valid protocol names. An existing TStrings descendent, aName, is required. The strings object will be cleared before the protocol names are added to the list.

The following code displays the protocol names in a TMemo control named Memo1:

```
FFClientConfigGetProtocolNames(Memo1.Lines);
```

See also: FFClientConfigGetProtocolName, FFClientConfigReadProtocol, FFClientConfigReadProtocolClass, FFClientConfigWriteProtocolClass, FFClientConfigWriteProtocolName

```
procedure FFClientConfigOverrideProtocol(  
    aProtocol : TffCommsProtocolClass);
```

↪ Overrides the stored protocol.

This routine will temporarily override the Protocol setting in the registry with the protocol specified by aProtocol. aProtocol is an instance of TffCommsProtocol class. Any TffCommsEngine with a protocol setting of ptRegistry that is made active after a call to this procedure will use the overridden protocol. Any TffClient made active, that is not directly connected to a TffRemoteServerEngine, will also use the specified override protocol. The override stays in effect until the client application is closed, or until the procedure is called with nil for aProtocol.

See also: FFClientConfigReadProtocol, FFClientConfigWriteProtocolClass

```
procedure FFClientConfigOverrideServerName(  
    const aServerName : TffNetAddress);
```

↪ Overrides the stored server name.

This routine will temporarily override the Server Name setting in the registry with the server name and address specified in TffNetAddress. Any TffCommsEngine that is made active after a call to this procedure will use the overridden server name. Any TffClient made active, that is not directly connected to a TffRemoteServerEngine, will also use the specified overridden server name. The override stays in effect until the client application is closed, or until the procedure is called with an empty string for aServerName.

See also: FFClientConfigReadServerName, FFClientConfigWriteServerName

```
procedure FFClientConfigReadProtocol(  
    var aProtocol : TffCommsProtocolClass;  
    var aProtocolName : TffShStr);
```

↳ Returns the current protocol name and class.

This routine will return the configured protocol class, and protocol name as defined in the client configuration. The protocol class is returned in parameter aProtocol. The protocol name is returned in parameter aProtocolName. By default, the configuration will be read from the registry. If the information is not found in the registry, the compiled defaults will be returned. If a previous call was made to FFClientConfigOverrideProtocol, then the protocol specified in the override will be returned.

See also: FFClientConfigReadProtocolClass, FFClientConfigOverrideProtocol,  
FFClientConfigWriteProtocolClass, FFClientConfigWriteProtocolName

```
function FFClientConfigReadProtocolClass : TffCommsProtocolClass;
```

↳ Returns the current ProtocolClass.

This routine returns the configured protocol class as defined in the client configuration. By default, the configuration will be read from the registry. If the information is not found in the registry, then the compiled defaults will be returned. If a previous call was made to FFClientConfigOverrideProtocol, then the protocol specified in the override will be returned.

See also: FFClientConfigOverrideProtocol, FFClientConfigReadProtocol,  
FFClientConfigWriteProtocolClass

```
function FFClientConfigReadServerName : TffNetAddress;
```

↳ Returns the default server name.

This routine will return the configured ServerName as defined in the Client Configuration. By default, the configuration will be read from the registry. If the information is not found in the registry, then the compiled defaults will be returned. If a previous call was made to FFClientConfigOverrideServerName, then the server name specified in the override will be returned.

See also: FFClientConfigOverrideServerName, FFClientConfigWriteServerName

## **FFClientConfigWriteProtocolClass**

**procedure**

```
procedure FFClientConfigWriteProtocolClass(  
    aProtocol : TffCommsProtocolClass);
```

↪ Stores the specified protocol in the registry.

This routine will save the specified protocol class to the registry for later retrieval. aProtocol is the instance of TffCommsProtocol class saved to the registry.

See also: FFClientConfigOverrideProtocol, FFClientConfigWriteProtocolName

## **FFClientConfigWriteProtocolName**

**procedure**

```
procedure FFClientConfigWriteProtocolName(  
    aProtocolName : TffShStr);
```

↪ Writing the specified protocol to the registry.

This routine will save the specified protocol name to the registry for later retrieval. aProtocolName is one of the following text strings: “Single User”, “TCP/IP”, “IPX/SPX”.

See also: FFClientConfigOverrideProtocol, FFClientConfigReadProtocolName

## **FFClientConfigWriteServerName**

**procedure**

```
procedure FFClientConfigWriteServerName(  
    aServerName : TffNetAddress);
```

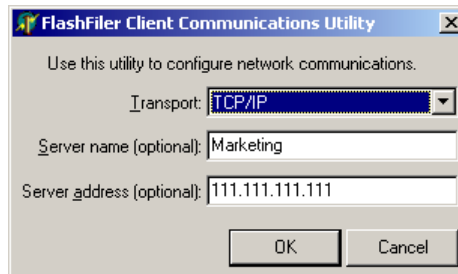
↪ Stores the specified server name in the registry.

This routine will save the specified server name to the registry for later retrieval. aServerName is the server name and address to be saved to the registry.

See also: FFClientConfigOverrideServerName, FFClientConfigWriteProtocolClass

## FlashFiler Client Communications Utility

Use the FlashFiler Client Communication Utility (FFCOMMS for short) to select the network protocol and optional server name on a client workstation. The main window of FFCOMMS is shown in Figure 16.1. Although the use of this program is not required, users have found that they get more consistent connections to the server if they initially set the configuration using this program. These settings define the communication parameters that FlashFiler Client applications use to connect to a server. The settings are, by default, used by all FlashFiler clients, including custom applications, FlashFiler tools like FlashFiler Explorer, the Delphi and C++Builder IDEs, and the FlashFiler Crystal Reports driver. The FFCOMMS utility stores the protocol name in the registry. The FlashFiler tools, and the IDE use this information to determine how to connect to the FlashFiler server.



*Figure 16.1: The Client Communications Utility main window.*

Transport is the default transport to save to the registry. Server Name and Server Address are optional. Server Name is the name of the server (e.g., Production, Test) to which client applications should connect. Server Address is the server's network address (e.g., 10.0.0.1 for TCP/IP). If your FlashFiler Server is not configured to listen for broadcast messages, you must specify a Server Name and Server Address.

---

# FlashFiler-Specific Routines (FFCLIntf)

There are certain functions in FlashFiler that don't fit easily into the VCL style. For example, FlashFiler tables have more structure than the standard CreateTable method was designed to handle. The FlashFiler Client API was written to fill in the gaps in these areas by providing new or expanded methods to supersede their VCL cousins.

The next section, Batched Record Routines, contains more FlashFiler-specific routines.

⚠ **Caution:** All of the FlashFiler-specific routines will be phased out in future versions of FlashFiler. The functionality of these routines is now included as a method of an appropriate component. We have included them in this release for backwards compatibility. The reference for each routine lists its preferred alternative.

## Methods

FFDbiAddAlias	FFDbiGetTaskStatus	FFDbiSetFailSafeTransaction
FFDbiAddFileBLOB	FFDbiOverrideFilter	FFDbiSetFilter
FFDbiAddIndex	FFDbiPackTable	FFDbiSetLoginParameters
FFDbiCreateTable	FFDbiReindexTable	FFDbiSetLoginRetries
FFDbiDeleteAlias	FFDbiRestoreFilter	FFDbiSetProtocol
FFDbiGetServerDateTime	FFDbiRestructureTable	FFDbiSetTableAutoIncValue



## Reference Section

### FFDbiAddAlias

function

```
function FFDbiAddAlias(  
    aSession : TffSession; const aAlias : TffName;  
    const aPath : TffPath) : TffResult;  
  
TffName = string[ffcl_GeneralNameSize];  
  
ffcl_GeneralNameSize = 31;  
  
TffPath = string[ffcl_Path];  
  
ffcl_Path = 219;
```

🔗 Creates a new permanent alias at the server.

aSession is the session component you are using. aAlias refers to the name of the alias you wish to create. If the alias name already exists on the server, the FFDbiAddAlias function will fail. aPath refers to a path on the server machine or to a UNC path that the server machine can access. For example, specifying C:\DATABASE as the path will create an alias that points to the server's C: drive, not the client's. To create an alias pointing to the client machine's C: drive, you must specify aPath with a UNC path (i.e. \\ClientMachine\Share\Database).

⚠️ **Caution:** This routine will be phased out in future versions of FlashFiler. In preparation, please use the AddAlias method of the TffSession component (see page 149) in place of this routine.

See also: FFDbiDeleteAlias

```
function FFDbiAddFileBLOB(aTable : TffDataSet; const iField : Word;  
    const aFileName : TffFullFileName): TffResult;  
  
    TffFullFileName = string[255];
```

↪ Inserts a reference to a file into a BLOB field.

FileBLOBs are FlashFiler's unique way of storing large quantities of data in minimal space. FileBLOBs are useful when a single file on the server contains data you would like to keep in a database, but without replicating the data in your BLOB file.

aTable is the table that will store the fileBLOB. iField references the number of the BLOB field. Fields are numbered starting at 1. aFileName is a string containing the full server relative path to the BLOB data file.

● **Caution:** This routine will be phased out in future versions of FlashFiler. In preparation, please use the AddFileBlob method of the TffTable component (see page 194) in place of this routine.

```
function FFDbiAddIndex(
  aTable : TffBaseTable; const aIndexDesc : TffIndexDescriptor;
  var aTaskID : Longint) : TffResult;

TffIndexDescriptor = packed record
  idNumber : Longint;
  idName : TffDictItemName;
  idDesc : TffDictItemDesc;
  idFile : Longint;
  idKeyLen : Longint;
  idCount : Longint;
  idFields : TffFieldList;
  idDups : Boolean;
  idAscend : Boolean;
  idNoCase : Boolean;
end;
```

↳ Adds an index to an existing FlashFiler table.

FFDbiAddIndex adds an index to the dictionary in the specified table, and then asks the server to create and populate the physical index. Since this process may take some time, the server starts an asynchronous task and returns its task ID immediately. You can query this task ID with the FFDbiGetTaskStatus routine to find out the progress of the add index operation.

aTable is the table to receive the new index. aIndexDesc refers to the record containing the new index information. aTaskID is returned by the function, and must be used when you make a call to FFDbiGetTaskStatus.

FFDbiAddIndex initiates the restructure and then returns immediately. If you need to determine the progress of the restructure operation, use the returned aTaskID and call FFDbiGetTaskStatus.

☛ **Caution:** This routine will be phased out in future versions of FlashFiler. In preparation, please use the AddIndex method of the TffTable component (see page 234) in place of this function.

```
function FFDbiCreateTable(aDatabase : TffDatabase;
    const aOverWrite : Boolean; const aTableName : TffTableName;
    aDictionary : TffDataDictionary) : TffResult;

TffTableName = string[ffcl_TableNameSize];
ffcl_TableNameSize = 31;
```

↪ Creates a table in a FlashFiler database.

FFDbiCreateTable creates a new table named aTableName with a structure defined by aDictionary in aDatabase. aOverwrite defines whether the newly created table should replace an existing table with the same name. aDictionary parameter refers to a TffDataDictionary object that contains the structure information for the new table. Documentation of the TffDataDictionary can be found in the “TffDataDictionary Class” section on page 572.

⚠ **Caution:** This routine will be phased out in future versions of FlashFiler. In preparation, please use the CreateTable method of the TffDatabase component (see page 181) in place of this function.

```
function FFDbiDeleteAlias(aSession : TffSession;
    const aAlias : TffName) : TffResult;

TffName = string[ffcl_GeneralNameSize];
ffcl_GeneralNameSize = 31;
```

↪ Permanently deletes an existing alias at the server.

aSession is the session component you are using. aAlias refers to the name of the alias you wish to delete. If the alias name does not exist on the server, DBIERR\_UNKNOWNDB is returned.

⚠ **Caution:** This routine will be phased out in future versions of FlashFiler. In preparation, please use the DeleteAliasEx method of the TffSession component (see page 154) in place of this routine.

```
function FFDbiGetServerDateTime(aSession : TffSession;  
    var aServerNow : TDateTime) : TffResult;
```

↳ Retrieves the current date and time values from the server.


This routine allows a client application to retrieve the current date and time from the FlashFiler server.

aSession refers to the session component you are using. The date and time of the server's machine are returned in the aServerNow parameter. This function does not respect any client time zone settings. If the server and client are in different time zones, the client will be responsible for the conversion.

⚠ **Caution:** This function will be phased out in future versions of FlashFiler. In preparation, please use the GetServerDateTime method of the TffSession component (see page 157) in place of this routine.

```
function FFDbiGetTaskStatus(aSession : TffSession;
    const aTaskID : Longint; var aCompleted : Boolean;
    var aStatus : TffRebuildStatus) : TffResult;
```


```
TffRebuildStatus = packed record
    rsStartTime : Longint;
    rsTotalRecs : Longint;
    rsRecsRead : Longint;
    rsRecsWritten : Longint;
    rsPercentDone : Longint;
    rsErrorCode : TffResult;
    rsFinished : Boolean;
end;
```

 Returns information about the progress of the current rebuild task.

When a client program initiates a pack, reindex, or restructure of a table, the task can take quite some time if the table is large. These rebuild tasks are done entirely at the server and control returns to the client application immediately when the task starts. This leaves the client free to perform other processing during the rebuild operation. However, there is no spontaneous notification from the server when the rebuild task completes. The client can use `FFDbiGetTaskStatus` to periodically check the status of the rebuild task.

`aSession` is the session component being used. `aTaskID` identifies the rebuild task. This is the value that was returned by the function that started the rebuild task (i.e., `FFDbiPackTable`, `FFDbiReindexTable`, or `FFDbiRestructureTable`). `aCompleted` is `True` if the task is complete. `aStatus` contains information about the current progress of the rebuild task. `aStatus` is invalid if the task is complete.

There is a difference between the `aCompleted` parameter and the `rsFinished` field of the `aStatus` record. When you poll the task status with this routine, you will eventually get a status with `aCompleted` set to `False` and `rsFinished` set to `True` (`rsPercentDone` will equal 100). When the server sends back this finished status record, it takes the opportunity to remove the task ID from its list of tasks. If you then poll the task status again, the server cannot find the task ID. Rather than return an error code, it just sets `Completed` to `True` (the task ID can't be found, so the task must have completed). Task IDs are only removed in two situations: when the client detaches from a server in which case all of its tasks are removed, and when the client polls the task status and the status is marked as finished.

 **Caution:** This routine will be phased out in future versions of FlashFiler. In preparation, please use the `GetTaskStatus` method of the `TffSession` component (see page 158) in place of this routine.

See also: `FFDbiPackTable`, `FFDbiReindexTable`, `FFDbiRestructureTable`

```
function FFDbiOverrideFilter(aTable : TffDataSet;
    aExprTree : pCANExpr; aTimeOut : TffWord32) : TffResult;

pCANExpr = ^CANExpr;

CANExpr = packed record
    iVer : Word;
    iTotalSize : Word;
    iNodes : Word;
    iNodeStart : Word;
    iLiteralStart : Word;
end;
```

↪ Overrides a cursor's existing filter with a new filter.

This function is used internally to temporarily override a server-side filter. `aTable` is the table component with the filter to be overridden. `aExprTree` is the expression tree for the filter. `aTimeOut` the time (in milliseconds) allowed for this operation before it times out.

☛ **Caution:** This routine will be phased out in future versions of FlashFiler. Please avoid using it.

See also: `FFDbiRestoreFilter`, `FFDbiSetFilter`

```
function FFDbiPackTable(aDatabase : TffDatabase;
    const aTableName : TffTableName;
    var aTaskID : Longint) : TffResult;

TffTableName = string[ffcl_TableNameSize];

ffcl_TableNameSize = 31;
```

↪ Frees up unused space in a table by removing deleted records.

After a pack, the table is not necessarily smaller. The table size is a multiple of the block size, so if there were fewer deleted records than will fit in a block, the table size is unaffected.

aDatabase is the database component connected to the table you want to pack. aTableName is the name of the table to pack. aTaskID is returned by the function, and must be used when you make a call to FFDbiGetTaskStatus.

FFDbiPackTable initiates the pack operation and then returns immediately. If you need to determine the progress of the pack operation, use the returned aTaskID and call FFDbiGetTaskStatus.



**Caution:** This routine will be phased out in future versions of FlashFiler. In preparation, please use the PackTable method of the TffDatabase component (see page 171) or the PackTable method of the TffTable component (see page 224) in place of this function.

See also: FFDbiGetTaskStatus

## FFDbiReindexTable

function

```
function FFDbiReindexTable(  
    aDatabase : TffDatabase; const aTableName : TffTableName;  
    const aIndexNum : Integer; var aTaskID : Longint) : TffResult;  
  
TffTableName = string[ffcl_TableNameSize];  
  
ffcl_TableNameSize = 31;
```



Reconstructs the key values for the specified index.i

FFDbiReindexTable reconstructs the key values for the index specified by aIndexNum in the table aTableName.

aDatabase is the database component used by the table being re-indexed. aTableName is the name of the table being re-indexed. aIndexNum parameter is 0 for the default Sequential Access Index, and is increased by an increment of 1 for each additional index defined for the table. aTaskID parameter is returned by the function, and must be used when you make a call to FFDbiGetTaskStatus.

FFDbiReindexTable initiates the reindex and then returns immediately. If you need to determine the progress of the index operation, use the returned aTaskID and call FFDbiGetTaskStatus.



**Caution:** This routine will be phased out in future versions of FlashFiler. In preparation, please use the ReindexTable method of the TffDatabase component (see page 172) or the ReindexTable method of the TffTable component (see page 243) in place of this routine.

See also: FFDbiGetTaskStatus



```
function FFDbiRestoreFilter(aTable : TffDataSet) : TffResult;
```

↪ Restores the server-side filter following a call to FFDbiOverrideFilter.

Used internally to restore a filter that was temporarily overridden. `aTable` is the table component with the filter to restore.

⚠ **Caution:** This function will be phased out in future versions of FlashFiler. Please avoid using it.

See also: FFDbiOverrideFilter, FFDbiSetFilter

```
function FFDbiRestructureTable(aDatabase : TffDatabase;
    const aTableName : TffTableName; aDictionary : TffDataDictionary;
    aFieldMap : TStrings; var aTaskID : Longint) : TffResult;

TffTableName = string[ffcl_TableNameSize];
ffcl_TableNameSize = 31;
```

↪ Changes the definition of a table.

FFDbiRestructureTable changes any part of the table definition. Fields can be added, changed, or removed. Indexes can be added, changed, removed, internalized, or externalized. Block sizes can be changed. Any attribute of the data dictionary can be altered while preserving the existing data.

`aDatabase` is the database component used by the table to be restructured. `aTableName` is the name of the table to restructure. `aDictionary` refers to a `TffDataDictionary` (see page 572) object that contains the structure information for the new table. `aFieldMap` is a list of strings containing field assignments of the form:

```
newfieldname = oldfieldname
```

aFieldMap is used to copy the data from the old structure to the new structure. If a field in the new table does not have an entry in the field map, it gets a null value in the new table. If a field in the old table does not appear in the field map, it is lost. If aFieldMap is nil, then all existing data is discarded and the new table is empty.

The aTaskID parameter is returned by the function, and must be used when you make a call to FFDBiGetTaskStatus.

FFDBiRestructureTable initiates the restructure and then returns immediately. If you need to determine the progress of the restructure operation, use the returned aTaskID and call FFDBiGetTaskStatus.

One of the common uses for a table restructure is to change the data type of an existing field (for example from a 16-bit integer to a 32-bit integer). However, not every data type can be converted into every other datatype. See “Converting Data Types” on page 531 for a list of the legal type conversions.

- ⚠ **Caution:** This routine will be phased out in future versions of FlashFiler. In preparation, please use the RestructureTable method of the TffTable component (see page 185) in place of this routine.

See also: FFDBiGetTaskStatus

## FFDBiSetFailSafeTransaction

function

```
function FFDBiSetFailSafeTransaction(aDatabase : TffDatabase;  
    const aFailSafe : Boolean) : TffResult;
```

- ➡ Sets the transaction mode of a database.

If fail-safe transactions are enabled, a journal file is created for each transaction. The journal file is used to recover data lost during a system crash. By default, fail-safe transactions are disabled. See “Transaction Journal Recovery” on page 45 for more information about transactions.

aDatabase is the database component being used. aFailSafe specifies whether failsafe mode should be on or off.

FFDBiSetFailSafeTransaction must be called once for every database used in the application, and will remain set until another call to FFDBiSetFailSafeTransaction turns it off.

- ⚠ **Caution:** This function will be phased out in future versions of FlashFiler. In preparation, please use the FailSafe property of the TffDatabase component (see page 168) in place of this function.

```
function FFDbiSetFilter(aTable : TffDataSet; aExprTree : pCANExpr;
  const aTimeout : TffWord32) : TffResult;

pCANExpr = ^CANExpr;

CANExpr = packed record
  iVer : Word;
  iTotalSize : Word;
  iNodes : Word;
  iNodeStart : Word;
  iLiteralStart : Word;
end;
```

✚ Assigns the server-side filter for aTable's cursor.

aTable is the table receiving the filter. aExprTree is the expression tree for the filter expression. aTimeOut is the time (in milliseconds) allowed to this operation, before it times out.

☠ **Caution:** This function will be phased out in future versions of FlashFiler. In preparation, please avoid using it.

See also: FFDbiOverrideFilter, FFDbiRestoreFilter

```
procedure FFDbiSetLoginParameters(
  const aUser : TffName; const aPass : TffName);

TffName = string[ffcl_GeneralNameSize];

ffcl_GeneralNameSize = 31;
```

✚ Sets the client username and password for future FlashFiler client sessions.

The aUser parameter specifies the name of the user as it appears in the FlashFiler Server configuration.

The aPass parameter specifies the password assigned to aUser.

If the user name and password are specified (not blank) using this routine, no login prompt is shown to the user. If the username is blank, a login prompt is displayed when an attempt is made to log onto a secure FlashFiler Server.

☠ **Caution:** This routine will be phased out in future versions of FlashFiler. In preparation, please use the SetLoginParameters method of the TffSession component (see page !!!) in place of this routine.

```
procedure FFDbiSetLoginRetries(const aRetries : Byte);
```

↪ Changes the allowable number of login retries by a client.

Default: 3

A user is allowed a certain number of attempts to log onto a secured server. FFDbiSetLoginRetries changes the number of retries. The valid range is 1 to 255.

aRetries specifies the maximum number of login attempts allowed.

⚠ **Caution:** This routine will be phased out in future versions of FlashFiler. In preparation, please use the SetLoginRetries method of the TffSession component (page 146) in place of this routine.

```
procedure FFDbiSetProtocol(aProtocol: TffCommsProtocolClass);
```

↪ Sets the client communications protocol.

The client communications protocol (now referred to as a transport) must match one of the transports enabled on the server, or no client/server communications can take place. The possible transports are single-user (for the client and server to operate as a single-user system with no physical network present), IPX/SPX, and TCP/IP. You can change the transport before taking any action that opens a session (i.e., before client communications is started with the default protocol).

The protocol classes are defined in FFLLPROT, which must be in the USES statement. The valid protocol classes are TffSingleUserProtocol, TffIPXSPXProtocol, and TffWinsockProtocol (TCP/IP).

⚠ **Caution:** This routine will be phased out in future versions of FlashFiler. In preparation, please use FFClientConfigWriteProtocolClass (see page 551) to set the default protocol in the registry.

```
function FFDbiSetTableAutoIncValue(aTable : TffDataSet;  
    const aValue : TffWord32) : TffResult;
```

🔗 Sets the auto-increment counter for a table.

aTable is the table you wish to modify. aValue parameter refers to the new value for the table's auto-increment counter.

After this method is called, the next record to be added to the table (via the Post method of the TffTable component) will be assigned an Autoinc key of aValue + 1.

**Note:** Most often this routine is called after an import process when the import manually sets the value in the autoinc field. When the autoincrement field is manually assigned, the server does not alter the auto-increment seed. In this case, you will need to call this routine to tell the table what the new autoinc value should be.

⚠ **Caution:** This routine will be phased out in future versions of FlashFiler. In preparation, please use the SetTableAutoIncValue method of the TffTable component (see page 229) in place of this function.

---

## Batched Record Routines

In situations where large numbers of records are to be processed, FlashFiler's normal way of retrieving one record at a time from the database is too slow. For example, retrieving a set of 1000 records from the server or inserting a set of 1000 new records into the server. For those situations, FlashFiler Client provides three routines for manipulating batches of records. This section describes those routines.

Note that the batched record routines provided by FlashFiler Client operate only on physical records: no data conversion to the logical types is performed. For information on logical data types see “FlashFiler Data Types” on page 524. Such data conversion is the application's responsibility. To set or get the field values from the records in a batch, use the methods of the `TffDataDictionary` (see page 572); especially `GetRecordField`, `SetRecordField`, `InitRecord`, `IsRecordFieldNull`, and `SetRecordFieldNull`.

● **Caution:** As noted in the FlashFiler-Specific Routines section, these batched routines will be phased out in future versions of FlashFiler. Please use the alternatives that are listed within the reference for each routine.

### Methods

`FFDbiGetRecordBatch`

`FFDbiGetRecordBatchEX`

`FFDbiInsertRecordBatch`

## Reference Section

### FFDbiGetRecordBatch

function

```
function FFDbiGetRecordBatch(  
    aTable : TffDataSet; const aRequestCount : Longint;  
    var aReturnCount : Longint; pRecBuff : Pointer) : TffResult;
```

🔗 Reads a batch of records from the specified table.

FFDbiGetRecordBatch allows the programmer to get a batch of records with a single message to the FlashFiler Server. Reducing message counts is a key factor in improving FlashFiler application performance over a network. The user supplies aRequestCount value and a record buffer capable of holding aRequestCount records. The server returns an array of aReturnCount records into the memory allocated by the client. If aReturnCount is less than aRequestCount, the table cursor has reached the end of table. aTable is the table from which to retrieve the records.

**Note:** FFDbiGetRecordBatch returns the physical representation of each record. You cannot use field objects to get at the data in each field; you must instead use the methods of the table's data dictionary object (the Dictionary property) to extract the data from each field. The data dictionary knows about null fields and where to find the field data in the record.

⚠ **Caution:** This function will be phased out in future versions of FlashFiler. In preparation, please use the GetRecordBatch method of the TffTable component (see page 213) in place of this function.

See also: FFDbiGetRecordBatchEx

### FFDbiGetRecordBatchEx

function

```
function FFDbiGetRecordBatchEx(aTable : TffDataSet;  
    const aRequestCount : Longint; var aReturnCount : Longint;  
    pRecBuff : Pointer; var aError : TffResult) : TffResult;
```

🔗 Reads a batch of records from the specified table.

This function is very similar to FFDbiGetRecordBatch (see previous reference), but FFDbiGetRecordBatchEx lets you know if there was a problem retrieving records from the table using the aError parameter.

⚠ **Caution:** This routine will be phased out in future versions of FlashFiler. In preparation, please use the GetRecordBatch method of the TffTable component (see page 213) in place of this routine.

See also: FFDbiGetRecordBatch

```
function FFDbiInsertRecordBatch(aTable : TffDataSet;  
    const aCount : Longint; pRecBuff : Pointer;  
    var aErrors : PffLongIntArray) : TffResult;
```

➤ Inserts a batch of records into the specified table.

FFDbiInsertRecordBatch allows the developer to insert a batch of records with a single message to the FlashFiler Server. Reducing message counts is a key factor in improving FlashFiler application performance over a network. aTable is the table into which the records are inserted. pRecBuff is a pointer to a buffer containing the records to be inserted. aCount specifies the number of records in the buffer referenced by pRecBuff. The user must supply the record count and a record buffer containing aCount records already initialized and filled using the cursor's current data dictionary. The server returns an array of error codes into memory allocated by the client. FFDbiInsertRecordBatch returns DBIERR\_NONE if all of the inserts were successful. If any of the inserts were unsuccessful, the returned value is non-zero and you must check the aErrors array to see which records were not added and why.

Note that FFDbiInsertRecordBatch requires you to use raw physical records. The VCL TField objects cannot be used to set up the field values in each record, you must instead use the table's data dictionary object (the Dictionary property). The dictionary has methods that initialize a record (i.e., set all fields to null) and that enable you to set individual fields.

⚠ **Caution:** This function will be phased out in future versions of FlashFiler. In preparation, please use the InsertRecordBatch method of the TffTable component (see page 217) in place of this function.



---

# SingleEXE Applications

SingleEXE applications are FlashFile applications that include the Client and Server code in the same application. This is different than a client/server application in which a client application uses a separate server application (usually FlashFile Server) to handle its requests. Typically, singleEXE applications are used when the application doesn't require multi-user capabilities. If you need a singleEXE application with multi-user capabilities, see "Creating your own server" on page 277. The method described below doesn't give you multi-user capabilities.

**Note:** In FlashFile 1, the only way to create a singleEXE application was to use the method described in this chapter. In FlashFile 2, faster performance may be obtained by placing a TffServerEngine component directly into the application. See "Following the Flow" on page 270 and the "ServerEngine property" on page 138 for more details. This section is provided mainly for backwards compatibility.

It is important to remember that only one instance of your singleEXE application can access the FlashFile data tables at a time. Keeping that in mind, here are the steps to create a singleEXE application that uses the single-user protocol:

1. Use FlashFile Explorer to change the client transport to Single-User.
2. Ensure singleEXE is not define in FFDEFINE.INC, i.e. it should read `{.$DEFINE singleEXE}`, not `{$DEFINE singleEXE}`.
3. Make sure you are running the FlashFile Server with the single-user transport enabled.
4. Create an alias under the Alias settings, (i.e., a name and directory associated with that name). See the next section, "Adding aliases to a singleEXE application" on page 571 for additional ways of creating your aliases.
5. Develop and test your project as a separate EXE application in Delphi or C++Builder.
6. When you're ready to compile the application into a singleEXE, open FFDEFINE.INC and change the singleEXE define from undefined to defined by removing the period in front, i.e. change `{.$DEFINE singleEXE}` to `{$DEFINE singleEXE}`
7. For those using C++Builder there is an additional step. You must include these two PAS files in your project:

FFDB.PAS

FFLLPROT.PAS

This forces C++Builder to make new images of these files and not use those in the previously generated LIB files that suppose SeparateEXE. You could go further and build a new library with the revised OBJ files. That is not absolutely necessary and only for those who know how to properly use the TLIB command line tool.

8. Recompile your application.
9. Shut down the FlashFiler Server you've been using to service the IDE.
10. Copy the FFSALIAS.FF2, FFSUSER.FF2, FFSINDEX.FF2 and FFSINFO.FF2 files from the directory where you were running the non-secure server to the same directory where your application resides.
11. Run your application.

**Note:** Some FlashFiler users have the misconception that singleEXE applications cannot create encrypted tables. This is a misunderstanding. You can use your own encryption routines to secure stand-alone, and singleEXE applications.

## Adding aliases to a singleEXE application

FlashFiler Server has a Config | Aliases menu option to enable you to add aliases, but that option is not available for singleEXE applications. Here are some of your options for setting up aliases in a singleEXE application:

1. Copy FlashFiler Server into your singleEXE application's directory. Start it up. Using the Config menu, set up the various parameters you wish (the general configuration items, the aliases, the user-defined indexes, etc.). In particular, two options must be set as follows: the "Disable all server output" option must be off and the "Disable saving configuration changes" option must be off as well. These two options ensure that FlashFiler Server will produce the files we need for our singleEXE application.

Three or four files will be created by the server: FFSINFO.FF2, FFSUSER.FF2 and FFSALIAS.FF2, with the possibility of FFSINDEX.FF2. These files must be placed in the same directory from which your singleEXE application will be executed. The linked-in server will read these files at run time and will be able to construct a list of aliases.

2. Create the aliases yourself as part of your start-up process in the singleEXE application before you open them. This is a tricky operation, best done in the OnCreate event handler for the main form in your application. Make as many calls to the AddAlias method of the TffSession session component (see page 149) as are needed to set up your alias list.
3. Link in the UFFSALAS unit into your singleEXE application to allow your users to set up the alias. Here, you need to add UFFSALAS to your project, add a menu item or button or some other visual object to call up the UFFSALAS form, and write the code to do so. The form's code takes care of reading/writing the server's alias configuration.
4. Use server configuration scripts. See "Server Scripting" on page 47 for more information.

---

## TffDataDictionary Class

In most cases, you will use FlashFiler Explorer to create and modify a FlashFiler data dictionary by creating or restructuring tables. However, you might find it necessary to work with TffDataDictionary objects to create or restructure tables programmatically. This section provides the details you need to do that.

The definition of a FlashFiler table can be described with three lists: a list of files, a list of fields, and a list of indexes. These lists are encapsulated by the TffDataDictionary class. Each FlashFiler table has a reference to a data dictionary, which is a TffDataDictionary object. FlashFiler Client routines can access a table's data dictionary directly using TffTable's Dictionary property as in the following example:

```
MyFFDataDict := MyFFTable.Dictionary;
```

The TffDataDictionary class reads and writes structures of type TffFieldDescriptor, TffIndexDescriptor, and TffFileDescriptor, which are used to describe each field, index, and file in a FlashFiler table. Methods and properties of TffDataDictionary are used to get information from these descriptors.

The possible types for the fields in a FlashFiler table are described in “FlashFiler Data Types” on page 524. TffFieldType defines those field types for use in the TffDataDictionary class.

## Hierarchy

TPersistent (VCL)

    TffDataDictionary (FFLLDict)

## Properties

BLOBFileNumber	FieldRequired	IndexDesc
BlockSize	FieldType	IndexDescriptor
BookmarkSize	FieldUnits	IndexFileNumber
DefaultFieldCount	FieldVCheck	IndexIsAscending
DiskFileName	FileBlockSize	IndexIsCaseInsensitive
FieldCount	FileCount	IndexKeyLength
FieldDecPl	FileDesc	IndexName
FieldDesc	FileDescriptor	IndexType
FieldDescriptor	FileExt	IsEncrypted
FieldLength	FileType	LogicalRecordLength
FieldName	IndexAllowDups	RecordLength
FieldOffset	IndexCount	

## Methods

AddField	ExtractKey	RemoveField
AddFile	GetFieldFromName	RemoveFile
AddIndex	GetIndexFromName	RemoveIndex
AddUserIndex	GetRecordField	SetDefaultFieldValues
Assign	HasAutoIncField	SetBaseName
BindIndexHelpers	InitRecord	SetRecordField
CheckRequiredRecordFields	InsertField	SetRecordFieldNull
CheckValid	IsIndexDescValid	SetValidityCheck
Clear	IsRecordFieldNull	WriteToStream
Create	ReadFromStream	

## Reference Section

### AddField

procedure

```
procedure AddField(const aIdent : TffDictItemName;
  const aDesc : TffDictItemDesc; aType : TffFieldType;
  aUnits : Integer; aDecPl : Integer; aReqFld : Boolean;
  const aValCheck : PffVCheckDescriptor);

TffDictItemName = string[ffcl_GeneralNameSize];

const ffcl_GeneralNameSize = 31;

TffDictItemDesc = string[ffcl_DescriptionSize];

const ffcl_DescriptionSize = 63;

PffVCheckDescriptor = ^TffVCheckDescriptor;

TffVCheckDescriptor = packed record
  vdHasMinVal : Bboolean;
  vdHasMaxVal : Boolean;
  vdHasDefVal : Boolean;
  vdFiller : Byte;
  vdMinVal : TffVCheckValue;
  vdMaxVal : TffVCheckValue;
  vdDefVal : TffVCheckValue;
  vdPicture : TffPicture;
end;
```

↪ Adds a field to the data dictionary.

The field is appended to the end of the data dictionary's field list. *aIdent* is the field identifier. *aDesc* is a text description of the field. *aType* is the field type, which must be one of the FlashFile field types (see “TffDataDictionary Class” on page 572). *aUnits* is the field size (for string fields) or the display width (for integer and floating point fields). *aDecPl* is the number of digits to the right of the decimal point (for floating-point fields). *aReqFld* is True if the field is required.

The following example creates a table called `NewTable` in the database `ServerAlias`. Three fields are included in the table.

```
var
    Dict      : TffDataDictionary;
    DB        : TffDatabase;
    FldArray  : TffFieldList;
    IdxHelpers : TffFieldIHLList;
begin
    DB := TffDatabase.Create(self);
    try
        DB.Databasename := 'LocalAlias';
        DB.AliasName := 'ServerAlias';
        DB.Connected := True;
        Dict := TffDataDictionary.Create(4096);
        try
            with Dict do begin
                AddField('EMP_NO', 'Employee number', fftInt16, 0, 0,
                    True, nil);
                AddField('NAME1', 'First name', fftShortString, 15, 0,
                    False, nil);
                AddField('NAME2', 'Last name', fftShortString, 20, 0,
                    True, nil);
                FldArray[0] := 2; {last name field}
                FldArray[1] := 1; {first name field}
                {use the default index helpers}
                IdxHelpers[0] := '';
                IdxHelpers[1] := '';
                AddIndex('Names', 'by Name', 0, 2, FldArray, IdxHelpers,
                    True, True, True);
            end;
            if DB.CreateTable(True, 'NewTable', Dict) <> 0 then
                raise Exception.Create('Error creating table');
            finally
                Dict.Free;
            end;
        finally
            DB.Free;
        end;
    end;
```

See also: `FieldCount`, `InsertField`, `RemoveField`

```
function AddFile(const aDesc : TffDictItemDesc;  
    const aExtension : TffExtension; aBlockSize : Longint;  
    aFileType : TffFileType) : Integer;  
  
TffDictItemDesc = string[ffcl_DescriptionSize];  
  
const ffcl_DescriptionSize = 63;  
  
TffExtension = string[ffcl_Extension];  
  
const ffcl_Extension = 3;  
  
TffFileType = (ftBaseFile, ftIndexFile, ftBLOBFile);
```

↳ Adds a file to the data dictionary.

aDesc is a short description of the file. The file name used by the server is the base table name plus aExtension. aFileType is the file type. There can be only one file of type ftBaseFile and one file of type ftBLOBFile in the dictionary. There can be any number of ftIndexFile types.

AddFile returns the index of the new file in the file list.

See also: AddIndex, DiskFileName, FileCount, IndexFileNumber, RemoveFile, SetBaseName

```
procedure AddIndex(const aIdent : TffDictItemName;
  const aDesc : TffDictItemDesc; aFile : Integer;
  aFldCount : Integer; const aFldList : TffFieldList;
  const aFldIHList : TffFieldIHList; aAllowDups : Boolean;
  aAscend : Boolean);

TffDictItemDesc = string[ffcl_DescriptionSize];
const ffcl_DescriptionSize = 63;

TffFieldList = array [0..pred(ffcl_MaxIndexFlds)] of Longint;
ffcl_MaxIndexFlds = 16;

TffFieldIHList = array [0..pred(ffcl_MaxIndexFlds)]
  of TffDictItemName;

TffDictItemName = string[ffcl_GeneralNameSize];
const ffcl_GeneralNameSize = 31;
```

➤ Adds a composite index to the data dictionary.

**aIdent** is the index name. **aDesc** is a description of the index. **aFile** is the file number for the index (0 is the base file). **aFldCount** is the number of fields in the composite index. **aFldList** is an array of field numbers. **aAllowDups** indicates whether keys can be duplicated. **aAscend** indicates whether the index is ascending or descending.



The following example adds an ascending index using last name and then first name (duplicate keys are allowed):

```
var
    Dict : TffDataDictionary;
    DB : TffDatabase;
    FldArray : TffFieldList;
    IdxHlprs : TffFieldIHList;
begin
    DB := TffDatabase.Create(self);
    try
        DB.Databasename := 'LocalAlias';
        DB.Connected := True;
        Dict := TffDataDictionary.Create(4096);
        try
            with Dict do begin
                AddField('EMP_NO', 'Employee number', fftInt16, 0, 0,
                    True, nil);
                AddField('NAME1', 'First name', fftShortString, 15, 0,
                    False, nil);
                AddField('NAME2', 'Last name', fftShortString, 20, 0,
                    True, nil);
                FldArray[0] := 2; {last name field}
                FldArray[1] := 1; {first name field}
                IdxHlprs[0] := '';
                IdxHlprs[1] := '';
                AddIndex('Names', 'by Name', 0, 2, FldArray, IdxHlprs,
                    True, True, True);
            end;
            if DB.CreateTable(True, 'NewTable', Dict) <> 0 then
                raise Exception.Create('Error creating table');
        finally
            Dict.Free;
        end;
    finally
        DB.Free;
    end;
end;
```

See also: AddFile, AddUserIndex, DiskFileName, FileBlockSize, FileCount, GetIndexFromName, IndexCount, IndexFileNumber, RemoveFile, RemoveIndex

```

procedure AddUserIndex(const aIdent : TffDictItemName;
  const aDesc : TffDictItemDesc; aFile : Integer;
  aKeyLength : Integer; aAllowDups : Boolean;
  aAscend : Boolean; aCaseInsens : Boolean);

TffDictItemName = string[ffcl_GeneralNameSize];
const ffcl_GeneralNameSize = 31;
TffDictItemDesc = string[ffcl_DescriptionSize];
const ffcl_DescriptionSize = 63;

```

➤ Adds a user-defined index to the data dictionary.

User-defined indexes allow you to programmatically decide how the index is computed. To implement a user-defined index, create a DLL with one function that builds a key and another function that compares keys. Then declare the DLL to the server. See “Config | User-Defined Indexes...” on page 41 and “User-Defined Indexes” on page 543 for more information.

aIdent is the index name. aDesc is a description of the index. aFile is the file number for the index (0 is the base file). aKeyLength is the length of the key you provide. aAllowDups indicates whether keys can be duplicated. aAscend indicates whether the index is ascending or descending.

See also: AddFile, AddIndex, FileCount, GetIndexFromName, IndexCount, IndexFileNumber, RemoveFile, RemoveIndex

```

procedure Assign(Source : TPersistent);

```

➤ Creates a copy of a data dictionary.

Source is an existing FlashFiler data dictionary.

The following example function gets a copy of the data dictionary from an existing FlashFiler table:

```

function CreateDictionaryFrom(
  aTable : TffTable) : TffDataDictionary;
begin
  Result := TffDataDictionary.Create(4096);
  Result.Assign(aTable.Dictionary);
end;

```

See also: Create, ReadFromStream, WriteToStream

```
procedure BindIndexHelpers;
```

- ↳ Binds each field in each index to its index helper.

Index Helpers are used to build and compare keys in an index. After the indexes are loaded, this routine associates each field in the indexes to its helper.

**BLOBFileNumber****run-time, read-only property**

```
property BLOBFileNumber : Integer
```

- ↳ The file number of the BLOB file.

If BLOBs are stored in the main data file, BLOBFileNumber is 0. If BLOBs are stored in a separate file, BLOBFileNumber is the number of that file. See “Binary Large Object (BLOB)” on page 19 for more information on BLOBs.

See also: AddFile, FileType, RemoveFile

**BlockSize****run-time property**

```
property BlockSize : Longint
```

- ↳ The block size of the base file.

BlockSize is a convenient way to access the block size of the base file. Using BlockSize is equivalent to calling FileBlockSize with an Index of 0 (the index of the base file is 0). Note that assigning a new block size doesn’t take affect until the TableRestructure method of the TffDataSet component is called.

See “Blocks” on page 20 for more information on block sizes.

See also: Create, FileBlockSize

**BookmarkSize****run-time property**

```
property BookmarkSize [aIndexID : Integer] : Integer
```

- ↳ The length of a bookmark for a given index.

The bookmark size is based on the current index.

## CheckRequiredRecordFields

function

```
function CheckRequiredRecordFields(aData : PffByteArray) : Boolean;  
PffByteArray = ^TffByteArray;  
TffByteArray = array [0..65531] of Byte;
```

↪ Determines whether all required fields in a record buffer are non-null.

FlashFile Server performs this check when a record is inserted or modified. However, a client application can minimize network traffic by calling `CheckRequiredRecordFields` to verify that all required fields are filled before the record is sent to the server.

See also: `AddField`, `FieldRequired`, `LogicalRecordLength`, `RecordLength`

## CheckValid

procedure

```
procedure CheckValid;
```

↪ Validates the data dictionary.

`CheckValid` provides a validity check for the data dictionary. It checks that at least one field is defined, that the record size will fit within a block, that all composite indexes have at least one referenced field, and that all fields referenced by a composite index exist. If the dictionary is invalid, `CheckValid` raises an exception.

The FlashFile table creation routines validate the data dictionary. However, you can use `CheckValid` for validation prior to use of the data dictionary.

```
try  
    Dictionary.CheckValid;  
except  
    on E : Exception do begin  
        {handle the error here...}  
    end;  
end;
```

## Clear

procedure

```
procedure Clear;
```

↪ Deletes all field, index, and file data from the data dictionary.

The definition for the data file (file number 0) is not deleted.

See also: `RemoveField`, `RemoveFile`, `RemoveIndex`

## Create

## constructor

```
constructor Create(aBlockSize : Longint);
```

↳ Creates a TffDataDictionary instance.

The block size of the data file component is set to aBlockSize. The block size can be changed later using the BlockSize property. The valid block sizes are 4096, 8192, 16384, 32768, and 65536.

See also: BlockSize, FileBlockSize

## DefaultFieldCount

## run-time, read-only property

```
property DefaultFieldCount : Integer
```

↳ Retrieves the number of fields in the data dictionary that have default values.

See also: SetDefaultFields

## DiskFileName

## run-time, read-only property

```
property DiskFileName [aFile : Integer] : TffFileNameExt  
ffcl_FileName = 31;  
ffcl_Extension = 3;  
TffFileNameExt = string[succ(ffcl_FileName + ffcl_Extension)];
```

↳ Returns the name of the specified file in the data dictionary.

All files in a FlashFiler table have the same name, but different extensions. The base file for a table has the extension FF2. You can choose the extension for the other files (index files and BLOB file).

aFile is the file number returned by AddFile when the file was created.

See also: AddFile, BLOBFileNumber, FileCount, FileDesc, FileDescriptor, FileExt, FileType, IndexFileNumber

## ExtractKey

## procedure

```
procedure ExtractKey(  
  aIndexID : Integer; aData : PffByteArray; aKey : PffByteArray);
```

↳ Given a record buffer and an index number, extracts the key for that index from the record.

aIndexID is an index number. aData is a record buffer. aKey is the buffer to hold the index key that is retrieved.

## FieldCount

run-time, read-only property

```
property FieldCount : Integer
```

↪ The number of fields in the data dictionary.

Many TffDataDictionary properties require the item number of a field in the field list. The field list is zero-based, so the valid range for item numbers is 0 to FieldCount-1.

See also: AddField, GetFieldFromName, InsertField

## FieldDecPl

run-time, read-only property

```
property FieldDecPl [aField : Integer] : Longint
```

↪ The number of decimal places in the specified field.

FieldDecPl is the number of digits for display of floating point numbers. aField is the field number. The field number can be obtained by calling GetFieldFromName.

⚠ **Caution:** The VCL's data-aware components ignore this information for integer and floating-point fields.

See also: FieldUnits, GetFieldFromName

## FieldDesc

run-time, read-only property

```
property FieldDesc [aField : Integer] : TffDictItemDesc  
TffDictItemDesc = string[ffcl_DescriptionSize];  
const ffcl_DescriptionSize = 63;
```

↪ The description of the specified field.

This is an optional field that describes the contents of a field. aField is the field number. The field number can be obtained by calling GetFieldFromName.

See also: FieldName, GetFieldFromName

```

property FieldDescriptor [aField : Integer] : PffFieldDescriptor
PffFieldDescriptor = ^TffFieldDescriptor;

TffFieldDescriptor = packed record
  fdNumber : Longint;
  fdName : TffDictItemName;
  fdDesc : TffDictItemDesc;
  fdUnits : Longint;
  fdDecPl : Longint;
  fdOffset : Longint;
  fdLength : Longint;
  fdVCheck : PffVCheckDescriptor;
  fdType : TffFieldType;
  fdRequired : Boolean;
  fdFiller : array [0..1] of Byte;
end;

```

↳ The descriptor of the specified field.

This property allows access to the entire field descriptor for the specified field. `aField` is the field number. The field number can be obtained by calling `GetFieldFromName`.

Although this property allows you to access the entire field descriptor, it is probably easier to access the information using the other published properties (such as `FieldName`, `FieldDesc`, etc.) of the data dictionary.

See also: `AddField`, `FieldDecPl`, `FieldDesc`, `FieldLength`, `FieldName`, `FieldOffset`, `FieldRequired`, `FieldType`, `FieldUnits`, `GetFieldFromName`, `InsertField`, `SetRecordField`

```

property FieldLength [aField : Integer] : Longint

```

↳ The length of the specified field.

`FieldLength` is the physical length of the field (or storage size) in bytes.

See also: `FieldOffset`, `InsertField`, `SetRecordField`

<b>FieldName</b>	<b>run-time, read-only property</b>
------------------	-------------------------------------

```
property FieldName [aField : Integer] : TffDictItemName
TffDictItemName = string[ffcl_GeneralNameSize];
const ffcl_GeneralNameSize = 31;
```

↪ The name of the specified field.

FieldName is used by the VCL data-aware components to provide labels for each field. aField is the field number. The field number can be obtained by calling GetFieldFromName.

See also: FieldDesc, FieldDescriptor, GetFieldFromName

<b>FieldOffset</b>	<b>run-time, read-only property</b>
--------------------	-------------------------------------

```
property FieldOffset [aField : Integer] : Longint
```

↪ The offset of the specified field in the record buffer.

aField is the field number. The field number can be obtained by calling GetFieldFromName.

See also: FieldLength, GetFieldFromName, GetRecordField, SetRecordField

<b>FieldRequired</b>	<b>run-time, read-only property</b>
----------------------	-------------------------------------

```
property FieldRequired [aField : Integer] : Boolean
```

↪ Determines whether the field is required.

FieldRequired returns True if the field must have a value, and False if the field can be blank. aField is the field number. The field number can be obtained by calling GetFieldFromName.

See also: AddField, GetFieldFromName, InsertField, SetRecordField

<b>FieldType</b>	<b>run-time, read-only property</b>
------------------	-------------------------------------

```
property FieldType [aField : Integer] : TffFieldType
```

↪ The type of the specified field.

aField is the field number. The field number can be obtained by calling GetFieldFromName. See “FlashFiler Data Types” on page 524 for a description of the valid field types.



```
property FieldUnits [aField : Integer] : Longint
```

↳ The units value for the specified field.

The units value defines either the length of a string field or the display width for a numeric field. The units value is ignored for any other data type. *aField* is the field number. The field number can be obtained by calling `GetFieldFromName`. See “Defining Fields” on page 61 for more information on the units value.

⚠ **Caution:** The VCL’s data-aware components ignore this information for integer and floating-point fields.

See also: `AddField`, `FieldType`, `GetFieldFromName`, `InsertField`

```
property FieldVCheck [aField : Integer] : PffVCheckDescriptor
```

```
PffVCheckDescriptor = ^TffVCheckDescriptor;
```

```
TffVCheckDescriptor = packed record
```

```
  vdHasMinVal : Boolean;
  vdHasMaxVal : Boolean;
  vdHasDefVal : Boolean;
  vdFiller : Byte;
  vdMinVal : TffVCheckValue;
  vdMaxVal : TffVCheckValue;
  vdDefVal : TffVCheckValue;
  vdPicture : TffPicture;
```

```
end;
```

↳ Returns the validity check for a field.

Validity checks are used to hold information required for minimum, maximum, and default field values. Minimum and maximum field values are not supported at this time.

*aField* is the field number. The field number can be obtained by calling `GetFieldFromName`. Returns `nil` if validity check not assigned.

See also: `SetFieldVCheck`

## **FileBlockSize**

**run-time, read-only property**

```
property FileBlockSize [aFile : Integer] : Longint
```

↳ The block size of the specified file.

Each file in the data dictionary can have a different block size. The valid block sizes are 4096, 8192, 16384, 32768, and 65536. aFile is the file number returned by AddFile when the file was created.

See “Blocks” on page 20 for more information on block sizes.

If you only need to access the block size of the base file, you can use BlockSize instead.

See also: AddFile, Create, BlockSize

## **FileCount**

**run-time, read-only property**

```
property FileCount : Integer
```

↳ The number of files in the data dictionary.

FlashFile tables can be stored in multiple files. There must always be one base file, which contains the data dictionary and the main data. BLOBs and indexes can optionally be stored in separate files. The file list is zero-based, with the base file as file 0.

See also: AddFile, BLOBFileNumber, IndexFileNumber, RemoveFile

## **FileDesc**

**run-time, read-only property**

```
property FileDesc [aFile : Integer] : TffDictItemDesc  
TffDictItemDesc = string[ffcl_DescriptionSize];  
const ffcl_DescriptionSize = 63;
```

↳ The description of the specified file.

FileDesc returns an optional text description for the specified file. aFile is the file number returned by AddFile when the file was created.

See also: AddFile, FileCount

## FileDescriptor

run-time, read-only property

```
property FileDescriptor [aFile : Integer] : PffFileDescriptor
PffFileDescriptor = ^TffFileDescriptor;

TffFileDescriptor = packed record
    fdNumber : Longint;
    fdDesc : TffDictItemDesc;
    fdExtension : TffExtension;
    fdBlockSize : Longint;
    fdType : TffFileType;
end;
```

↳ The descriptor of the specified file.

This property allows access to the entire file descriptor for the specified file. `aFile` is the file number returned by `AddFile` when the file was created.

Although this property allows you to access the entire file descriptor, it is probably easier to access the information using the other published properties (such as `FileDesc` and `FileExt`) of the data dictionary.

See also: `AddFile`, `BLOBFileNumber`, `FileBlockSize`, `FileCount`, `FileDesc`, `FileExtension`, `FileType`, `IndexFileNumber`, `RemoveFile`

## FileExt

run-time, read-only property

```
property FileExt [aFile : Integer] : TffExtension
TffExtension = string[ffcl_Extension];
const ffcl_Extension = 3;
```

↳ The extension of the specified file.

All files in a FlashFile table have the same name, but different extensions. The base file for a table has the extension `FF2`. You can choose the extension for the other files (`BLOB` and `index` files).

`aFile` is the file number returned by `AddFile` when the file was created.

See also: `AddFile`, `DiskFileName`, `FileCount`, `RemoveFile`

## FileType

run-time, read-only property

```
property FileType [aFile : Integer] : TffFileType  
TffFileType = (ftBaseFile, ftIndexFile, ftBLOBFile);
```

↪ The type of the specified file.

FileType indicates whether the file is a base file, index file, or BLOB file. aFile is the file number returned by AddFile when the file was created.

See also: AddFile, DiskFileName, FileCount, FileExt, RemoveFile

## GetFieldFromName

function

```
function GetFieldFromName(  
    const aFieldName : ffDictItemName) : Integer;  
TffDictItemName = string[ffcl_GeneralNameSize];  
const ffcl_GeneralNameSize = 31;
```

↪ Returns the field number for the specified field name.

GetFieldFromName gets the field number of the field specified by aFieldName. The returned field number can be used as the index for properties such as FieldDesc and FieldName. GetFieldFromName returns -1 if the field is not found.

See also: FieldCount, FieldDesc, FieldName

## GetIndexFromName

function

```
function GetIndexFromName(  
    const aIndexName : TffDictItemName) : Integer;  
TffDictItemName = string[ffcl_GeneralNameSize];  
const ffcl_GeneralNameSize = 31;
```

↪ Returns the index number for the specified index name.

GetIndexFromName gets the index number of the index specified by aIndexName. The returned index number can be used as the Index for properties such as IndexDesc and IndexName. GetIndexFromName returns -1 if the index is not found.

See also: IndexCount, IndexDesc, IndexName

```

procedure GetRecordField(aField : Integer; aData : PffByteArray;
    var aIsNull : Boolean; aValue : Pointer);

PffByteArray = ^TffByteArray;

TffByteArray = array [0..65531] of Byte;

```

↳ Reads the specified field in the record buffer.

GetRecordField returns the value of the field specified by aField from the record buffer specified by aData. The value is placed in the buffer to which aValue points. If the field is blank, aIsNull is set to True and aValue is not set.

See also: CheckRequiredRecordFields, FieldDescriptor, LogicalRecordLength, RecordLength, SetRecordField

```

function HasAutoIncField(var aField : Integer) : Boolean;

```

↳ Determines whether the record includes an auto-increment field.

HasAutoIncField returns True if the record includes an auto-increment field. The field index of the first auto-increment field in the record is returned in the variable aField. If there is no AutoInc field in the record, HasAutoIncField returns False. AutoInc fields are described in “FlashFiler Data Types” on page 524.

See also: AddField, FieldType

```

property IndexAllowDups [aIndexID : Integer] : Boolean

```

↳ Determines whether the index allows duplicate keys.

When a record is inserted or modified, a key is built for the record. If the index does not allow duplicate keys and the new key matches an existing key, the record cannot be posted.

aIndexID is the index number. The index number can be obtained by calling GetIndexFromName.

See also: AddIndex, AddUserIndex, IndexCount

## IndexCount

run-time, read-only property

```
property IndexCount : Integer
```

↪ The number of indexes in the data dictionary.

A FlashFiler table has at least one index, the Sequential Access Index. This is a pseudo-index that allows you to retrieve records in sequential insertion sequence. You can define additional indexes. The index list is a zero-based list of all the indexes for a table. The Sequential Access Index is index number 0 in the index list.

See also: AddIndex, AddUserIndex, IndexFileNumber, RemoveIndex

## IndexDesc

run-time, read-only property

```
property IndexDesc [aIndexID : Integer] : TffDictItemDesc  
TffDictItemDesc = string[ffcl_DescriptionSize];  
const ffcl_DescriptionSize = 63;
```

↪ The description of the specified index.

IndexDesc is an optional field that describes the index. aIndexID is the index number. The index number can be obtained by calling GetIndexFromName.

See also: AddIndex, AddUserIndex, IndexCount, IndexName

```

property IndexDescriptor [aIndexID : Integer] : PffIndexDescriptor
PffIndexDescriptor = ^TffIndexDescriptor;

TffIndexDescriptor = packed record {Index descriptor}
  idNumber : Longint;
  idName : TffDictItemName;
  idDesc : TffDictItemDesc;
  idFile : Longint;
  idKeyLen : Longint;
  idCount : Longint;
  idFields : TffFieldList;
  idFieldIHlprs : TffFieldIHList;
  idDups : Boolean;
  idAscend : Boolean;
  idNoCase : Boolean;
end;

```

↳ The descriptor of the specified index.

This property allows access to the entire index descriptor for the specified index. `aIndexID` is the index number. The index number can be obtained by calling `GetIndexFromName`.

Although this property allows you to access the entire index descriptor, it is probably easier to access the information using the other published properties (such as `IndexName`, `IndexDesc`) of the data dictionary.

See also: `AddIndex`, `AddUserIndex`, `IndexAllowDups`, `IndexCount`, `IndexDesc`, `IndexFileNumber`, `IndexIsAscending`, `IndexKeyLength`, `IndexName`, `IndexType`, `RemoveIndex`

```

property IndexFileNumber [aIndexID : Integer] : Longint

```

↳ The file number of the specified index.

`aIndexID` is the index number. The index number can be obtained by calling `GetIndexFromName`.

See also: `AddFile`, `AddIndex`, `AddUserIndex`, `IndexCount`, `RemoveIndex`

## **IndexIsAscending**

**run-time, read-only property**

```
property IndexIsAscending [aIndexID : Integer] : Boolean
```

↳ Determines whether the index keys are in ascending order.

aIndexID is the index number. The index number can be obtained by calling `GetIndexFromName`.

See also: `AddIndex`, `AddUserIndex`, `GetIndexFromName`, `IndexCount`

## **IndexIsCaseInsensitive**

**run-time, read-only property**

```
property IndexIsCaseInsensitive [aIndexID : Integer] : Boolean
```

↳ Determines whether the index keys are case sensitive.

aIndexID is the index number. The index number can be obtained by calling `GetIndexFromName`.

See also: `AddIndex`, `AddUserIndex`, `GetIndexFromName`, `IndexCount`

## **IndexKeyLength**

**run-time, read-only property**

```
property IndexKeyLength [aIndexID : Integer] : Longint
```

↳ The key length for the specified index.

aIndexID is the index number. The index number can be obtained by calling `GetIndexFromName`.

See also: `AddIndex`, `AddUserIndex`, `GetIndexFromName`, `IndexCount`

## **IndexName**

**run-time, read-only property**

```
property IndexName [aIndexID : Integer] : TffDictItemName
```

```
TffDictItemName = string[ffcl_GeneralNameSize];
```

```
const ffcl_GeneralNameSize = 31;
```

↳ The name of the specified index.

aIndexID is the index number. The index number can be obtained by calling `GetIndexFromName`.

See also: `AddIndex`, `AddUserIndex`, `GetIndexFromName`, `IndexCount`



## IndexType

run-time, read-only property

```
property IndexType [aIndexID : Integer] : TffIndexType
TffIndexType = (itComposite, itUserDefined);
```

↳ The type of the specified index.

FlashFiler supports both composite and user-defined indexes. See “Index” on page 24 for a description of the index types.

Index is the index number (you can get it by calling `GetIndexFromName`).

See also: `AddIndex`, `AddUserIndex`, `GetIndexFromName`, `IndexCount`

## InitRecord

procedure

```
procedure InitRecord(aData : PffByteArray);
PffByteArray = ^TffByteArray;
TffByteArray = array [0..65531] of Byte;
```

↳ Initializes a record buffer.

`InitRecord` can be used to clear all information from a record. It sets all of the fields to null.

See also: `RecordLength`, `LogicalRecordLength`, `IsRecordFieldNull`, `SetRecordField`

## InsertField

procedure

```
procedure InsertField(AtIndex : Integer;
  const aIdent : TffDictItemName; const aDesc : TffDictItemDesc;
  aType : TffFieldType; aUnits : Integer; aDecPl : Integer;
  aReqFld : Boolean; const aValCheck : PffVCheckDescriptor);
TffDictItemName = string[ffcl_GeneralNameSize];
const ffcl_GeneralNameSize = 31;
TffDictItemDesc = string[ffcl_DescriptionSize];
const ffcl_DescriptionSize = 63;
```

↳ Inserts a field into the data dictionary.

InsertField inserts a field at AtIndex and renumbers the following fields to accommodate the new field. The offsets of all subsequent fields are recalculated.

aIdent is the field identifier. aDesc is a text description of the field. aType is the field type, which must be one of the FlashFiler field types defined in “TffDataDictionary Class” on page 572. aUnits is the field size (for string fields) or the display width (for integer and floating point fields). aDecPl is the number of digits to the right of the decimal point (for floating point fields). aReqFld is True if the field is required.

See also: AddField, FieldCount, RemoveField

## IsIndexDescValid

function

```
function IsIndexDescValid(
    const aIndexDesc : TffIndexDescriptor) : Boolean;

TffIndexDescriptor = packed record
    idNumber : Longint;
    idName : TffDictItemName;
    idDesc : TffDictItemDesc;
    idFile : Longint;
    idKeyLen : Longint;
    idCount : Longint;
    idFields : TffFieldList;
    idFieldIHLprs : TffFieldIHLList;
    idDups : Boolean;
    idAscend : Boolean;
    idNoCase : Boolean;
end;
```

↪ Returns True if the given index descriptor defines a valid index.

aIndexDesc is the index descriptor to validate.

## IsEncrypted

property

```
property IsEncrypted : Boolean
```

↳ Returns True if the table's files are encrypted.

## IsRecordFieldNull

function

```
function IsRecordFieldNull(  
    aField : Integer; aData : PffByteArray) : Boolean;  
PffByteArray = ^TffByteArray;  
TffByteArray = array [0..65531] of Byte;
```

↳ Determines whether the specified field is null.

IsRecordFieldNull returns True if the field specified by aField is null. aData is the record buffer that contains the field.

See also: InitRecord, LogicalRecordLength, RecordLength, SetRecordField

## LogicalRecordLength

run-time, read-only property

```
property LogicalRecordLength : Longint
```

↳ The length of the logical record for the data dictionary.

LogicalRecordLength is the sum of the lengths of all the fields in the record. It is not the number of bytes required to physically store the record, because FlashFiler stores additional data with each record.

If you need the number of bytes required to physically store the record, use RecordLength.

See also: CheckRequiredRecordFields, GetRecordField, InitRecord, IsRecordFieldNull, RecordLength, SetRecordField

## ReadFromStream

procedure

```
procedure ReadFromStream(S : TStream);
```

↳ Restores a data dictionary from a stream.

See also: Assign, WriteToStream

## **RecordLength**

**run-time, read-only property**

property RecordLength : Longint

↳ The length of the physical record for the data dictionary.

RecordLength is the number of bytes required to physically store the record. It is not the sum of the field lengths, because FlashFiler stores additional data with each record.

If you need the sum of the field lengths, use LogicalRecordLength.

See also: CheckRequiredRecordFields, GetRecordField, InitRecord, IsRecordFieldNull, LogicalRecordLength, SetRecordField

## **RemoveField**

**procedure**

procedure RemoveField(aField : Longint);

↳ Removes a field from the data dictionary.

RemoveField removes a field and renumbers all subsequent fields to accommodate the removal. The offsets of all subsequent fields are recalculated. aField is the field number. The field number can be obtained by calling GetFieldFromName.

See also: AddField, FieldCount, GetFieldFromName, InsertField

## **RemoveFile**

**procedure**

procedure RemoveFile(aFile : Longint);

↳ Removes a file from the data dictionary.

RemoveFile removes a file and renumbers all subsequent files to accommodate the removal. All references to subsequent files are also renumbered (e.g., the file number stored with an index).

If the file contains an index, RemoveFile automatically calls RemoveIndex. aFile is the file number returned by AddFile when the file was created.

See also: AddFile, FileCount

## RemoveIndex

procedure

```
procedure RemoveIndex(aIndex : Longint);
```

↳ Removes an index from the data dictionary.

RemoveIndex removes an index and renumbers all subsequent indexes to accommodate the removal. aIndex is the index number. The index number can be obtained by calling GetIndexFromName.

See also: AddIndex, GetIndexFromName, IndexCount

## SetBaseName

procedure

```
procedure SetBaseName(const BN : TffTableName);
```

```
TffTableName = string[ffcl_TableNameSize];
```

```
const ffcl_TableNameSize = 31;
```

↳ Sets the table's base name.

All files in a FlashFiler table have the same name, but different extensions. SetBaseName sets the name for all the files. This name is used in error messages relating to the dictionary.

See also: AddFile, FileCount

## SetDefaultFieldValues

procedure

```
procedure SetDefaultFieldValues(var aData : PffByteArray);
```

```
PffByteArray = ^TffByteArray;
```

```
TffByteArray = array [0..65531] of Byte;
```

↳ Set any null fields, which have a default value, to their default value.

aData is the record buffer. See “Defining fields” on page 61 for more information about default field values.

See also: DefaultFieldCount, FieldVCheck, SetDefaultFieldValues, SetValidityCheck

```

procedure SetRecordField(
    aField : Integer; aData : PffByteArray; aValue : Pointer);
PffByteArray = ^TffByteArray;
TffByteArray = array [0..65531] of Byte;

```

↪ Sets the specified field in the record buffer.

SetRecordField sets the value of the field specified by aField in the record buffer specified by aData. aValue is the value to which the field is set. It must be a pointer to a variable of a FlashFiler data type (e.g., an fftDouble field must be set using a pointer to a variable of type Double). No type checking is performed. The valid FlashFiler data types are described in “FlashFiler Data Types” on page 524.

See also: CheckRequiredRecordFields, FieldDescriptor, LogicalRecordLength, RecordLength, SetRecordField

```

procedure SetValidityCheck(aField : Integer; var aExists : Boolean;
    const aVCheck : TffVCheckDescriptor);
PffVCheckDescriptor = ^TffVCheckDescriptor;
TffVCheckDescriptor = packed record
    vdHasMinVal : Boolean;
    vdHasMaxVal : Boolean;
    vdHasDefVal : Boolean;
    vdFiller : Byte;
    vdMinVal : TffVCheckValue;
    vdMaxVal : TffVCheckValue;
    vdDefVal : TffVCheckValue;
    vdPicture : TffPicture;
end;

```

↪ Set or clear a field's validity check information.

Use this method to assign validity check information or to remove validity check information from a field. aField is the field to which the information is assigned. To remove validity check information, set aExists to False. To add a new set of validity check information, set aExists to True and supply values for the validity check record aVCheck.

See also: FieldVCheck, SetDefaultValues

```
procedure WriteToStream(S : TStream);
```

↳ Saves a data dictionary to an existing stream.

See also: [Assign](#), [ReadFromStream](#)

---

# Crystal Reports Support

FlashFile ships with a dynamic link library (DLL) to enable Seagate Software's Crystal Reports to directly access FlashFile databases. To use these database drivers, you must already have installed the 32-bit version of Crystal Reports version 4.5 - 7. FlashFile 2 does not support 16-bit versions of Crystal Reports or version 8 and above. The driver ships as an authenticated DLL that you will install as a Crystal Reports database driver.

## Installation

The database driver is called P2BFF2xx.DLL where <xx> refers to the version of the build. It's located in FlashFile's Crystal subdirectory.

Before installing the driver, it is assumed that Crystal Reports has already been installed on the machine. Copy the P2BFF2xx.DLL file into the C:\WINDOWS\CRYSTAL directory. This is the location where Crystal Reports stores most of its database drivers. If you don't already have a C:\WINDOWS\CRYSTAL directory, and you're certain you've correctly installed Crystal Reports, search your hard disks for a native Crystal Reports database driver such as PDBPDX.DLL, PDBXBSE.DLL, or PDBBDE.DLL. Copy your FlashFile driver into that same directory. Crystal Reports also keeps drivers in the C:\WINDOWS\SYSTEM directory and the CRW application directory. You should install the FlashFile driver into C:\WINDOWS\CRYSTAL unless you have a need to move it to one of these other directories.

When Crystal Reports opens a data file, it scans the directory containing its database drivers and loads each one until it finds one that responds positively that it can recognize the data file given to it. Unfortunately, some of the native Crystal Reports drivers incorrectly respond that they can recognize FlashFile data files. When this happens, the table structure displayed by Crystal Reports usually contains only a single field called FIELD1.

You can verify whether Crystal has settled on an incorrect driver by selecting Database | Convert Database Driver. The grayed-out From line shows the name of the driver that Crystal loaded to process this data file. It should say P2BFF2xx.DLL. If it does not, then you've stumbled onto a native Crystal Reports driver that is not behaving robustly. You must remove this errant driver from the directory (or rename it so that it no longer matches the pattern PDB\*.DLL or P2B\*.DLL).

Remember, for 32-bit Crystal Reports, even if it says it's loaded the driver PDBBDE.DLL, it's really referring to P2BBDE.DLL. All the 32-bit drivers are prefixed with P2B although this display always reports PDB prefixes in both versions.



## Setting the Network Configuration

Since the Crystal Reports database driver is actually a FlashFiler client application, it needs to be aware of the network protocol to use to connect to the FlashFiler server. Use the “FlashFiler Client Communications Utility” on page 552 to set the protocol and optional fixed server name values for each client workstation.

## Using Crystal Reports

Before trying to use Crystal Reports with FlashFiler, ensure your FlashFiler Server is running.

Accessing FlashFiler data files through Crystal Reports is similar to accessing desktop files. You'll have to select the physical FF2 file from a drive and directory. For example, select File | New from the main menu. Click the “Custom>>>” button. Click “Data File”. Then select your FlashFiler data file. Paths to other machines will be converted to Universal Naming Convention format before being processed by the FlashFiler server. Paths to the local machine are only valid if the server is also running on the local machine.

You can change Crystal Report's default wildcard specifier to accommodate FlashFiler datafiles as follows. Simply change HKEY\_CURRENT\_USER\Software\Seagate Software\Crystal Reports\DatabaseOptions\DatabaseSelector to “\*.FF2.”

## Technical Support

This driver was developed by TurboPower Software Company and is not supported in any way by Seagate Software. DO NOT CONTACT SEAGATE SOFTWARE FOR TECHNICAL SUPPORT REGARDING THE FLASHFILER DATABASE DRIVER FOR CRYSTAL REPORTS. Refer all technical support questions related to the Crystal Reports driver directly to TurboPower Software. For further information, see “Technical Support” on page 16.

---

## Example Programs

FlashFiler ships with several example programs. As described below, each program is designed to orient you with a specific set of FlashFiler components and/or functions. The examples include Chat, ExBLOB, ExCust, ExFilter, ExOrders, Extend, Mythic Proportions, and Plugins.

The tables required for ExBLOB, ExCust, ExFilter, and ExOrders are located in the Examples subdirectory. Before using these examples, enter an alias named Tutorial that points to this directory in FlashFiler Server.

### Chat

Chat Server and Chat Client demonstrate the bi-directional capabilities of TffLegacyTransport. Along the way, you learn how to create a custom command handler and network messages. See “Finding new uses for transports” on page 279 for more information about this example.

The project files for this example are located in Examples\Delphi\Chat and Examples\CBuildr\Chat.

### ExBLOB

ExBLOB is a simple example that demonstrates how to open, close, filter, and iterate through a FlashFiler table. This is a good starting point for new database developers.

The project file for ExBLOB is located in Examples\Delphi and Examples\CBuildr.

### ExCust

ExCust is another simple example that demonstrates using data-aware components and navigating through a FlashFiler table.

The project file for ExCust is located in Examples\Delphi and Examples\CBuildr.

### ExFilter

ExFilter adds filtering to the ExCust example.

The project file for ExFilter is located in Examples\Delphi and Examples\CBuildr.

### ExOrders

ExOrders demonstrates how to use master/detail relationships.

The project file for ExOrders is located in Examples\Delphi and Examples\CBuildr.

## Extend

Extend shows you how to extend the functionality of the FlashFiler server engine. See “Extending the Server Engine” on page 310 for details on this example.

The Extend project files are located in Examples\Delphi\Extend and Examples\CBuildr\Extend.

## Mythic Proportions

Mythic Proportions is a large example that is broken down into three smaller projects: Order Entry, Order Processing, and Web-based inventory. Order Entry is a FlashFiler client application that is used to enter orders into the Mythic Proportions order entry system. Order Processing is the FlashFiler server application that processes the orders from Order Entry. The last part of the example is the Web-based inventory project that allows employees to take a look at Mythic Proportions inventory form anywhere on the Internet.

The FlashFiler tables used in these examples are located in Examples\MythicData. You need to establish an alias named Mythic that's directed to this directory before using any of the Mythic Proportions examples.

### Order Entry

Order Entry shows you how to setup a basic client application that uses TffRemoteServerEngine, TffLegacyTransport, TffClient, TffSession, TffDatabase, and TffTable components. This example includes data entry screens using data-aware controls. The example includes such operations as using Locate, OnNewRecord, calculated fields, master/detail relationships, filters, ranges, and BLOB streams.

The Order Entry project is discussed in Chapter 6 and the files are located in Examples\Delphi\Mythic\Client and Examples\CBuildr\Mythic\Client.

### Order Processing

The Order Processing example shows you how to embed a FlashFiler server engine within a client application and allow remote clients to connect to that same server engine. Order Processing is a FlashFiler Server that supports multiple Order Entry clients. Order Processing also moves new orders through their lifecycle. It is able to do so in a very fast manner because it talks directly to the embedded server engine instead of communicating through a transport.

On a periodic basis, Order Processing looks for a batch of new orders. It simulates credit card validation, order confirmation via email, and routes ready-to-ship orders to the appropriate warehouse.

The Order Processing project is located in Examples\Delphi\Mythic\OrdProc and Examples\CBuildr\Mythic\OrdProc.

## Web-based inventory

The Web-based inventory part of the example gives the Mythic Proportions inventory system a Web interface. Users connect to the inventory database and can view how many of each product is in stock at each warehouse. This example uses an ISAPI connection to get access to the underlying FlashFiler tables.

The Web-based inventory is located in Examples\Delphi\Mythic\ISAPI and Examples\CBuildr\Mythic\ISAPI.

## Plugins

Plugins shows you how to extend the functionality of a FlashFiler server using plugin engines and plugin command handlers. This example extends a server engine to log client errors in a central log file.

The project files are located in Examples\Delphi\Plugins and Examples\CBuildr\Plugins.

---

# Modifying the FlashFiler Source Code

Often, users modify the FlashFiler source code in order to add a customization or apply an interim bug fix. This section lists some cautions, tells you where to find a list of known bugs and interim bug fixes, and how to recompile the FlashFiler code.

## Cautions for modifying FlashFiler source code

All interim fixes to FlashFiler will appear in the next release. The interim fixes are made available to licensed FlashFiler users. If you feel you can't wait for the next release, or you want to make other changes to the source code, keep the following cautions in mind.

You're modifying the code at your own risk. TurboPower does not provide any help to apply these fixes or to recompile packages, applications or DLLs. It is assumed that if you are capable of applying these patches, you are capable of recompiling FlashFiler.

TurboPower's patch system requires an unaltered version of the product in order to patch properly. If you make any changes, you will no longer have an unaltered version and the patch to the next minor version of FlashFiler may fail. Our recommendation is to make a backup of your unadulterated FlashFiler directory (and subdirectories) before making any changes. If you don't, you will have to reinstall FlashFiler prior to installing the next version patch.

## Known bugs and interim fixes

The list of known FlashFiler bugs and interim fixes is always available from <ftp://ftp.turbopower.com/pub/flash/updates>. The bug list is named `FFBUGSxxx.TXT` and the interim fix file is named `FFFIXxxx.TXT`, where `xxx` is the FlashFiler version number. Please review the cautions listed previously before applying any of the fixes.

---

# How to Recompile FlashFiler

After making changes to the FlashFiler code (and reading the cautions listed in the previous section), you must recompile all the FlashFiler packages and applications. In case you're not familiar with this process, the steps are listed here:

1. Remove your existing FlashFiler components. Go to Component | Install Packages and remove all FlashFiler packages listed.
2. Ensure FlashFiler is in your IDE environment path. Go to Tools | Environment Options, choose Library tab, and ensure the path to your FlashFiler files (typically C:\TURBOPOWER\FLASHFILER) is listed in the library path.
3. Compile the run-time package. Open the run-time package project. The project file is named F200\_Rv\dpr where *vv* is the appropriate version number for your IDE (30 for Delphi 3, 40 for Delphi 4, 50 for Delphi 5, 35 for C++ Builder 3, 41 for C++ Builder 4, and 51 for C++ Builder 5). Change the output directory for the project (Project | Options, Directories/Conditional tab, Output Directory) to a location in your system path (we recommend Windows\System for Windows 9x and WinNT\System32 for Windows NT and Win2K). Press the Compile button on the Package editor window and save changes.
4. Compile and install the design time package. Open the design time package project. The project is named F200\_Dv\dpr where *vv* is the appropriate version for your IDE (see previous step). Change the output directory for the project (Project | Options, Directories/Conditional tab, Output Directory) to a location in your system path (see our recommendations in the previous step). Press the Install on the Package editor window and save changes.
5. Ensure you have aBin subdirectory in your FlashFiler installation.
6. Build the FlashFiler Server project. Open up the FFServer.dpr project from the FlashFiler directory. Change the project's output directory to the Bin subdirectory (described in steps 3 and 4). Build the project (Project | Build FFServer).
7. Build the FlashFiler Explorer project. Open up the FFE.dpr project from the Explorer subdirectory. Change the project's output directory to the Bin subdirectory. Build the project (Project | Build FFE).



---

# Identifier Index

## A

Aborted 360  
Active 148, 193  
ActiveBuffer 193  
ActiveCount 373  
AddAlias 149  
AddAliasEx 149  
AddField 574  
AddFile 576  
AddFileBlobEx 194  
AddIndex 234, 577  
AddIndexEx 235  
AddInterest 457  
AddToReply 360  
AddUserIndex 579  
After 443  
AfterCancel 194  
AfterClose 194  
AfterConvert 510  
AfterDelete 195  
AfterEdit 195  
AfterInsert 195  
AfterOpen 195  
AfterPost 196  
AfterRefresh 196  
AfterScroll 196  
AggFields 197  
AliasName 180  
Alloc 413, 415, 422  
AnyCellIsEmpty 97  
Append 197  
AppendRecord 197  
ApplyRange 236  
Assign 579  
AutoCalcFields 198  
AutoClientName 135  
AutoObjName 122

AutoSessionName 150  
AVG 472

## B

BchCheckInactive 388  
Before 443  
BeforeCancel 199  
BeforeClose 199  
BeforeConvert 510  
BeforeDelete 199  
BeforeEdit 200  
BeforeInsert 200  
BeforeOpen 200  
BeforePost 200  
BeforeRefresh 201  
BeforeScroll 201  
BeginUpdate 97, 323  
BindIndexHelpers 580  
BlankRow 97  
BLOBFileNumber 580  
BlockReadSize 201  
BlockSize 580  
BOF 202  
Bookmark 202  
BookmarkSize 580  
BookmarkValid 203  
BtBeginUpdatePrim 324  
BtCheckListener 324  
BtCheckSender 325  
BtCheckServerName 325  
BtEndUpdatePrim 326  
BtInitialize 326  
BtInternalReply 326  
BtLogErr 326  
BufferManager 408  
BytesToGo 361



## C

Cancel 203, 511  
 Canceled 511  
 CancelRange 237  
 CancelTransfer 264  
 CancelUpdate 326  
 CanModify 203, 257  
 CHAR\_LENGTH 472  
 CHARACTER\_LENGTH 472  
 CheckActive 132  
 CheckBrowseMode 204  
 CheckInactive 132  
 CheckRequiredRecordFields 581  
 CheckValid 581  
 ChunkSize 263  
 Clear 581  
 ClearFields 204  
 Client 150  
 ClientID 135, 361  
 ClientName 136, 151  
 Clients 122  
 Close 132, 204  
 CloseDatabase 151  
 CloseDataSets 167  
 CmdHandler 406, 429  
 CmdHandlerCount 406, 429  
 COALESCE 473  
 CommandHandler 327, 448  
 Commit 167  
 CommitFrequency 511  
 CommsEngine 151  
 CommsEngineName 136, 152  
 CompareBookmarks 205  
 ConfigDir 408  
 Configuration 408  
 Connected 180  
 ConnectionCount 327, 353  
 ControlsDisabled 205

Convert 511  
 CopyRow 98  
 COUNT 473  
 Create 181, 264, 361, 381, 512, 582  
 CreateBlobStream 206  
 CreateTable 181, 237  
 CURRENT\_DATE 474  
 CURRENT\_TIME 474  
 CURRENT\_TIMESTAMP 475  
 CurrentTransport 327  
 CurrPosition 263  
 CurrSize 263  
 CursorID 206

## D

Database 207  
 DatabaseCount 152  
 DatabaseID 183  
 DatabaseName 183, 207  
 Databases 153  
 DataSetCount 183  
 DataSets 184  
 DataSource 257  
 DefaultFieldCount 582  
 DefaultHandler 397  
 DefaultTimeout 123  
 Delete 208  
 DeleteAlias 153  
 DeleteAliasEx 154  
 DeleteIndex 237  
 DeleteTable 208  
 Destination 512  
 Destroy 184, 264  
 Dictionary 208  
 DieDieDie 379, 381  
 DiskFileName 582  
 dmXxx 319, 338, 391, 430, 436, 449  
 dsXxx 229

**E**

Edit 209  
 EditKey 238  
 EditRangeEnd 238  
 EditRangeStart 238  
 EmptyTable 209  
 Enabled 89, 328  
 EndUpdate 98, 328  
 EngineManager 389  
 ErrorCode 363  
 EstablishConnection 329, 353  
 EventLog 81, 330, 363  
 EventLogEnabled 81, 434  
 EventLogOptions 330  
 Exclusive 184, 239  
 Exec 413, 416, 422  
 ExecDirect 417, 423  
 EXTRACT 475  
 ExtractKey 582

**F**

FailSafe 168  
 Failure 443  
 FF1DirOpen 517  
 FF1FreeMem 517  
 FF1GetMem 517  
 FF1IsFileBLOB 517  
 FF1ReallocMem 518  
 FF1TableClose 518  
 FF1TableDataDictionary 518  
 FF1TableEOF 518  
 FF1TableFieldValue 518  
 FF1TableFirst 519  
 FF1TableNext 519  
 FF1TableOpen 519  
 FF1TableRecordCount 519  
 FFClientConfigGetProtocolName 548  
 FFClientConfigGetProtocolNames 548  
 FFClientConfigOverrideProtocol 549

FFClientConfigOverrideServerName 549  
 FFClientConfigReadProtocol 550  
 FFClientConfigReadProtocolClass 550  
 FFClientConfigReadServerName 550  
 FFClientConfigWriteProtocolClass 551  
 FFClientConfigWriteProtocolName 551  
 FFClientConfigWriteServerName 551  
 FFCIREng 8  
 FFDB 8  
 FFDbBase 8  
 FFDbiAddAlias 554  
 FFDbiAddFileBLOB 555  
 FFDbiAddIndex 556  
 FFDbiCreateTable 557  
 FFDbiDeleteAlias 557  
 FFDbiGetRecordBatch 568  
 FFDbiGetRecordBatchEx 568  
 FFDbiGetServerDateTime 558  
 FFDbiGetTaskStatus 559  
 FFDbiInsertRecordBatch 569  
 FFDbiOverrideFilter 560  
 FFDbiPackTable 560  
 FFDbiReindexTable 561  
 FFDbiRestoreFilter 562  
 FFDbiRestructureTable 562  
 FFDbiSetFailSafeTransaction 563  
 FFDbiSetFilter 564  
 FFDbiSetLoginParameters 564  
 FFDbiSetLoginRetries 565  
 FFDbiSetProtocol 565  
 FFDbiSetTableAutoIncValue 566  
 ffeaXxx 443, 444, 445, 446  
 ffesXxx 83, 317, 386, 451  
 FFLBase 8  
 FFLComm 8  
 FFLCore 8  
 FFLGrid 8  
 FFLLLgcy 8  
 FFLReq 8  
 FFLThrd 8  
 FFMaxBlobChunk 123  
 FFMMFreeMem 521

FFMMGetMem 521  
 FFMMReallocMem 521  
 ffnmXxx 437  
 ffrmXxx 337, 348, 362, 366  
 FFSession 125  
 FFSrCmd 8  
 FFSrEng 9  
 FFSrIntm 9  
 FFSrSec 9  
 fftpXxx 330  
 fftXxx 466, 524, 528, 531, 532, 533  
 FieldCount 583  
 FieldDecPl 583  
 FieldDesc 583  
 FieldDescriptor 584  
 FieldLength 584  
 FieldName 585  
 FieldOffset 585  
 FieldRequired 585  
 FieldType 585  
 FieldUnits 586  
 FieldVCheck 586  
 FileBlockSize 587  
 FileCount 587  
 FileDesc 587  
 FileDescriptor 588  
 FileExt 588  
 Filename 89  
 FileType 589  
 Filter 210  
 Filtered 210  
 FilterEval 211  
 FilterOptions 211  
 FilterTimeout 212  
 FindAutoFFClient 125  
 FindDatabase 154  
 FindDefaultFFClient 126  
 FindDefaultFFSession 126  
 FindFFClientName 127  
 FindFFDatabaseName 127

FindFFSessionName 128  
 FindKey 239  
 FindNearest 239  
 Flush 373  
 ForceClosed 133  
 FreeCount 373  
 FreeInstance 79, 95, 377  
 FreeStmt 413, 418, 425  
 Frequency 382  
 FROM 481

## G

GetAliasNames 155  
 GetAliasNamesEx 155  
 GetAliasPath 155  
 GetCurrentRecord 212  
 GetDatabaseNames 156  
 GetDefaultFFClient 129  
 GetDefaultFFSession 129  
 GetFFClientNames 129  
 GetFFDatabaseNames 130  
 GetFFDataDictionary 169  
 GetFFSessionNames 130  
 GetFieldData 212  
 GetFieldFromName 589  
 GetFreeDiskSpace 170  
 GetIndexFromName 589  
 GetIndexNames 240  
 GetInterestedMonitors 402  
 GetName 330, 354  
 GetRecordBatch 213  
 GetRecordField 590  
 GetServerDateTime 157  
 GetServerEngines 434  
 GetServerNames 136, 331, 355, 403, 409, 412  
 GetTableNames 157, 185  
 GetTaskStatus 158  
 GetTimeout 159, 170  
 GetTransports 435  
 GotoCurrent 215

GotoKey 240  
GotoNearest 240  
GROUP BY 481

## H

HasAutoIncField 590  
HAVING 481

## I

InactiveCount 374  
IndexAllowDups 590  
IndexCount 591  
IndexDefs 240  
IndexDesc 591  
IndexDescriptor 592  
IndexFieldCount 241  
IndexFieldNames 241  
IndexFields 241  
IndexFileNumber 592  
IndexIsAscending 593  
IndexIsCaseInsensitive 593  
IndexKeyLength 593  
IndexName 242, 593  
IndexType 594  
InitialCount 374  
InitRecord 594  
Insert 216  
InsertField 594  
InsertRecordBatch 217  
Interested 458, 462  
InterestedActions 454  
InTransaction 170  
IsAlias 159  
IsConnected 137, 331, 356  
IsDefault 137, 159  
IsEncrypted 596  
IsIndexDescValid 595  
IsOwned 133  
IsReadOnly 403

IsRecordFieldNull 596  
IsSequenced 219  
IsSQLBased 171  
ixXxx 234

## K

KeyExclusive 242  
KeyFieldCount 242  
KeySize 242

## L

LastRowIsEmpty 98  
LcLog 81  
LoadFF1DLL 521  
Locate 219  
Lock 364  
Lookup 220  
LOWER 475

## M

MasterFields 250  
MasterSource 250  
MAX 476  
MaxCount 374  
MIN 476  
Mode 332  
ModifyAlias 160  
MsgCount 332  
MsgID 364

## N

NewInstance 79, 95, 377  
NoAutoSaveCfg 404  
Notify 455  
NULLIF 477

## O

ObjectView 220  
 omReadOnly 416  
 omReadWrite 416  
 OnAddClient 333  
 OnCalcFields 221  
 OnCancel 512  
 OnChooseServer 160  
 OnComplete 512  
 OnConnectionLost 334, 356  
 OnDeleteError 221  
 OnEditError 222  
 OnEnterCell 99  
 OnExitCell 99  
 OnFilterRecord 222  
 OnFindServers 161  
 OnLogin 162  
 OnNetBios 513  
 OnNewRecord 223  
 OnPostError 223  
 OnProgress 513  
 OnRemoveClient 335  
 OnServerFilterTimeout 223  
 OnSortColumn 99  
 OnStartUp 162  
 OnStateChange 85  
 Open 133  
 OpenDatabase 162  
 ORDER BY 481

## P

PackTable 171, 224  
 ParamByName 258  
 ParamCheck 258  
 ParamCount 259  
 Params 259  
 piXxx 419, 426

## POSITION 477

Position 265  
 Post 225, 336, 347  
 Prepare 259, 413, 418, 425  
 Prepared 260  
 Process 338, 379, 390, 397, 430, 436, 448  
 ProcessThreaded 375  
 ProgressFrequency 514  
 Protocol 143, 357  
 ProtocolClass 144  
 ptXxx 143, 357

## R

Read 265  
 ReadFromStream 596  
 ReadOnly 185, 243  
 RecNo 225  
 RecordLength 597  
 RecordsProcessed 514  
 Refresh 225  
 ReindexTable 172, 243  
 RemoveAllInterest 459  
 RemoveField 597  
 RemoveFile 597  
 RemoveIndex 598  
 RemoveInterest 459  
 RenameTable 225  
 Reply 339  
 ReplyData 365  
 ReplyDataLen 365  
 ReplyMode 366  
 ReplyMsgID 366  
 Request 340, 349  
 RequestData 367  
 RequestDataLen 367  
 RequestLive 260  
 ResetMsgCount 341  
 RespondToBroadcasts 342  
 Restart 431, 437  
 RestoreToRow 100

RestructureTable 185, 226  
 Rollback 173  
 RowIsEmpty 100  
 RowIsFilled 100

## S

SaveRow 101  
 ScInitialize 85  
 ScPrepareForShutdown 85  
 ScShutdown 85  
 ScStartup 86  
 Seek 266  
 SELECT 478, 481  
 ServerEngine 138, 163, 188, 228, 394, 459, 515  
 ServerName 145, 342, 409  
 Session 130, 189, 228  
 SessionCount 138  
 SessionID 163  
 SessionName 163, 189, 228  
 Sessions 139  
 SetBaseName 598  
 SetDefaultFieldValues 598  
 SetKey 244  
 SetLoginParameters 164  
 SetLoginRetries 164  
 SetParams 413, 419, 426  
 SetRange 244  
 SetRangeEnd 245  
 SetRangeStart 245  
 SetRecordField 599  
 SetReply 368  
 SetTableAutoIncValue 229  
 SetValidityCheck 599  
 Shutdown 86, 431, 438  
 Size 266  
 Sleep 343  
 Source 515  
 SparseArrays 229  
 SQL 261

SQLEngine 409  
 StartOffset 369  
 StartTransaction 174  
 Startup 86, 432, 438  
 State 229, 343  
 Stop 86, 432, 438  
 SUBSTRING 482  
 SUM 483  
 Supported 343

## T

TableExists 175  
 TableName 246  
 TDataSet.RecNo 492  
 Temporary 190  
 TerminateConnection 343, 358  
 TffBaseClient 8, 134  
 TffBaseCommandHandler 8, 387  
 TffBaseDatabase 166  
 TffBaseDatabase, 8  
 TffBaseEngineExtender 8, 453  
 TffBaseEngineManager 8, 428  
 TffBaseEngineMonitor 8, 456  
 TffBaseLog 88  
 TffBasePluginCommandHandler 8, 447  
 TffBasePluginEngine 8, 450  
 TffBaseServerEngine 8, 401  
 TffBaseTable 8  
 TffBaseTransport 8, 321  
 TffBlobStream 262  
 TffClient 8, 141  
 TffCommsEngine 8, 142  
 TffComponent 8, 78  
 TffDatabase 8, 177  
 TffDataConverter 508  
 TffDataDictionary 572  
 TffDataset 8, 191  
 TffDBListItem 8, 131  
 TffEngineManager 9, 433  
 TffEventLog 91

TffIntermediateCommandHandler 9, 393  
 TffIntermediateServerEngine 9, 405  
 TffLegacyTransport 8, 351, 488  
 TffLoggableComponent 80  
 TffObject 8, 94  
 TffPooledThread 8, 378  
 TffQuery 251  
 TffRemoteServerEngine 8, 410  
 TffRemoveClientEvent 335  
 TffRequest 8, 359  
 TffSecurityExtender 9  
 TffSecurityMonitor 9, 460  
 TffServerCommandHandler 8, 395  
 TffServerEngine 9, 407  
 TffSession 8  
 TffSQLEngine 421  
 TffSrBaseCursor 442  
 TffSrClient 442, 443  
 TffSrCursor 445  
 TffSrDatabase 441, 444  
 TffSrSession 441  
 TffStateComponent 82  
 TffStringGrid 8, 96  
 TffTable 8, 247  
 TffTableProxy 8  
 TffThread 376  
 TffThreadedTransport 8, 345  
 TffThreadPool 8, 372  
 TffTimerThread 8, 380  
 ThreadPool 350  
 Timeout 139, 165, 190, 231, 369  
 TotalRecords 515  
 TransactionCorrupted 175  
 Translate 231

Transport 412  
 TransportCount 392  
 Transports 392  
 TRIM 484  
 Truncate 266  
 TryStartTransaction 175  
 TwwSearchDlg 487

## U

uFFEgMgr 9  
 Unlock 370  
 UpdateCount 344  
 UPPER 484  
 UserName 140

## V

Version 79, 231

## W

WaitForReply 372  
 WakeUpThread 370, 371  
 WHERE 481  
 Work 344, 358  
 Write 267  
 WriteBlock 89, 92  
 WriteString 89, 92  
 WriteStringFmt 90, 93  
 WriteStrings 90, 93  
 WriteToStream 600

---

# Subject Index

## A

- adding
  - aliases to a single EXE application 571
  - field 574
  - file 576
  - FileBLOB 194
  - index 234, 235, 577
  - user-defined index 579
- adding alias 149
- admin user, default password 39
- administration rights 39
- after image 45
- alias
  - adding 149
  - autoinc value, setting 229
  - creating permanent 554
  - definition 19, 32
  - deleting 40, 154, 557
  - deleting from server 153
  - exists 149
  - modifying 160
  - retrieving list of names 129, 155, 156
  - retrieving server-relative path 155
  - returning list of names 130
  - validating 159
- applying range 236
- architecture
  - advanced 269
  - client 270
- ASCII files, importing 65, 534
- attaching servers 57
- autoinc value, setting 566
- automatic components 118

## B

- base classes
  - changing 85
  - identifying 87
  - making inactive 132
  - plugin engine 450
  - shutting down 85
  - starting 86
  - startup 85
  - testing 132
- batched record routines
  - getting 213, 568
  - inserting 213, 217, 568, 569
- batched records
  - getting 568
  - inserting 568
- BDE data types 524
- BDE export to ASCII utility 8
- BDE2FF 8
- before image 45
- BETA 8
- BIN 7
- binary files, importing 65, 528
- Binary Large Object (BLOB) 19
- BLOB
  - creating 264
  - creating stream 206
  - determining size 266
  - file number 580
  - freeing an instance of 264
  - specifying size 263
  - truncating data 266
  - writing data 267
- BLOB field, inserting a reference 555
- BLOB stream 115, 264



- block
  - BLOB 19, 21
  - data 20
  - definition 20
  - file, size of 587
  - index 20
  - size 580
- bookmark
  - comparing 205
  - size 580
  - validating 202, 203
- Borland Database Engine (BDE) 1
- broadcasts 38, 209
- B-Tree Filer, importing files 65, 534
- buffer manager, returning 408
- buffer variable
  - initializing 324
  - transferring 326, 328
- buffer, determining size 510

## C

- calculated fields 107
- cancelling
  - operation 203
  - range 237
  - table edit 203
  - table insert 203
- cell
  - copying contents 98
  - deleting contents 97
  - gaining focus 99
  - indicating empty 98
  - losing focus 99
- changing table name 69, 175
- Chat Client 293
  - defining command handler 293
  - implementing command handler
    - methods 294
  - integrating with user interface 297
  - TffChatClientHandler methods 293

- Chat Server
  - adding command handler 283
  - adding the transport 280
  - defining data structures 281
  - implementing message handlers 289
  - implementing user interface methods
    - 291
  - methods 284
- checking required fields 581
- class
  - hierarchy 10
  - methods 14
  - overview 13
  - properties 14
  - reference section 14
- client
  - adding new event 333
  - architecture 103
  - indicating connection to server 137
  - indicating default 137
  - processing request 430, 436
  - retrieving timeout 139, 159, 231
  - returning 125
  - returning default 126
  - returning list of accessible servers 136
  - returning list of names 129
  - returning name 127
  - returning object 129
  - setting communications protocol 565
  - specifying 361
  - specifying name 136, 151, 152
  - specifying ownership of session 150, 151
  - specifying timeout 139, 215
  - specifying unique identifier 135
  - specifying user name 140
  - termination request 335
- client application
  - always use logical data types 529
  - sending a reply 326
- client classes hierarchy 120
- client communications utility 552

- client component
  - connecting to server engine 138
  - generating unique name 135
  - indicating default client 137
  - indicating ownership 133
  - retrieving number of sessions attached 138
  - specifying client name 136
- client configuration 547
  - overriding stored server name 549
  - overriding the stored protocol 549
  - returning default server name 550
  - returning list of protocol names 548
  - returning protocol class 550
  - returning protocol name 548, 550
  - storing protocol 551
  - storing server name 551
- client source files 2
- client/server architecture 2
- columns, sorting 99
- command handler
  - defining 327
  - freeing message 388
  - processing request from transport 390
  - providing access to 406, 429
  - providing access to transports 392
  - returning number of 406, 429
  - routing messages 397
  - routing requests to message handler 397
  - specifying engine manager 389
  - specifying server engine 394
  - states 385
  - verifying number of transports 392
- command line conversion utility 504
- comms engine
  - defined 22
  - explained 142
  - specifying name and address 145
- compatibility 486
  - components not compatible 487
  - data-aware grid 486
- compatibility (continued)
  - Express Quantum Grid 486
  - InfoPower 486
- completing insert operation 225
- component
  - making state inactive 132
  - specifying client name 136
- component helper
  - automatic client 125
  - client name 127
  - client names 129
  - client object 129
  - database names 130
  - database object 127
  - default client 126
  - default session 126, 129, 130
  - default session object 125
  - session name 128
  - session names 130
- composite index 2, 24, 64
- config
  - alias 40
  - aliases 40
  - general configuration 35
  - network configuration 38
  - user permissions 39
  - user-defined indexes 41
- configuring development environment 12
- connection
  - client/server 37
  - establishing 329, 353
  - losing 334, 356
  - number of 327, 353
  - terminating 343, 358
- connectivity
  - cannot connect to network server 487
  - cannot locate network server 487
  - client loses connection 488
  - client/server applications 487
  - default transport 488
  - table cannot open its database 488

- conversion
  - BLOBs 508
  - cancelling 511
  - cancelling table 512
  - creating an instance of
    - TffDataConverter 512
  - destination directory 500
  - determining progress frequency 514
  - encrypted tables 500
  - exit codes 506
  - FlashFiler 1 table to FlashFiler 2 table 511
  - identifying cancellation status 511
  - number of records converted 514
  - progress point 513
  - returning FlashFiler 2 server 515
  - returning location of converted table 512
  - returning source 515
  - returning total number of records 515
  - source directory 500
  - status 506
  - table conversion completed 512
  - using NetBIOS protocol 513
- Conversion Memory Manager 520
- conversion status 506
- conversion utilities
  - command line 500
  - GUI 500
- CONVERT 8
- converter
  - after conversion 510
  - before conversion 510
  - determining buffer size 510
- converting
  - data types 531
  - server tables 505
- creating
  - BLOB stream 206, 262
  - data dictionary 579, 582
  - database 181
  - index 234, 235, 577

- creating (continued)
  - plugins 300
  - table 181, 237
  - transaction journal 45
  - your own server 277
- Crystal Reports 489
  - installing 601
  - support 601
  - using 602
- cursor
  - definition 22
  - positioning 240
- cursor, positioning 240

## D

- data
  - importing into tables 65, 534
  - refreshing 225
- data access component
  - opening 133
- data block 20
- data block, logging 92
- data conversion 8, 66, 499, 531
- data conversion utilities 3, 8
- data dictionary
  - adding field 574
  - adding file 576
  - adding index 577
  - adding user-defined index 579
  - autoinc field 590
  - binding index helpers 580
  - block size 580
  - bookmark size 580
  - case sensitive index keys 593
  - clearing 581
  - copying 579
  - creating 582
  - definition 23
  - deleting field 597
  - deleting files 597

- data dictionary (continued)
  - deleting index 598
  - description of field 583
  - determining field autoinc 590
  - encrypted file 596
  - extracting key from record 582
  - field descriptor 584
  - field length 584
  - field name 585
  - field offset 585
  - field type 585
  - field units 586
  - file block size 587
  - file description 587
  - file extension 588
  - file name 582
  - file number of BLOBS 580
  - file number of index 592
  - file type 589
  - getting field with name 589
  - getting index with name 589
  - index description 591, 592
  - index descriptor 595
  - index key length 593
  - index keys ascending 593
  - index name 593
  - index type 594
  - initializing record 594
  - inserting a field 594
  - key duplicates 590
  - logical record length 596
  - null field 596
  - number of decimal places in field 583
  - number of files 587
  - number of indexes 591
  - physical record length 597
  - providing access to 208
  - reading field 590
  - removing field from 597
  - removing file 597
  - removing index 598
  - required field 585
- data dictionary (continued)
  - required fields 581
  - restoring from stream 596
  - saving 600
  - setting field default value 598
  - setting field in record buffer 599
  - validating 581
  - write to stream 600
- data entry forms, setting up 111
- data file repair facilities 3
- data integrity 31
- data module
  - data entry forms 111
  - Engine Manager 275
  - setting up 104
- data stream
  - aborting read/write operation 264
  - indicating current position 265
  - positioning 266
  - specifying current offset 263
- data types
  - ASCII import files 539
  - BDE 524
  - binary/B-Tree Filer import files 540
  - comparison 528
  - converting 531
  - importing 539
  - logical 524
  - physical 524
  - SQL 466
  - translating 528
- data, importing 65, 534
- data-aware controls, disabling 205
- data-aware grid 486
- database
  - accessing 153
  - active 130
  - adding alias 149
  - alias 127
  - applying updates 167
  - changing table format 185
  - class hierarchy 166

database (continued)

- closing 151
- closing datasets 167
- committing transaction 167
- context menu 59
- corrupt transaction 175
- creating 181
- creating table 181, 557
- defining exclusive access 184
- definition 22
- destroying 184
- determining SQL 171
- explicit 177
- finding 154
- implicit 177
- integrity 31
- list of names 129
- list of tables 157
- managing component 207
- name 207
- new transaction 174
- number managed by session 152
- opening 162
- pack operation for table 171
- providing access to tables 184
- read-only access 185
- reindexing table 172
- renaming 59
- retrieving data dictionary 169
- retrieving names of all tables 157
- retrieving timeout 170
- returning amount of available disk space 170
- returning identifier 183
- returning list of names 130, 156
- returning name 127, 183
- rolling back transaction 173
- server connection 180
- server engine 188
- session name 189
- setting fail-safe mode 168
- specifying number of active tables 183

database (continued)

- specifying opened alias 180
- specifying session component 189
- specifying timeout 165, 190
- SQL based 171
- starting transaction 175
- table names 185
- temporary 190
- terminating current transaction 173
- timeout value 170
- transactions started 170
- verifying existence of table 175

database ancestor classes 119

database context menu

- creating a new table 61
- refreshing list 60

dataset

- adding new empty record 197
- after cancel operation event 194
- after closed event 194
- after edit event 195
- after open event 195
- after record inserted event 195
- after record posted event 196
- after refreshed event 196
- after scroll event 196
- applying filter 210
- automatically recording changes 204
- before cancel event 199
- before closed event 199
- before edit event 200
- before record inserted event 200
- before record posted event 200
- before record removed event 199
- before refreshed event 201
- before scroll event 201
- before table opened event 200
- bookmark valid 203
- changing structure of table 226
- closing 204
- comparing bookmarks 205
- completing edit operation 225

## dataset (continued)

- contain aggregate fields 197
- copying strings 231
- creating BLOB stream 206
- creating object for array field 229
- current record 225
- database 207
- database name 207
- defining event after record deleted 195
- defining fields 61
- defining filter 210
- defining filter behavior 211
- defining open or closed status 193
- deleting 59
- deleting current record 208
- deleting table 208
- disabling data-aware controls 205
- discarding modifications 203
- editing 209
- emptying of all records 209
- enabling editing of current record 209
- error during delete 221
- error during editing 222
- evaluating filter 211
- filter options 211
- filter string 210
- filtering records 222
- freeing unused space 224
- inserting reference into BLOB field 194
- locating record 219
- looking up records 220
- modifying 203
- new record added 223
- number of records read 201
- OnCalcFields 198
- packing 224
- placing into key edit mode 238
- pointing to active buffer 193
- positioning cursor at beginning 202
- positioning cursor to current record 215
- post error 223
- posting record 197

## dataset (continued)

- preparing for new record insertion 216
- providing access to data dictionary 208
- reading batch of records 215
- recalculating fields 221
- redefining 69
- refreshing data 225
- renaming table 225
- restructuring 226
- retrieving timeout value 212, 215
- returning bookmark 202
- returning current record 212
- returning unique identifier 206
- returning value of field object 212
- returning version number 231
- reverting record fields to initialized state 204
- sequence numbers 219
- session name 228
- setting auto-increment value 229
- specifying timeout 231
- storing fields 220
- supporting sequence numbers 219
- timeout of server filter 223
- validating bookmarks 202

## date and time values 536

### date fields 524

### date mask characters 537

### date value, formatting 536

### debug log 496

### debugging log 43

### defining

- engine extender actions 454

- fields in master table 250

- master table 250

### definition of

- alias 19

- Binary Large Object (BLOB) 19

- blocks 20

- client 21

- comms engine 22

- cursor 22

- definition of (continued)
  - data dictionary 23
  - database 22
  - deleted record chain 23
  - embedded server 23
  - index 24
  - multi-threaded client 24
  - multi-threaded server 25
  - query 25
  - remote server 25
  - security 25
  - session 26
  - Single User Protocol 26
  - table 26
  - TDataSet Descendant Model 26
  - thread pool 27
  - transaction 27
  - transaction manager 28
  - transport 28
- deleted record chain 23
- deleting
  - alias 59, 153, 557
  - fields 581
  - files 581
  - index 581
  - table 208
- deleting alias 40
- deploying FlashFiler Server 54
- description of
  - file 587
  - index 591
- descriptor
  - field 584
  - index 592
- design-time errors
  - cannot locate or connect to network server 489
  - index name with spaces 489
- determining field autoinc 590
- disk activity, reducing 496
- duplicate keys 590

## E

- embedded server 23
- engine extender
  - defining actions 454
  - notifying of actions 455
- engine manager
  - data module 275
  - disabling logging of errors 434
  - enabling logging of errors 434
  - processing client request 430, 436
  - providing access to command handler 429
  - restarting server engine 431, 437
  - retrieving list of server engines 434
  - retrieving transports 435
  - returning number of command handlers 429
  - shutting down server engine 431, 438
  - specifying 389
  - starting 432, 438
  - stopping 432, 438
- engine monitor 442
  - adding interest 457
  - creating 310, 442
  - identifying server engine 459
  - indictating interest 458
  - monitoring server objects 442
  - removing all interest 459
  - removing interest 459
- engine notifications 443
- error messages 489
- event monitoring 441
- EventLog
  - specifying type of messages written 330
  - writing messages to 81, 89, 90, 92, 93, 330
- examples
  - ExBLOB 603
  - ExCust 603
  - ExFilter 603
  - ExOrders 603

- examples (continued)
  - Extend 604
  - Mythic Proportions 604
  - Order Entry 604
  - Order Processing 604
  - summary 603
  - Web-based inventory 605
- Express Quantum Grid 486
- extending the server engine 310

## F

- fail-safe transaction
  - describing setting 563
- field
  - adding 574
  - decimal places 583
  - defining 250
  - description of 583
  - descriptor 584
  - detecting autoinc 590
  - getting number of 574, 589
  - inserting 62, 594
  - length 62, 584
  - mapping 69
  - name 61, 585
  - null 596
  - obtaining validity check 586
  - reading 590
  - recalculating 221
  - required 585
  - schema 534
  - setting 599
  - setting default value 598
  - setting validity check 599
  - transaction journal 45
  - type 62, 585

- units 62, 586
- fields
  - adding 61
  - defining 61
  - description of 62
- file
  - adding 63, 64, 576
  - BLOB 194, 555
  - block size 20, 61, 587
  - description 587
  - encrypted 25, 596
  - extension 588
  - logging 89
  - name 582
  - number of 587
  - removing from data dictionary 597
  - type 589
- file number of index 592
- files supplied 7
- filter
  - applying 210
  - client-side 210
  - defining 210
  - options 211
  - overriding existing 560
  - restoring server-side 562
  - server-side 211
  - setting 564
  - timeout 211, 223
  - using 112
- finding
  - new uses for transports 279
  - records 114
- FlashFiler 1 DLL
  - allocating memory 517, 521
  - calling DLL routines 516
  - closing table 518
  - detecting file BLOB 517



FlashFiler 1 DLL (continued)  
     establishing source directory 517  
     freeing memory 517  
     loading 521  
     moving to first record 519  
     moving to next record 519  
     opening table 519  
     reallocating memory 518  
     retrieving data dictionary 518  
     retrieving field value 518  
     returning number of records 519

FlashFiler Explorer  
     creating a new database 58, 59  
     database context menu 59  
     deleting a database 59  
     detaching from FlashFiler Server 57  
     importing files 65  
     operating 56  
     refreshing list of active servers 57  
     registering server name 57  
     renaming a database 59  
     server context menu 57  
     SQL statements 59  
     table context menu 67  
     unregistering server 57  
     viewing indexes 68

FlashFiler Server  
     adding alias 40  
     attaching 57  
     caching 35  
     configuring 32  
     configuring user permissions 39  
     customizing limits 276  
     detaching 57  
     exiting 35  
     finding 57  
     help 43  
     Keep-Alive Options 37

FlashFiler Server (continued)  
     logging 43  
     menu 34  
     modifying 275  
     naming 35  
     operating 33  
     registering 57  
     reset counters 43  
     security 25  
     setting priority 36  
     starting automatically 37  
     starting minimized 37  
     tables 32  
     unregistering 57

FlashFiler Service 44

FlashFiler-specific routines 553

freeing unused space 171, 224, 560

FROM 481

## G

getting  
     batched records 213, 568  
     field number 589  
     index number 589  
     task status 158, 559

global constants 121

global variables 121

GUI Conversion Utility  
     destination table, specifying 501  
     encryption 500  
     watching conversion status 502

GUI conversion utility  
     Directory List 501  
     Drive List 501  
     Progress Bar 502  
     source table, specifying 501  
     Status Bar 502  
     Status View 502  
     Table List 501

## H

### help

- C++Builder 7
  - FlashFile Explorer 76
  - FlashFile Server 43
  - general 15
  - suggested reading 17
  - technical support 16
- HELPCBUILDER\FBCB.HLP 7

## I

- IDE integration 12
- importing data into FlashFile tables 534
- importing files
  - ASCII 539
  - binary 540
  - B-Tree Filer 540
  - data types supported 539
  - example 542
  - from FlashFile Explorer 65
  - schema file 534
- index
  - adding to table 234, 235, 556
  - adding user-defined 64, 543, 579
  - binding field to index helper 580
  - cancel
    - edit 203
    - insert 203
  - deleting 237
  - duplicate keys 64, 590
  - extracting key 582
  - file number of 592
  - finding by name 589
  - key length 64, 593
  - keys ascending 64, 593
  - name 593
  - number of 591
  - reconstructing key values 243, 561
  - referencing definitions 240
  - removing from data dictionary 598

### index (continued)

- retrieving description 591
- returning key size 242
- returning name 242
- specifying columns to use 241
- storing in external file 64
- type 594
- user-defined 543
- validating 595
- viewing 68

### InfoPower 486

- initializing record buffer 594
- inserting batched records 217, 568, 569
- installing
  - encryption routines 52
  - FlashFile 7, 12, 490
  - singleEXE 490
- interest
  - adding 457
  - querying 458
  - removing 459
  - removing all 459
- Internet 1, 3
- IPX/SPX 38, 351, 357

## K

### Keep-Alive Options 37

### key

- extracting 579
- field count 241
- length 593
- key field, defining count 242
- key size, returning 242
- key words, SQL 471

## L

- large table support 2
- length of field 62, 584
- Listen for Broadcasts 38

- live result sets 255
- locating record 219
- log
  - See also* logging
  - debugging 43
  - identifying 363
  - naming 81
  - writing events 363
  - writing to 81, 89, 91–93, 403
- logging
  - See also* log
  - array of strings 90, 93
  - centralized 80, 88, 91
  - data block 89, 92
  - disabling 434
  - enabling 81, 89, 434
  - filename 89
  - formatted messages 90, 93
  - string 81, 89, 92
- LogicalRecordLength 596
- login
  - parameters, setting 162, 564
  - retries, setting 565
- looking up records 220
- lookup fields 109, 498

## M

- master/detail relationship 110, 250
- Maximum RAM
  - defining 35
  - setting 496
- memory
  - allocating 79, 95, 377
  - deallocating 79, 95, 377
  - management 78, 79
- memory manager
  - allocating memory 78, 94, 521
  - freeing memory 78, 94, 521
  - resizing memory 521

- message
  - freeing 388
  - logging 89
  - resetting count 341
  - routing 383, 395, 397, 448
  - specifying type 364
  - statistics 34, 495
- message handler, routing requests 397
- modifying
  - FlashFiler Server 275
  - path 160
  - result set 257
- monitors, returning interested 402
- multithread applications 118
- multi-threaded client 24
- multi-threaded server 25

## N

- name of
  - field 64, 585
  - file 582
  - index 593
- name, generating 135
- naming conventions 15, 465
  - columns 465
  - tables 465
- NetBIOS 504
- network configuration, setting 38, 602
- number of
  - files 587
  - indexes 591

## O

- object
  - automatic 118, 122
  - forcing close 133
  - storing reference 122
  - timeout 123, 139, 165, 190
- Orpheus 4

## P

packing table 171, 560

parameter

- by DataSource 254

- by name 253

- by position 254

- providing access to 259

- returning name 258

- returning number 259

- setting 419, 426

- updating list 258

parameterized queries 252

performance

- batch methods 496

- custom encryption 497

- data-aware controls and 494

- disabling debug log 491

- disk activity 496

- embed the server engine 493

- explicit transactions 496

- extenders 497

- fail-safe transaction 497

- lookup fields 498

- manually disabling DBGrids 494

- monitoring message traffic 495

- monitors 497

- ranges 498

- RecordCount method and 497

- server-side filters 498

- single-user transport 498

- TffTable.Close and 497

- TffTable.Open and 497

- user-indexes 497

physical data types 524

plugin

- creating 300

- creating abstract plugin engine 301

- creating real plugin engine 302

- creating remote plugin engine 302

- integrating 307

- states 450

plugin command handler

- creating 305

- processing message 448

- routing message 448

plugin engine 437

- overview 450

- shutting down 431, 438

- starting 432, 438

- stopping 432, 438

preparing SQL statement 417, 423

property modifications

- completed 328

- discarding 326

protocol

- comms engine, setting 142

- identifying 357

- overriding 549

- returning class 144

- returning name 548

- returning name and class 550

- setting 565

- Single User 26

- specifying 143

- storing in registry 551

- value 357

- writing to registry 551

## Q

## query 25

- executing SQL statement 261
- indicating whether a live result set is expected 260
- parameter name 258
- prepared for execution 260
- preparing 251
- preparing SQL statement for execution 259
- providing access to parameters 259
- returning number of parameters 259
- specifying data source component 257
- updating parameter list 258

## R

## range

- applying 236, 244
- cancelling 237
- interpreting boundaries 242
- setting end 245
- setting start 245

## ranges

- defined 113
- setting end 244
- setting start 244

## reading batch 568

## reading field 590

## README.HLP 7

## read-only access, defining 243

## recalculating fields 221

## record

- adding new 197
- current 225
- deleting current 208
- emptying dataset 209
- enabling editing 209
- filtering 222
- finding nearest 239
- finding specific 239

## record (continued)

- initializing 594
- initializing buffer 594
- inserting batch 217, 569
- inserting new 216
- length 596
- locating 219
- looking up 220
- physical length 597
- posting 197
- reading batch 213
- reading batch of 215
- removing deleted 224
- returning current 212
- reverting to initialized state 204
- setting range 113

## records 568

## redefining table 185

## reducing disk activity

- setting maximum RAM appropriately 496
- using explicit transactions 496

## reducing messages between client and server 493

## referencing index definitions 240

## reindexing table 561

## remote server 25

*See also* server

## definition of 104

## returning list 403

## returning name 409, 412

## specifying name 342

## temporarily suspending connection 343

## transport connection 104

## removing deleted records 224

## reply

## appending data 360

## expecting 366

## sending from a Listening transport 339

## sending to client application 326

## setting 368

## specifying message identifier 366

- request
  - aborting 360
  - identifying number processed 332
  - locking 364, 444
  - processing 363, 448
  - routing 394
  - sending 336
  - sending and receiving 358
  - sending to remote server 336
  - sending to transport 347, 349
  - unlocking 370
- required fields 581
- restarting plugin engine 437
- restructuring tables 562
- returning
  - alias names 130
  - current time 157
  - name of table 246
  - number of parameters 259
  - parameter name 258
  - protocol class 550
  - task status 158
- returning list of alias names 156
- run-time errors
  - file could not be opened 490
- run-time errors (continued)
  - invalid field values when posting new record 491
  - Server is too slow 491
  - SysTools date field returns invalid date 492
  - TffTable.RecNo Always Returns -1 492

## S

- saving data dictionary 600
- scalability 3
- schema file
  - DATMASK keyword 536
  - FIELD keyword 538

- schema file (continued)
  - FILETYPE keyword 535
  - RECLENGTH keyword 536
- script file 47
- searching, using ranges with null fields 492
- secure mode 25
- security 1, 25
- SELECT 481
- server
  - adding database alias 149
  - adding functionality 300
  - choosing 160
  - connecting data module 104
  - connection to transport 331
  - creating 277
  - creating a permanent alias 554
  - database connection 180
  - defining exclusive access to database 184
  - deleting alias 153
  - deleting existing alias 557
  - deleting given alias 154
  - extending 310, 439
  - filter timeout 223
  - finding 161
  - freeing resources 132
  - identifying connection to transport 356
  - indicating connection to client 137
  - login 162
  - overriding stored name 549
  - retrieving current date/time values 558
  - retrieving name 331, 355
  - retrieving path for alias 155
  - returning current date 157
  - returning database identifier 183
  - returning default name 550
  - scripting 47
  - start minimized 36
  - starting automatically 37
  - statistics 34, 495
  - storing name in registry 551
  - task status 158
  - verifying name 325

- server application 2
- server engine
  - accessing configuration 408
  - assigning unique identifier to dataset 206
  - connecting to 138
  - creating engine extender 311
  - creating engine monitor 310
  - extending 310
  - identifying 459
  - identifying unique ID assigned to session 163
  - integrating into the FlashFile Server 313
  - overview 399
  - providing access to command handler 406
  - read-only 403
  - referencing 163, 188
  - restarting 431, 437
  - retrieving list of 434
  - returning buffer manager 408
  - returning interested monitors 402
  - returning list of remote server 403
  - returning name 409
- server engine (continued)
  - returning number of connected command handlers 406
  - returning remote server names 409, 412
  - shutting down 431, 438
  - specifying 394
  - specifying directory 408
  - specifying protocol used 143
  - specifying SQL engine 409
  - specifying transport 412
  - specifying unique identifier 135
  - starting 432, 438
  - stopping 432, 438
  - writing to disk 404
- server object classes 441
- server table, converting 505
- session
  - accessing database components 153
  - choosing server 160
  - closing database 151
  - deleting alias from server 153
  - finding server 161
  - identifying unique ID 163
  - indicating default 159
  - name 189, 228
  - number of database components 152
  - referencing server engine 163
  - retrieving list of alias names 155
  - retrieving timeout value 159
  - returning default 126, 129, 130
  - returning list of database names 156
  - returning list of names 130
  - returning name 128
  - setting password 164
  - specifying 189
  - specifying active 148
  - specifying client name 152
  - specifying name 163
  - specifying number of login retries 164
  - specifying timeout 165
  - specifying UserID 164
  - starting 162
- session component
  - generating unique name 150
  - returning number attached to client 138
- setting
  - autoinc value 566
  - login parameters 164, 564
  - login retries 565
  - transaction mode 563
- setting range end 245
- setting range start 245
- ShareMem 520
- Single User Protocol 26
- singleEXE
  - applications 570
  - faster 5

- singleEXE applications, adding aliases 571
- specifying
  - columns to use as an index 241
  - data source component 257
  - Password 164
- SQL
  - calculating average 472
  - calculating character length 472
  - converting all characters to uppercase 484
  - converting string to all lowercase 475
  - evaluating to NULL 473
  - executing on server 417, 423
  - extracting substring 482
  - formatting time value 530
  - key words 471
  - preparing for execution 259
  - removing leading and/or trailing characters 484
  - result set 255
  - retrieving data from table 478
  - retrieving largest value in column or expression 476
  - retrieving smallest value in column or expression 476
  - returning current date 474
  - returning current date and time 475
  - returning current time 474
  - returning NULL if operands are equal 477
  - returning position of substring 477
  - returning row count 473
  - returning specified portion of datetime field 475
  - supported functions 468
  - supported statements 467
  - syntax conventions 464
  - syntax definitions 472
  - totaling numeric values 483
- SQL clause
  - GROUP BY 481
  - HAVING 481
  - ORDER BY 481
  - SELECT 481
  - WHERE 481
- SQL engine
  - allocating resources 415, 417, 422, 423
  - executing SQL statement 416, 422
  - freeing resources 418, 425
  - preparing SQL statement for execution 418, 425
  - setting parameters 419, 426
  - specifying 409
- SQL reference 463
- SQL supported functions
  - aggregate 468
  - conversion 468
  - date 469
  - other functions 470
- starting plugin engine 432, 438
- state engine
  - change event 85
  - current state 87
  - diagram 82
  - initializing 85
  - plugin engine 450
  - shutdown 85
- state engine (continued)
  - shutting down 86
  - starting 86
  - stopping 86
- stopping plugin engine 432, 438
- storing fields 220
- string grid
  - cell focus event 99
  - cell losing focus 99
  - copying cell contents 98
  - defined 96



- string grid (continued)
  - deleting cell contents 97
  - disabling redrawing 97
  - empty row 97
  - enabling redrawing of grid 98
  - indicating empty cell 98
  - preserving row contents 101
  - restoring row 100
  - sorting columns 99
  - specifying empty row 100
  - specifying filled row 100
- suggested reading 17
- supported functions
  - string 469
- supported SQL data types 466
- supported SQL statements 467
- symbols 14, 15
- system requirements 6

## T

- table
  - adding index 234, 235, 556
  - applying filter 210
  - applying range 236, 244
  - autoinc value 566
  - calculated fields 107
  - cancelling applied range 237
  - cancelling conversion 512
  - changing definition 562
  - changing structure 226
  - closing all open in database 167
  - composite indexes 65
  - converting 499
  - creating 181, 237, 557
  - creating master/detail relationship 110
  - defining fields 61
  - defining fields in master table 250
  - defining key field count 242
  - defining master table 250
  - defining read-only access 243

- table (continued)
  - deleting 208
  - deleting index 237
  - emptying 209
  - enabling setting of keys 244
  - exclusive access 239
  - finding nearest record 239
  - finding records 114
  - finding specific record 239
  - importing data 65, 534
  - list of names 185
  - lookup fields 109
  - name 246
  - name of index 242
  - names of all indexes 240
  - number of items in index fields 241
  - packing 171, 560
  - placing dataset into key edit mode 238
  - placing dataset into key edit mode for
    - the end of a range 238
  - placing dataset into key edit mode for
    - the start of a range 238
  - positioning cursor 240
  - referencing index definitions 240
  - reindexing 172, 561
  - removing deleted records 224, 560
  - renaming 225
  - restructuring 562
  - retrieving data dictionary 169
  - retrieving name 157
  - returning key size 242
  - setting base name 598
- table (continued)
  - size 4
  - specifying columns to use as an index
    - 241
  - specifying number 183
  - using filters 112
  - verifying existence 175
- table context menu
  - viewing definition 67

- task status
  - getting latest data 559
  - packing 171, 560
- TBatchMove support 487
- TCP/IP 3, 38, 330
- TDataSet Descendant Model 26
- technical support 16
- TffTimerThread, creating an instance 381
- thread
  - accessing TffRequest 364
  - aquiring from pool 375
  - blocking 370
  - calling specified event 379
  - identifying active 373
  - identifying transport used 327
  - terminating 379, 381
  - unblocking 371
- thread pool
  - defining initial number of threads 374
  - defining maximum number of threads 374
  - identifying available threads 374
  - identifying unfilled slots 373
  - indicating 350
  - removing inactive threads 373
- timeout
  - value 170
- timer event, time between invocations 382
- transaction
  - corrupt 175
  - defined 27
  - explicit 177
  - fail-safe 563
  - mode, setting 563
  - new 174
  - starting 175
  - terminating 173
- transaction journal recovery 2, 45
- transaction management 2
- transaction manager 28
- transport
  - aborting requests 360
  - appending reply data 360
  - blocking thread 370
  - client termination request 335
  - connections 317
  - creating instance of TffRequest 361
  - defining command handler 327
  - determining mode 332
  - discarding property modifications 326
  - establishing connection 329, 353
  - expecting a reply 366
  - finding new uses 279
  - forcing temporary disconnection 343
  - handling requests from clients 332
  - identifying 327
  - identifying connection to server 356
  - identifying number of updates 344
  - identifying protocol 357
  - indicating supported drivers 343
  - indicating supported operating system 343
  - indicating thread pool 350
  - initializing variables 324
  - losing connection 334, 356
  - messages 330
  - modes 316
  - modifying properties 323
  - name 330, 354
  - number of connections 327, 353
  - number of requests processed 332
  - processing request 390
  - processing request from client 338
  - property modifications 328
  - providing access to 392
  - receiving requests 328
  - remote server connection 104
  - resetting message count 341
  - responding to broadcast requests 342
  - response waiting time 369

- transport (continued)
    - retrieving 435
    - returning error code 363
    - sending a reply 339
    - sending a request 347, 349
    - sending a request to a remote server 340
    - sending and receiving requests 358
    - sending request to remote server 336
    - sending requests 328
    - sending requests to server 332
    - server connection 331
    - specifying 412
    - specifying client 361
    - specifying data 365, 367
    - specifying data length 365, 367
    - specifying message identifier 366
    - specifying message type 364
    - specifying number of bytes sent 361
    - specifying remote server name 342
    - specifying starting position 369
    - states 316
    - terminating connection 343, 358
    - terminating exclusive access 370
    - transferring buffer variables 326
    - unblocking thread 371
    - verifying listen mode 324
    - verifying number 392
    - verifying send mode 325
    - verifying server name 325
  - transport (continued)
    - waiting for a reply 340, 349
    - writing messages to EventLog 330
    - writing to event log 363
  - troubleshooting 486
  - type of
    - field 585
    - file 589
  - type of (continued)
    - index 594
- ## U
- UNC path 41
  - units 8
  - units for field 586
  - updates, identifying number of 344
  - updating parameter list 258
  - user ID definitions 32
  - user name, specifying 140
  - user permissions 39
  - user-defined index
    - adding 579
    - configuring 41
  - user-defined index descriptions 32
- ## V
- variable, initializing 85



The TurboPower family of tools—  
Winners of 6 *Delphi Informant* Readers' Choice Awards for 2000!

For over fifteen years you've depended on TurboPower to provide the best tools and libraries for your development tasks. Now try *Abbrevia* and *SysTools*—two of TurboPower's best selling products risk free. Both are fully compatible with *Delphi* and *C++Builder*, and are backed with our expert support and 60-day money back guarantee.

DEVELOP, DEBUG, OPTIMIZE  
FROM START TO FINISH, TURBOPOWER  
HELPS YOU BUILD YOUR BEST

## ABBREVIA™

*With Abbrevia 2, it's easy to add full featured PKZIP and MS-CAB compatible data compression to your programs. Just drop in the native VCL components and you're ready to go. Abbrevia 2 includes all-new visual controls, our professional COM Objects for web development, enhanced spanning and more!*

## SYSTOOLS™

*SysTools 3 means never having to write the same old routines again. That's because it's the only library with more than 1000 reliable, optimized, time-tested routines you'll use in virtually every project you build. For everything from low-level system access to high-level financial calculations, SysTools is the one product that will pay for itself the first time you use it.*



Try the full range of  
TurboPower products.  
Download free Trial-Run Editions  
from our Web site.

[www.turbopower.com](http://www.turbopower.com)

FlashFiler 2 requires Microsoft Windows 9X, Me, NT or 2000, and Borland Delphi 3 and above or C++Builder 3 and above.

**TurboPower Software Company**  
©2000, TurboPower Software Co.



**TURBOPOWER®**  
Software Company