InterBase 6

Developer's Guide

Borland/INPRISE

100 Enterprise Way, Scotts Valley, CA 95066 http://www.interbase.com

1INT0050WW21003 6E1R0699

Table of Contents

List of Tables

List of Figures

CHAPTER

1

Using the InterBase Developer's Guide

The InterBase *Developer's Guide* focuses on the needs of developers who use the Borland development tools: Delphi, C++ Builder, and JBuilder. It includes discussions of writing UDFs, writing install programs using InterBase's Install API, programming with the ODBC driver. It is also an extensive guide to using InterBase Express (IBX) data access components.

Who should use this book

The *Developer's Guide* assumes a general familiarity with SQL, data definition, data manipulation, and programming practice. It is a resource for:

- Programmers programmers writing applications with Borland's Delphi, C++ Builder, or JBuilder.

Topics covered in this book

The following table lists the chapters in the *Developer's Guide* and provides a brief description of each one:

Chapter	Description
Chapter 1, "Using the InterBase Developer's Guide"	Introduces the book, and describes its intended audience
Chapter 2, "Client/Server Concepts"	Describes the architecture of client/server systems using InterBase, including the definition of client and server, and options for application development
Chapter 3, "Programming Applications with Delphi or C++ Builder"	Describes programming InterBase applications using the Borland Database Engine (BDE) with Delphi and C++ Builder
Chapter 4, "Programming Applications with JBuilder"	Describes building InterBase applications using InterClient, InterServer, and JBuilder
Chapter 5: "Programming Applications with ODBC"	Describes programming InterBase applications with ODBC and OLE DB, including programming with the ODBC driver and configuring and using ODBC datasources
Chapter 6, "Working with UDFs and Blob Filters"	Describes how to write UDFs, create UDF libraries, declare the functions to the database, and call the functions; includes a discussion of Blob filters
Chapter 7, "Using the Install and Licensing APIs"	Describes how to use the functions in the Licensing and Install APIs to write install applications
Chapter 8: "Introduction to IBX"	Introduces the InterBase Express (IBX) data access components
Chapter 9: "Designing Database Applications"	Describes some common considerations for designing a database application and the decisions involved in designing a user interface, including how to use databases and database architecture
Chapter 10: "Building One- and Two-tiered Applications"	Describes one- and two tiered applications, datasets, and transactions

TABLE 1.1 Chapters in the *Developer's Guide*.

Chapter	Description
Chapter 11: "Connecting to Databases"	Describes database components and how to manipulate database connections
Chapter 12: "Understanding Datasets"	Describes the functionality of *TDataSet* that is inherited by the dataset objects used in database applications
Chapter 13: "Working with Tables"	Describes how to use the *TIBTable* dataset component in your database applications
Chapter 14: "Working with Queries"	Describes how to use the *TIBQuery* dataset component in your database applications
Chapter 15: "Working with Stored Procedures"	Describes how to use InterBase stored procedures in your database applications
Chapter 16: "Working with Cached Updates"	Describes when and how to use cached updates, as well as the *TIBUpdateSQL* component, which can be used in conjunction with cached updates to update virtually any dataset
Chapter 17: "Debugging with SQL Monitor"	Describes how to use the *TIBSQLMonitor* component to monitor the dynamic SQL that passes through the InterBase server
Chapter 19: "Working with InterBase Services"	Describes how to build the InterBase services into your applications
Chapter 20: "Programming with Database Events"	Describes how to use a *TIBEvents* component in your IBX-based application to register interest in and handle InterBase server events.
Chapter 21: "Writing Installation Wizards"	Describes how to use the *TIBSetup*, *TIBInstall*, and *TIBUninstall* components to build an InterBase installation program
Chapter 22: "Migrating to InterBase Express"	Describes the process of migrating from BDE-based database applications to IBX applications
Appendix A, "Differences Between Delphi Components and InterBase Components"	Describes the new properties, methods, and events included in the IBX components, as well as the properties, methods,and events not included in IBX

TABLE 1.1 Chapters in the *Developer's Guide*.

CHAPTER

2

Client/Server Concepts

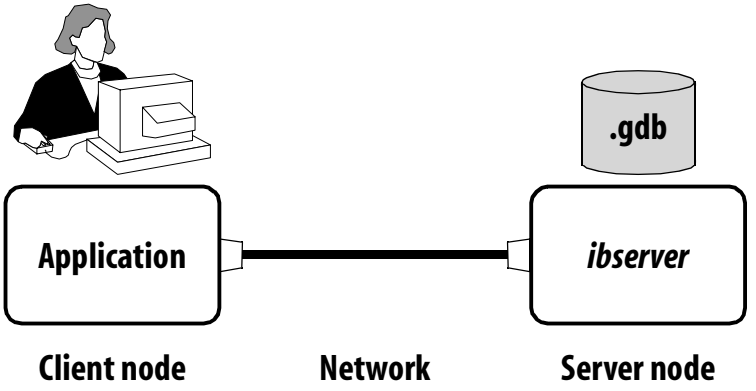
This chapter describes in high-level language the architecture of client/server systems using InterBase. The chapter covers topics including the definition of client and server, and options for application development.

Definition of a client

An InterBase client is an application, typically written in C, C++, Delphi or Java, that accesses data in an InterBase database.

In the more general case, an InterBase client is any application process that uses the InterBase client library, directly or via a middleware interface, to establish a communication channel to an InterBase server. The connection can be *local* if the application executes on the same node as the InterBase server, or *remote* if the application must use a network to connect to the InterBase server.

FIGURE 2.1 Basic client/server relationship



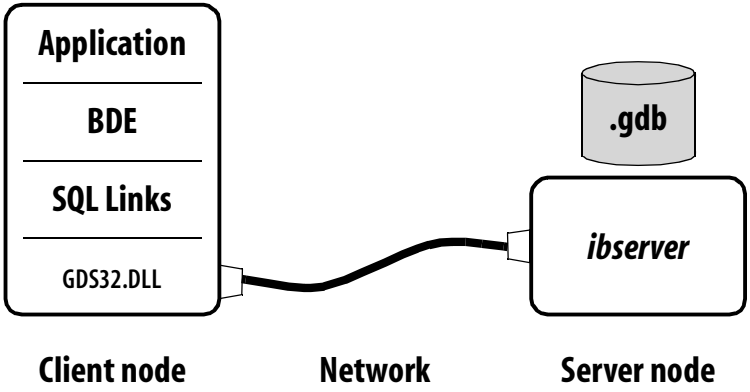
Note InterBase is designed to allow clients running one operating system to access an InterBase server on a different platform and operating system from the client. For example, a common arrangement is to have inexpensive Windows 98 PCs acting as client workstations to concurrently access a departmental server running Windows NT, NetWare, or any of several vendors of UNIX.

The InterBase client library

The InterBase client library is a library that developers of client applications use to initiate connections to a server and to programmatically perform database operations. The library uses the operating system client network interface to communicate with one or more InterBase servers, and implements a special InterBase client/server application protocol on top of a network protocol (see "Network protocols" in the *Operations Guide*).

The client library provides a set of high-level functions as an Application Programmer's Interface (API) for communication with an InterBase server. Any client application or middleware must use the API to access an InterBase database. The *API Guide* provides reference documentation and guidelines for using the API to develop high-performance applications.

FIGURE 2.2 Role of the InterBase client library



Definition of a server

The InterBase server is a software process that executes on the node that hosts the storage space for databases. The server process is the only process on any node that can perform direct I/O to the database files.

Clients send to the server process requests to perform several different types of actions on the database, including:

- Search the database based on criteria
- Collate, sort and tabulate data
- Return sets of data
- Modify data values
- Insert new data into the database
- Remove data from the database
- Create new databases or data structures
- Execute procedural code on the server
- Send messages to other clients currently connected

The server process is fully network-enabled; it services connection requests that originate on another node. The server process implements the same InterBase application protocol that the client uses.

Many clients can remain connected to the multithreaded server process simultaneously. The server regulates access to individual data records within the database, and enforces exclusive access to records when clients request to modify the data in the records.

Application development

Once you create and populate a database, you can access the information through an application. If you use one of Borland's client tools, you can access information through your existing application. You can also design and implement a new application by embedding SQL statements or API calls in an application written in a programming language such as C or C++.

Borland client tools applications

Client/server versions of Borland client tools such as Delphi, Borland C++, Paradox, and Visual dBASE can access InterBase databases using Borland SQL Links. Server query reporting is built into the client tool providing Windows application support. This enables you to build sophisticated, user-friendly database applications with minimal programming effort.

InterBase Express (IBX) for Delphi

IBX is a library of Delphi components that allows Delphi programmers to access InterBase features without having to go through the Borland Database Engine (BDE). The version of IBX that ships with Delphi 5 accesses only InterBase 5.x databases. An enhanced version of IBX ships with InterBase 6. This version includes components for accessing the InterBase 6 Install API and Licensing API when installed in Delphi.

The Borland Database Engine

Most Borland application development tools use middleware technology based on the *Borland Database Engine* (BDE). The BDE is a library that provides a unified API for applications to interface programmatically with the database client library of any database vendor for which there is an *SQL Links* driver available. For instance, a C++ application programmer uses the BDE functions to access data from a BDE *alias*. The programmer configures the BDE alias to use the InterBase driver for SQL Links, and this configuration leads BDE to dynamically load the appropriate library that implements BDE functions with equivalent functions in the InterBase API.

The most important advantage is that application engineers can write code that is independent from a given proprietary database product API, and thereby reduce porting expense if project requirements call for the engineer to change database server technology. For instance, porting an application from using Paradox tables to an InterBase database can be accomplished in large part simply by reconfiguring the BDE alias to use the appropriate SQL Links driver, and specifying the path of the InterBase server and database.

The BDE technology has an internal architecture that implements features to accommodate database technologies that do not offer those features.

- The interaction between BDE's caching and InterBase's own caching is confusing. Client-side caching gives a lot of benefit with little associated cost when the database resides on the same machine as the client, and the volume of data is low. Applying client-side caching in a client/server system with datasets that are greater in size by orders of magnitude can result in poor network performance as the client refreshes its cache over a network. See "Configuring the Superserver cache" in the *Operations Guide*.
- The differences between the BDE's Local SQL interpreter and InterBase's server-side SQL interpreter are also subtle. For consistency's sake, you should configure applications to pass SQL statements through the BDE and on to the server's SQL interpreter.

Embedded applications

You can write your own application using C or C++, or another programming language, and embed SQL statements in the code. You then preprocess the application using **gpre**, the InterBase application development preprocessor. **gpre** takes SQL embedded in a host language such as C or C++, and generates a file that a host-language compiler can compile.

The preprocessor matches high-level SQL statements to the equivalent code that calls functions in InterBase's client API library. Therefore, using embedded SQL affords the advantages of using a high-level language, and the runtime performance and features of the InterBase client API.

For more information about compiling embedded SQL applications, see the *Embedded SQL Guide*.

Predefined database queries

Some applications are designed with a specific set of requests or tasks in mind. These applications can specify exact SQL statements in the code for preprocessing. The **gpre** preprocessor translates statements at compile time into an internal representation. These statements have a slight speed advantage over dynamic SQL, since they do not need to incur the overhead of parsing and interpreting the SQL syntax at runtime.

Dynamic applications

Some applications need to handle ad hoc SQL statements entered by users at run time; for example, allowing a user to select a record by specifying criteria to a query. This requires that the program construct the query based on user input.

InterBase uses Dynamic SQL (DSQL) for generating dynamic queries. At run time, your application passes DSQL statements to the InterBase server in the form of a character string. The server parses the statement and executes it.

BDE provides methods for applications to send DSQL statements to the server and retrieve results. ODBC applications rely on DSQL statements almost exclusively, even if the application interface provides a way to visually build these statements. For example, Query By Example (QBE) or Microsoft Query provide convenient dialogs for selecting, restricting and sorting data drawn from a BDE or ODBC data source, respectively.

You can also build templates in advance for queries, omitting certain elements such as values for searching criteria. At run time, supply the missing entries in the form of *parameters* and a buffer for passing data back and forth.

For more information about DSQL, see the *Embedded SQL Guide*.

API applications

The InterBase API is a set of functions that enables applications to construct and send SQL statements to the InterBase engine and receive results back. All database work can be performed through calls to the API.

Advantages of using the InterBase API

While programming with the API requires an application developer to allocate and populate underlying structures commonly hidden at the SQL level, the API is ultimately more powerful and flexible. Applications built using API calls offer the following advantages over applications written with embedded SQL:

- Control over memory allocation
- Simplification of compiling procedure—no precompiler

- Access to error messages
- Access to transaction handles and options

API function categories

API functions can be divided into seven categories, according to the object on which they operate:

- Database attach and detach
- Transaction start, prepare, commit, and rollback
- Blob calls
- Array calls
- Database security
- Informational calls
- Date and integer conversions

The *API Guide* has complete documentation for developing high-performance applications using the InterBase API.

The Install API and the Licensing API

The Install API provides a library of functions that enable you to install InterBase programmatically. You have the option of creating a silent install that is transparent to the end user. The functions in the Licensing API permit you to install license certificates and keys as well.

Multi-database applications

Unlike many relational databases, InterBase applications can use multiple databases at the same time. Most applications use only one database, but others need to use several databases that could have the same or different structures.

For example, each project in a department might have a database to keep track of its progress, and the department could need to produce a report of all the active projects. Another example where more than one database would be used is where sensitive data is combined with generally available data. One database could be created for the sensitive data with access to it limited to a few users, while the other database could be open to a larger group of users.

With InterBase you can open and access any number of databases at the same time. You cannot join tables from separate databases, but you can use cursors to combine information. See the *Embedded SQL Guide* for information about multi-database applications programming.

CHAPTER

3

Programming Applications with Delphi or C++ Builder

This chapter covers programming InterBase applications using the Borland Database Engine (BDE) with Delphi or C++ Builder. Both Dephi and C++ Builder ship with extensive online documentation on programming database applications, and you should use that documentation as your main source of information This chapter describes how to best use these programs with InterBase, including:

- **Optimizing the InterBase SQL Links driver**
- **Working with TTable and TQuery**
- **Using generators**

Note With the introduction of InterBase Express (IBX) in Delphi 5, it is now possible to create InterBase applications without the overhead of the BDE. Part II of this book describes how to use the IBX components. For more information, see **"Introduction to IBX" on page 125**.

Optimizing the InterBase SQL Links driver

Use the BDE Administrator to configure the InterBase SQL Links driver. To start the BDE Administrator, select it from the Borland Delphi or C++ in the Programs menu. To view the InterBase driver definition, click on the Configuration tab, and then expand Drivers and Native from the Configuration tree. Click on INTRBASE to display the InterBase driver settings.

There are three options you can change to optimize the InterBase driver:

- DRIVER FLAGS
- SQLPASSTHRU MODE
- SQLQUERY MODE

These are discussed in the following sections.

Setting the driver flags

Depending on your database needs, you should set the DRIVER FLAGS option to either 512 or 4608 to optimize InterBase. The recommended value for DRIVER FLAGS is 4608.

- If you set DRIVER FLAGS to 512, you specify that the default transaction mode should be repeatable read transactions using hard commits. This reduces the overhead that automatic transaction control incurs.
- If you set DRIVER FLAGS to 4608, you specify that the default transaction mode should be repeatable read transactions using soft commits. Soft commits are an InterBase feature that lets the driver retain the cursor while committing changes. Soft commits improve performance on updates to large sets of data.

  When using hard commits, the BDE must re-fetch all records in a dataset, even for a single record change. This is less expensive when using a desktop database, because the data is transferred in core memory. For a client/server database such as InterBase, refreshing a dataset consumes the network bandwidth and degrades performance radically. With soft commit, the cursor is retained and a re-fetch is not performed.

  Note Soft commits are never used in explicit transactions started by BDE client applications. This means that if you use the *StartTransaction* and *Commit* methods to explicitly start and commit a transaction, then the driver flag for soft commit is ignored.

Setting the SQL pass-through mode

The SQLPASSTHRU MODE option specifies whether the BDE and passthrough SQL statements can share the same database connections. By default SQLPASSTHRU MODE is set to SHARED AUTOCOMMIT. To reduce the overhead that automatic transaction control incurs, set this option to SHARED NOAUTOCOMMIT.

If, however, you want to pass transaction control to your server, set this option to NOT SHARED. Depending on the quantity of data, this can increase InterBase performance by a factor of ten.

The recommended setting for this option is SHARED NOAUTOCOMMIT.

Setting the SQL query mode

Set the SQLQRYMODE to SERVER to allow InterBase, instead of the BDE, to interpret and execute SQL statements.

Working with TTable and TQuery

Although *TTable* is very convenient for its RAD methods and its abtract data-aware model, it should never be used with InterBase. *TTable* is not designed to be used with client/server applications; it is designed for use on relatively small tables in a local database, accessed in core memory.

TTable gathers information about the metadata of a table, and tries to maintain a cache of the dataset in memory. It refreshes its client-side copy of the data when you issue the *Post* method or the *TDatabase.Rollback* method. This incurs a huge network overhead for most client/server databases, which have much larger datasets and are accessed over a network. In a client/server architecture, you should use *TQuery* instead.

Setting TQuery properties and methods

Set the following *TQuery* properties and methods as indicated to optimize InterBase performance:

- *CachedUpdates* property: set this property to *False* to allow the server to handle updates, deletes, and conflicts.

- *RequestLive* property: set this property to *False* to prevent the VCL from keeping a client-side copy of rows; this has a benefit to performance because less data must be sent over the network

  In a client/server configuration, a "fetch-all" severely affects database performance, because it forces a refresh of an entire dataset over the network. Here are some instances in which cause a *TQuery* to perform a fetch-all:

- *Locate* method: you should only use *Locate* on local datasets
- *RecordCount* property: although it is nice to get the information on how many records are in a dataset, calculating the *RecordCount* itself forces a fetch-all.
- *Constraints* property: let the server enforce the constraint.
- *Filter* property: let the server do the filtering before sending the dataset over the network.
- *Commit* method: forces a fetch-all when the BDE DRIVER FLAGS option is not set to 4096 (see **"Setting the driver flags" on page 32**), or when you are using explicit transaction control.

Using generators

Using an InterBase trigger to change the value of a primary key on a table can cause the BDE to produce a record or key deleted error message. This can be overcome by adding a generator to your trigger.

For example, when your client sends a record to the server, the primary key is NULL. Using a trigger, InterBase inserts a value into the primary key and posts the record. When the BDE tries to verify the existence of the just-inserted record, it searches for a record with a NULL primary key, which it will be unable to find. The BDE then generates a record or key deleted error message.

To get around this, do the following:

1. Create a trigger similar to the following. The "if" clause checks to see whether the primary key being inserted in NULL. If so, a value is produced by the generator; if not, nothing is done to it.

```
Create Trigger COUNTRY_INSERT for COUNTRY
active before Insert position 0
as
begin
  if (new.Pkey is NULL) then
  new.Pkey = gen_id(COUNTRY_GEN,1);
end^
```

2. Create a stored procedure that returns the value from the generator:

```
Create Procedure COUNTRY_Pkey_Gen returns (avalue INTEGER)
as
begin
 avalue = gen_id(COUNTRY_GEN,10);
end^
```

3. Add a *TStoredProc* component to your Delphi or C++ Builder application and associate it with the `COUNTRY_Pkey_Gen` stored procedure.
4. Add a *TQuery* component to your application and add the following code to the *BeforePost* event:

```
If(TQuery.state = dsinsert) then
begin
  StoredProc1.ExecProc;
  TQuery.FieldByName('Pkey').AsInteger :=
   StoredProc1.ParamByName('avalue').AsInteger;
end;
```

This solution allows the client to retrieve the generated value from the server using a *TStoredProc* component and an InterBase stored procedure. This assures that the Delphi or C++ Builder client will know the primary key value when a record is posted.

CHAPTER

4

Programming Applications with JBuilder

This chapter covers building InterBase database applications with InterClient, InterServer, and JBuilder, including:

- **Installing InterClient classes into JBuilder**
- **Installing and administering InterServer**
- **Programming with InterClient**
- **Troubleshooting InterClient programs**
- **Deploying InterClient programs**
- **InterClient/JDBC compliance specifications**

Installing InterClient classes into JBuilder

InterClient is an all-Java thin-client JDBC driver specifically designed to access InterBase databases. It is included in the JBuilder Client/Server product, but can be used with the Professional version as well, and is available free from the web at www.interbase.com. JBuilder rapid application development design makes integrating JDBC drivers easy, and InterClient is no exception.

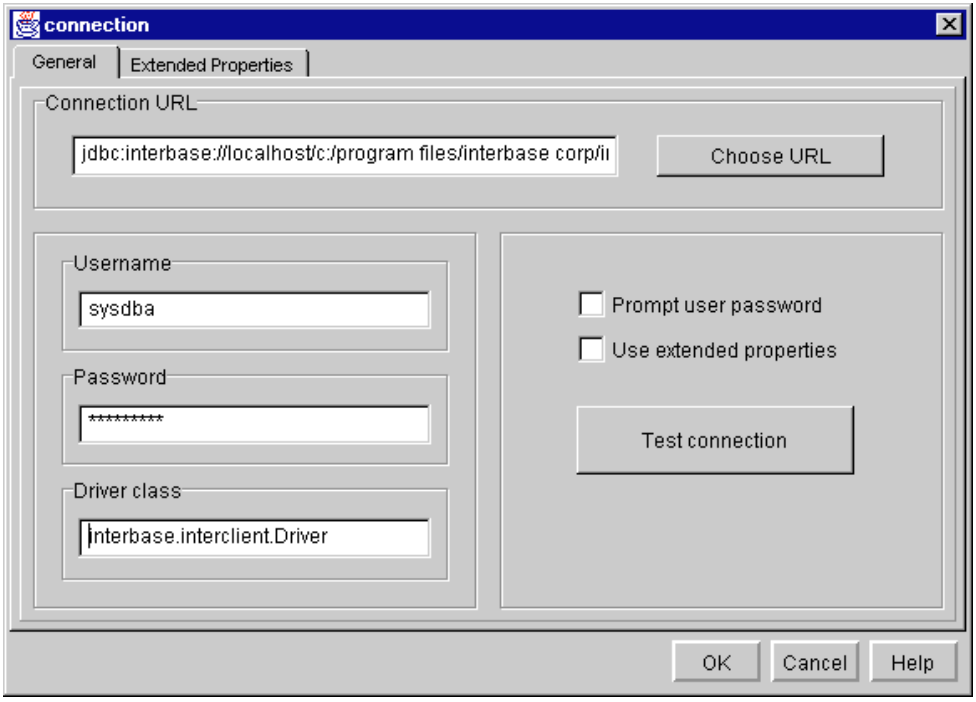
When you install InterClient (version 1.11 or later), you will need modify the **JBUILDER.INI** file as outlined in the InterClient *JBuilder Integration Notes*. Without these changes, you will not be able to run InterClient-based applications from within the JBuilder IDE.

Database application basics

If you want your JBuilder application or applet to connect to a database, use a *Database* component to establish the connection, a *DataSet* component (such as a *TableDataSet* or *QueryDataSet* component) to provide the data, and a data-aware control to display the results (such as a *GridControl*). Follow these steps for any JDBC driver. What distinguishes InterClient from other JDBC drivers is the values you specify for the connection parameters of the *Database* component.

When you edit the connection properties of a Database component, JBuilder displays the Connection dialog.

FIGURE 4.1 Connection dialog



To connect to an InterBase database with your Java application/applet, you need to specify the following connection parameters: the name of a JDBC driver class, a username, a password, and a connection URL. The name of the InterClient JDBC driver class is always the same:

```
interbase.interclient.Driver
```

Spelling and capitalization are important. If you spell the driver class incorrectly, you may get a *ClassNotFoundException*, and consequently, a "No suitable driver" error when the connection is attempted. The username and password parameters are the same that you would use when connecting to a database with IBConsole or any other tool. For the sake of simplicity, these examples use *sysdba* (the InterBase *root* user) and *masterkey* for username and password, respectively.

There are other useful features of this dialog, as well. Once you fill in your URL, you can press the **Test connection** button to ensure that the connection parameters are correct. The **Prompt user password** check box forces the user to enter a proper username and password before establishing a connection. The **Use extended properties** check box and property page is not used by InterClient.

Using JDBC URLs

The JDBC URL is the parameter used to locate the actual database to which you want to connect. A JDBC URL can be broken down into three parts, all separated by colons: the keyword *jdbc*, the subprotocol name, and the datasource name or location. The *jdbc* keyword is needed to distinguish JDBC URLs from other URLs, such as those for HTTP or FTP. The subprotocol name is used to select the proper JDBC driver for the connection. Every JDBC driver has its own subprotocol name to which it responds. InterClient URLs always have a subprotocol of *interbase*. Other JDBC drivers have their own unique subprotocol names, for example, the JDBC-ODBC Bridge answers JDBC URLs with the subprotocol of *odbc*.

The third part of a InterClient URL holds the name of the server that is running InterServer and the location (relative to InterServer) of the database to which you want to connect. The syntax for an InterClient URL is as follows:

```
jdbc:interbase://servername/pathToDatabase.gdb
```

Here are a few possible configuration options and their corresponding JDBC URLs.

For the **atlas** database on a Unix machine named *sunbox* you might use something like this (the path on the Unix machine is **/usr/databases/atlas.gdb**):

```
jdbc:interbase://sunbox//usr/databases/atlas.gdb
```

To access the **employee** database on an NT machine named *mrbill*, you might use something like this (notice the drive letter):

```
jdbc:interbase://mrbill/c:/interbas/examples/employee.gdb
```

These examples assume that InterServer and InterBase are running on the same machine, which is the fastest type of connection. If InterServer is running on *sunbox* and InterBase is running on *mrbill*, you could use this URL to get to the NT machine through the Unix machine :

```
jdbc:interbase://sunbox/mrbill:c:/interbas/examples/employee.gdb
```

Tip This last example is one of the best configurations for web applications that require high load and security. Because InterServer listens on port 3060, and InterBase itself listens on port 3050, you could use this in a firewall scenario.

If the client and the server are on the same machine and you wanted to make a local connection, use *loopback* as the server name. For example, in NT:

```
jdbc:interbase://loopback/c:/interbas/examples/employee.gdb
```

Other than these connection-specific issues, InterClient can be used like any other JDBC driver with JBuilder. With Local InterBase, JBuilder Professional and Client/Server versions, it makes it easy to develop and test powerful database applications in Java.

Note Currently, there is no true local access for Solaris and HP.

Installing and administering InterServer

InterServer is a server-side driver, which serves as a translator between the InterClient-based clients and the InterBase database server.

This section covers how to start, stop, and view information on InterServer, both as an application and a service.

Starting InterServer on Windows NT and 95

InterServer must be started to enable InterClient connections to a database. If InterServer is started as an application, an icon is displayed in the task tray, located on the right side of your task bar (or bottom, if your task bar is positioned vertically). The server can be configured to automatically start up at system boot.

To configure InterServer as an application or a service, select the InterServer Configuration Utility icon from the InterBase InterClient program group, and then select the startup mode:

- Manual Startup — You must select the InterServer icon from the InterBase InterClient program group to start InterServer.
- Windows Startup — Configures the server to automatically start up at system boot

The Advanced section of the Configuration Utility is disabled if InterServer is configured to run as an application.

If InterServer is configured to run as a service, you can start, stop, or pause it using the Advanced page of the InterServer Configuration Utility or with the Windows NT control panel services utility.

The Advanced page of the InterServer Configuration Utility gives you the following options:

- Stop InterServer is shut down and connections are closed
- Pause Connections between InterClient applications and the InterBase server are held open but all requests for service are halted
- Started Connections between InterClient applications and the InterBase server are running.

Shutting down InterServer

To shut down the InterServer when it is running as application, right click the InterServer icon in the task tray and choose Shutdown. If any connections are open, a warning message appears. If you have open connections, it is recommended that you close them before shutting down the server. You also must close all client applications you are running.

To shut down the InterServer service, press the stop light in the Advanced section of the Configuration Utility.

Viewing InterServer information and properties

To view current server information and properties, right click the InterServer application icon and choose Properties from the popup menu, or look at the General page of the InterServer configuration utility.

The Property page provides:

- Location of the InterClient installation directory.
- Version of the InterClient package.
- InterServer capabilities
- Operating system

The General page of the InterServer configuration utility provides the same information as the Startup Configuration page of the InterServer application, which is:

- InterServer version
- Server status.
- Operating system

- Location of the InterClient installation directory
- Server startup
- Startup mode

Programming with InterClient

As an all-Java JDBC driver, InterClient enables platform-independent, client/server development for the Internet and corporate Intranets. The advantage of an all-Java driver versus a native-code driver is that you can deploy InterClient-based applets without having to manually load platform-specific JDBC drivers on each client system. Web servers automatically download the InterClient classes along with the applets. Therefore, there is no need to manage local native database libraries, which simplifies administration and maintenance of customer applications. As part of a Java applet, InterClient can be dynamically updated, further reducing the cost of application deployment and maintenance.

InterClient allows Java applets and applications to:

- Open and maintain a high-performance, direct connection to an InterBase database server
- Bypass resource-intensive, stateless Web server access methods
- Allow higher throughput speeds and reduced Web server traffic

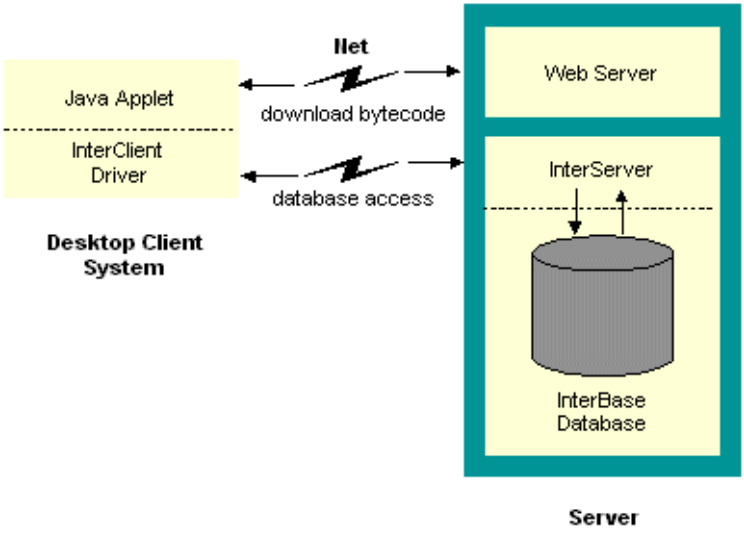
InterBase developers who are writing new Java-based client programs can use InterClient to access their existing InterBase databases. Because InterClient is an all-Java driver, it can also be used on the Sun NC (Network Computer), a desktop machine that runs applets. The NC has no hard drive or CD ROM; users access all of their applications and data via applets downloaded from servers.

InterClient architecture

The InterClient product consists of two major pieces:

- A client-side Java package, called InterClient, containing a library of Java classes that implement most of the JDBC API and a set of extensions to the JDBC API. This package interacts with the JDBC Driver Manager to allow client-side Java applications and applets to interact with InterBase databases.
- A server-side driver, called InterServer. This server-side middleware serves as a translator between the InterClient-based clients and the InterBase database server.

FIGURE 4.2 InterClient architecture



Developers can deploy InterClient-based clients in two ways:

- As Java applets, which are Java programs that can be included in an HTML page with the *<APPLET>* tag, served via a web server, and viewed and used on a client system using a Java-enabled web browser. This deployment method doesn't require manual installation of the InterClient package on the client system. It does require a Java-enabled browser and the JDBC Driver Manager to be installed on the client system.
- As Java applications, which are stand-alone Java programs for execution on a client system. This deployment method requires the InterClient package, the JDBC Driver Manager, and the Java Runtime Environment (JRE), which is part of the Java Developer's Kit (JDK) installed on the client system.

InterClient communication

InterClient is a driver for managing interactions between a Java applet or application and an InterBase database server. On a client system, InterClient works with the JDBC Driver Manager to handle client requests through the JDBC API. To access an InterBase database, InterClient communicates via a TCP/IP connection with an InterServer translator. InterServer forwards InterClient requests to the InterBase server and passes back the results to the InterClient process on the client machine.

Developing InterClient programs

This section provides a detailed description of how to use InterClient to develop Java applications, including:

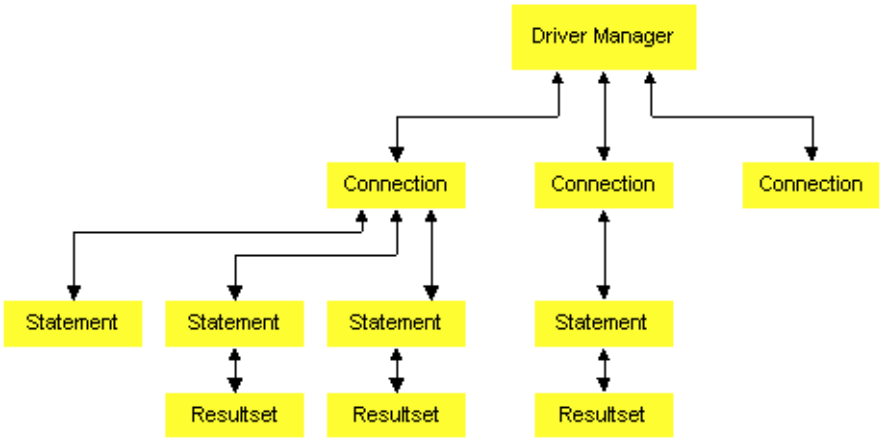
- Using the JDBC interfaces
- Using InterClient drivers
- Accessing InterClient extensions
- Opening a database connection
- Executing SQL statements

Using the JDBC interfaces

The JDBC API is a set of Java interfaces that allow database applications to open connections to a database, execute SQL statements, and process the results. These include:

java.sql.DriverManager	loads the specific drivers and supports creating new database connections
java.sql.Connection	represents a connection to a specific database
java.sql.Statement	allows the application to execute a SQL statement
java.sql.PreparedStatement	represents a pre-compiled SQL statement
java.sql.CallableStatement	represents a call to a stored procedure in the database
java.sql.ResultSet	controls access to the rows resulting from a statement execution

FIGURE 4.3 JDBC interfaces



Importing the InterClient classes

The InterClient classes provide the code that actually implements the JDBC API. The *java.sql* package defines the standard JDBC API interfaces. Importing this package allows you to reference all of the classes in the *java.sql* interface without first typing the "java.sql" prefix. For clarity's sake, this document prefixes all class names with "java.sql," but it isn't necessary if you import the package. You can import this package with the following line:

```
import java.sql.*;
```

The DriverManager class

The *DriverManager* class is part of the *java.sql* package. The JDBC framework supports multiple database drivers. The *DriverManager* manages all JDBC drivers that are loaded on a system; it tries to load as many drivers as it can find. For each connection request, it locates a driver to connect to the target database URL. The *DriverManager* also enforces security measures defined by the JDBC specification.

The Driver class

Each database driver must provide a *Driver* class that implements the *java.sql.Driver* interface. The *interbase.interclient.Driver* class is an all-Java implementation of a JDBC driver that is specific to InterBase. The *interbase.interclient* package supports most of the JDBC classes and methods plus some added extensions that are not part of the JDBC API.

To access an InterBase database, the InterClient driver communicates via a TCP/IP connection with an InterServer process. InterServer forwards InterClient requests to the InterBase server. InterBase processes the SQL statements and passes the results back to the InterServer, which then passes the results to the InterClient driver.

MULTI-THREADING

Any JDBC driver must comply with the JDBC standard for multi-threading, which requires that all operations on Java objects be able to handle concurrent execution.

For a given connection, several threads must be able to safely call the same object simultaneously. The InterClient driver is "thread-safe." For example, your application can execute two or more statements over the same connection concurrently, and process both result sets concurrently, without generating errors or ambiguous results.

The Connection class

After instantiating a *Driver* object, you can open a connection to the database when *DriverManager* gives you a *Connection* object. A database driver can manage many connection objects.

The *Connection* object establishes and manages the connection to your particular database. Within a given connection, you can execute SQL statements and receive the result sets.

The *java.sql.Connection* interface represents a connection to a particular database. The JDBC specification allows a single application to support multiple connections to one or more databases, using one or more database drivers. When you establish your connection using this class, the *DriverManager* selects an appropriate driver from those loaded based on the subprotocol specified in the URL, which is passed as a connection parameter.

About InterClient drivers

This section describes:

- **Loading the InterClient driver**
- **Explicitly creating the InterClient driver**

Loading the InterClient driver

The InterClient driver must be loaded before your application can attempt to connect to an InterBase database. To explicitly load the InterClient driver with the *DriverManager*, include the following line in your program before using the driver to establish a database connection:

```
Class.forName("interbase.interclient.Driver");
```

The first time the Java interpreter sees a reference to *interbase.interclient.Driver*, it loads the InterClient driver. When the driver is loaded it automatically creates an instance of itself, but there is no handle for it that lets you access that driver directly by name. This driver is "anonymous"; you do not need to reference it explicitly to make a database connection. You can make a database connection simply by using the *java.sql.DriverManager* class.

It's the responsibility of each newly loaded driver to register itself with the *DriverManager*; the programmer is not required to register the driver explicitly. After the driver is registered, the *DriverManager* can use it to make database connections.

Explicitly creating the InterClient driver

When writing a client program, you can interact either with the *DriverManager* class or with a database driver object directly. To reference an InterClient driver directly, you must explicitly create another instance (in addition to the anonymous one that's created automatically during loading) of the driver using the *java.sql.Driver* class:

```
java.sql.Driver driver = new interbase.interclient.Driver();
```

Now you can reference the driver classes and methods with "driver.XXX()." If all you need to do is connect to the database and execute SQL statements, you do not need to create a driver object explicitly; the *DriverManager* handles everything for you. However, there are a few cases when you need to reference the driver by name. These include:

- Getting information about the driver itself, such as a version number.
- Tailoring a driver for debugging purposes. For more information, see **"Debugging your application" on page 61**.

The *DriverManager* sees a driver as only one of many standard JDBC drivers that can be loaded. If you need to create a connection to another type of database in the future, you need only to load the new driver with *forName()* or declare another driver explicitly with

```
java.sql.Driver driver = new XXX.Driver
```

USING JAVA.SQL.DRIVER METHODS

The *java.sql.Driver* class has different methods than *java.sql.DriverManager*. If you want to use any of the *java.sql.Driver* methods, you need to create an explicit driver object. The following are a few of the driver methods:

- *getMajorVersion()* gets the driver's major version number
- *getMinorVersion()* gets the driver's minor version number

The example below shows how to interact with the database by referencing the driver directly:

```
//create the InterClient driver object as a JDBC driver
java.sql.Driver driver = new interbase.interclient.Driver();

//get the connection object
java.sql.Connection connection = driver.connect(dbURL, properties);

//reference driver to get the driver version number
java.sql.String version = driver.getMajorVersion() +
driver.getMinorVersion();
System.out.print("You're using driver", + version");
```

IMPORTANT Do not define a driver object as a type *interbase.interclient.Driver* as follows:

```
interbase.interclient.Driver driver = new interbase.interclient.Driver();
```

This method creates a driver object that is an instance of the *interbase.interclient.Driver* class, *not* a generic instance of the *java.sql.Driver* class. This method is inappropriate for a database-independent client program because it hard-codes the InterClient driver into your source code, together with all of the classes and methods that are specific to the InterClient driver. Because this code is not portable, you would not be able to use this program to provide access to another type of database (besides InterBase) in the future.

Even if InterBase is the only database connection you think you will ever use, do not be tempted to assign your JDBC objects (drivers, connections, statements, and so on) to *interbase.interclien*t. Again, if you need direct access to the driver, assign it to the *java.sql.Driver* class because it creates a generic JDBC driver. It's better to design your system to be as database-independent and portable as possible.

Accessing InterClient extensions to the JDBC

To access InterClient-specific classes and methods (*Driver*, *Connection*, *Statement*, and so forth), you must first cast your JDBC objects before applying the interbase.interclient method. However, you do not need to declare the original objects this way. Always create the object with a generic JDBC class, and cast the object to a class of type *interbase.interclient.Driver*, i*nterbase.interclient.Connection*, *interbase.interclient.Statement*, and so forth.

The following code fragment shows how to cast the JDBC driver object, *icDriver* in order to access a hypothetical InterClient-specific driver method *isBuzzwordCompliant()*:

```
//create the InterClient driver object as a generic JDBC driver
java.sql.Driver driver = new interbase.interclient.Driver();

//Cast driver as type interbase.interclient.Driver and call the method
if ((interbase.interclient.Driver)driver.isBuzzwordCompliant())
System.out.println("It's Buzzword compliant too!");
```

Suppose you've used the *DriverManager* to get the connection, but you want to access an InterClient-specific *Connection* method, called "foobar". Here's an example of casting the connection object (instead of the driver object) to i*nterbase.interclient.Connection*:

```
//create the InterClient driver object as a generic JDBC driver
java.sql.Driver driver = new interbase.interclient.Driver();

//Create the connection object as a generic JDBC connection
java.sql.Connection connection =
java.sql.DriverManager.getConnection(url, properties);

//Call foobar by casting connection to type
interbase.interclient.Connection
(interbase.interclient.Connection)connection.foobar();
```

Tip By using explicit casts whenever you need to access InterClient-specific extensions, you can find these InterClient-specific operations easily if you ever need to port your program to another driver.

Opening a database connection

After loading the driver, or explicitly creating one, you can open a connection to the database. There are two ways to do this: with the *DriverManager*'s *getConnection()* method or the *driver* object's *connect()* method.

Using the DriverManager to get a connection

When you want to access a database, you can get a *java.sql.Connection* object from the JDBC management layer's *java.sql.DriverManager.getConnection()* method. The *getConnection()* method takes a URL string and a *java.util.Properties* object as arguments. For each connection request, the *DriverManager* uses the URL to locate a driver that can connect to the database represented by the URL. If the connection is successful, a *java.sql.Connection* object is returned. The following example shows the syntax for establishing a database connection:

```
java.sql.Connection connection = java.sql.DriverManager.getConnection
      (url,properties);
```

The *Connection* object in turn provides access to all of the InterClient classes and methods that allow you to execute SQL statements and get back the results.

Using InterClient driver object to get a connection

If you are using the driver object to get a connection, use the *connect()* method. This method does the same thing and takes the same arguments as *getConnection()*.

For example:

```
//Create the InterClient driver object explicitly
java.sql.Driver driver = new interbase.interclient.Driver();

//Open a database connection using the driver's connect method of the
java.sql.Connection connection = driver.connect(url, properties);
```

Choosing between the Driver and DriverManager methods

Suppose that you have created an explicit driver object. Even though you could use the driver's *connect()* method, you should always use the generic JDBC methods and classes unless there is some specific reason not to, such as the ones discussed previously. For example, suppose you declared an explicit driver object so you could get driver version numbers, but now you need to create a connection to the database. You should still use the *DriverManager.getConnection()* method to create a connection object instead of the *driver.connect()* method.

Note This is not the case when you are using the InterClient Monitor extension to trace a connection. See **"Debugging your application" on page 61** for a detailed explanation.

Defining connection parameters

The database URL and connection properties arguments to *connect()* or *getConnection()* must be defined before trying to create the connection.

SYNTAX FOR SPECIFYING DATABASE URLS

InterClient follows the JDBC standard for specifying databases using URLs. The JDBC URL standard provides a framework so that different drivers can use different naming systems that are appropriate for their own needs. Each driver only needs to understand its own URL naming syntax; it can reject any other URLs that it encounters. A JDBC URL is structured as follows:

```
jdbc:subprotocol:subname
```

The subprotocol names a particular kind of database connection, which is in turn supported by one or more database drivers. The *DriverManager* decides which driver to use based on which subprotocol is registered for each driver. The contents and syntax of subname in turn depend upon the subprotocol. If the network address is included in the subname, the naming convention for the subname is:

```
//hostname:/subsubname
```

subsubname can have any arbitrary syntax.

DEFINING AN INTERCLIENT URL

InterClient URLs have the following format:

```
jdbc:interbase://server/full_db_path
```

"interbase" is the subprotocol, and server is the hostname of the InterBase server. *full_db_path* (that is, "subsubname") is the full pathname of a database file, including the server's root (/) directory. If the InterBase server is a Windows NT system, you must include the drive name as well. InterClient doesn't support passing any attributes in the URL. For local connections, use:

```
server = "localhost"
```

Note The "/" between the server and *full_db_path* is treated as a delimiter only. When specifying the path for a Unix-based database, you must include the initial "/" for the root directory in addition to the "/" for the delimiter.

In a Unix-based database, the following URL refers to the database **orders.gdb** in the directory **/dbs** on the Unix server *accounts*.

```
dbURL = "jdbc:interbase://accounts//dbs/orders.gdb"
```

In a Windows95, Windows 98, or NT server, the following URL refers to the database **customer.gdb** in the directory **/dbs** on drive C of the server *support*.

```
dbURL = "jdbc:interbase://support/C:/dbs/customer.gdb"
```

DEFINING THE CONNECTION PROPERTIES

Connection properties must also be defined before trying to open a database connection. To do this, pass in a *java.util.Properties* object, which maps between tag strings and value strings. Two typical properties are "user" and "password." First, create the *Properties* object:

```
java.util.Properties properties = new java.util.Properties();
```

Now create the connection arguments. user and password are either literal strings or string variables. They must be the username and password on the InterBase database to which you are connecting:

```
properties.put (“user”, "sysdba");
properties.put (“password”, "masterkey");
```

Now create the connection with the URL and connection properties parameters:

```
java.sql.Connection connection =
   java.sql.DriverManager.getConnection(url, properties);
```

Security

Client applications use standard database user name and password verification to access an InterBase database. InterClient encrypts the user name and password for transmission over the network.

Executing SQL statements

After creating a *Connection* object, you can use it to obtain a *Statement* object that encapsulates and executes SQL statements and returns a result set.

Classes for executing SQL statements

There are three java.sql classes for executing SQL statements:

- *Statement*
- *PreparedStatement*
- *CallableStatement*

THE STATEMENT CLASS

The *java.sql.Statement* interface allows you to execute a static SQL statement and to retrieve the results produced by the query. You cannot change any values with a static statement. For example, the following SQL statement displays information once for specific employees:

```
SELECT first_name, last_name, dept_name
FROM emp_table
WHERE dept_name = 'pubs';
```

The *Statement* class has two subtypes: *PreparedStatement* and *CallableStatement*.

- *PreparedStatement*

  The *PreparedStatement* object allows you to execute a set of SQL statements more than once. Instead of creating and parsing a new statement each time to do the same function, you can use the *PreparedStatement* class to execute pre-compiled SQL statements multiple times. This class has a series of "setXXX" methods that allow your code to pass parameters to a predefined SQL statement; it's like a template to which you supply the parameters. Once you have defined parameter values for a statement, they remain to be used in subsequent executions until you clear them with a call to the *PreparedStatement.clearParameters* method.

  For example, suppose you want to be able to print a list of all new employees hired on any given day. The operator types in the date, which is then passed in to the *PreparedStatement* object. Only those employees or rows in "emp_table" where "hire_date" matches the input date are returned in the result set.

  ```
  SELECT first_name, last_name,
  emp_no FROM emp_table WHERE hire_date = ?;
  ```

  See **"Selecting data with PreparedStatement" on page 55** for more on how this construct works.

- *CallableStatement*

  The *CallableStatement* class is used for executing stored procedures with OUT parameters. Since InterBase does not support the use of OUT parameters, there's no need to use *CallableStatement* with InterClient.

  Note You can still use a *CallableStatement* object if you do not use the OUT parameter methods.

CREATING A STATEMENT OBJECT

Creating a *Statement* object allows you to execute a SQL query, assuming that you have already created the connection object. The example below shows how to use the *createStatement* method to create a *Statement* object:

```
java.sql.Statement statement = connection.createStatement();
```

Querying data

After creating a *Connection* and a *Statement* or *PreparedStatement* object, you can use *executeQuery* method to query the database with SQL SELECT statements.

SELECTING DATA WITH THE STATEMENT CLASS

The *executeQuery* method returns a single result set. The argument is a string parameter that is typically a static SQL statement. The *ResultSet* object provides a set of "get" methods that let you access the columns of the current row. For example, *ResultSet.next* lets you move to the next row of the *ResultSet*, and the *getString* method retrieves a string.

This example shows the sequence for executing SELECT statements, assuming that you have defined the *getConnection* arguments:

```
//Create a Connection object:
java.sql.Connection connection =
java.sql.DriverManager.getConnection(url,properties);

//Create a Statement object
java.sql.Statement statement = connection.createStatement();

//Execute a SELECT statement and store results in resultSet:
java.sql.ResultSet resultSet = statement.executeQuery
("SELECT first_name, last_name, emp_no
FROM emp_table WHERE dept_name = 'pubs'");
//Step through the result rows
System.out.println("Got results:");
while (resultSet.next ()){

//get the values for the current row
String fname = resultSet.getString(1);
String lname = resultSet.getString(2);
String empno = resultSet.getString(3);

//print a list of all employees in the pubs dept
System.out.print(" first name=" + fname);
System.out.print(" last name=" + lname);
System.out.print(" employee number=" + empno);
System.out.print("\n");
}
```

SELECTING DATA WITH PREPAREDSTATEMENT

The following example shows how to use *PreparedStatement* to execute a query:

```
//Define a PreparedStatement object type
java.sql.PreparedStatement preparedStatement;

//Create the PreparedStatement object
preparedStatement = connection.prepareStatement("SELECT first_name,
last_name, emp_no FROM emp_table WHERE hire_date = ?");
//Input yr, month, day
```

```
java.sql.String yr;
java.sql.String month;
java.sql.String day;
System.in.readln("Enter the year: " + yr);
System.in.readln("Enter the month: " + month);
System.in.readln("Enter the day: " + day);

//Create a date object
java.sql.Date date = new java.sql.Date(yr,month,day);

//Pass in the date to preparedStatement's ? parameter
preparedStatement.setDate(1,date);

//execute the query. Returns records for all employees hired on date
resultSet = preparedStatement.executeQuery();
```

Finalizing objects

Applications and applets should explicitly close the various JDBC objects (*Connection*, *Statement*, and *ResultSet*) when they are done with them. The Java "garbage collector" may periodically close connections, but there's no guarantee when, where, or even if this will happen. It's better to immediately release a connection's database and JDBC resources rather than waiting for the garbage collector to release them automatically. The following *close* statements should appear at the end of the previous *executeQuery()* example.

```
resultSet.close();
statement.close();
connection.close();
```

Modifying data

The *executeUpdate()* method of the *Statement* or *PreparedStatement* class can be used for any type of database modification. This method takes a string parameter (an SQL INSERT, UPDATE, or DELETE statement), and returns a count of the number of rows that were updated.

INSERTING DATA

An *executeUpdate* statement with an INSERT statement string parameter adds one or more rows to a table. It returns either the row count or 0 for SQL statements that return nothing:

```
int rowCount= statement.executeUpdate
("INSERT INTO table_name VALUES (val1, val2,…)";
```

If you do not know the default order of the columns, the syntax is:

```
int rowCount= statement.executeUpdate
("INSERT INTO table_name (col1, col2,…) VALUES (val1, val2,…)";
```

The following example adds a single employee to "emp_table":

```
//Create a connection object
java.sql.Connection connection =
java.sql.DriverManager.getConnection(url, properties);

//Create a statement object
java.sql.Statement statement = connection.createStatement();

//input the employee data
Java.lang.String fname;
Java.lang.String lname;
Java.lang.String empno;
System.in.readln("Enter first name: ", + fname);
System.in.readln("Enter last name: ", + lname);
System.in.readln("Enter employee number: ", + empno);

//insert the new employee into the table
int rowCount = statement.executeUpdate
("INSERT INTO emp_table (first_name, last_name, emp_no)
VALUES (fname, lname, empno)");
```

UPDATING DATA WITH THE STATEMENT CLASS

The *executeUpdate* statement with a SQL UPDATE string parameter enables you to modify existing rows based on a condition using the following syntax:

```
int rowCount= statement.executeUpdate(
"UPDATE table_name SET col1 = val1, col2 = val2,
WHERE condition");
```

For example, suppose an employee, Sara Jones, gets married wants you to change her last name in the "last_name" column of the EMPLOYEE table:

```
//Create a connection object
java.sql.Connection connection =
java.sql.DriverManager.getConnection(dbURL,properties);

//Create a statement object
java.sql.Statement statement = connection.createStatement();

//insert the new last name into the table
int rowCount = statement.executeUpdate
("UPDATE emp_table SET last_name = 'Zabrinski'
WHERE emp_no = 13314");
```

UPDATING DATA WITH PREPAREDSTATEMENT

The following code fragment shows an example of how to use *PreparedStatement* if you want to execute the update more than once:

```
//Define a PreparedStatement object type
java.sql.PreparedStatement preparedStatement;

//Create the Prepared_Statement object
preparedStatement = connection.prepareStatement(
"UPDATE emp_table SET last_name = ? WHERE emp_no = ?");
//input the last name and employee number
String lname;
String empno;
System.in.readln("Enter last name: ", + lname);
System.in.readln("Enter employee number: ", + empno);
int empNumber = Integer.parseInt(empno);

//pass in the last name and employee id to preparedStatement's ?
//parameters

//where '1' is the 1st parameter, '2' is the 2nd, etc.
preparedStatement.setString(1,lname);
preparedStatement.setInt(2,empNumber);

//now update the table
int rowCount = preparedStatement.executeUpdate();
```

DELETING DATA

The *executeUpdate()* statement with a SQL DELETE string parameter deletes an existing row using the following syntax:

```
DELETE FROM table_name WHERE condition;
```

The following example deletes the entire "Sara Zabrinski" row from the EMPLOYEE table:

```
int rowCount = statement.executeUpdate
("DELETE FROM emp_table WHERE emp_no = 13314");
```

Executing stored procedures

A stored procedure is a self-contained set of extended SQL statements that are stored in a database as part of its metadata. Stored procedures can pass parameters to and receive return values from applications. From the application, you can invoke a stored procedure directly to perform a task, or you can substitute the stored procedure for a table or view in a SELECT statement. There are two types of stored procedures:

- Select procedures are used in place of a table or view in a SELECT statement. A selectable procedure generally has no IN parameters. See note below.
- Executable procedures can be called directly from an application with the EXECUTE PROCEDURE statement; they may or may not return values to the calling program.

Use the *Statement* class to call select or executable procedures that have no SQL input (IN) parameters. Use the *PreparedStatement* class to call select or executable stored procedures that have IN parameters.

Note Although it is not commonly done, it is possible to use IN parameters in a SELECT statement. For example:

```
create procedure with_in_params(in_var integer)
returns (out_data varchar(10))
as
begin
  for select a_field1 from a_table
  where a_field2 = :in_var
  into :out_data
  do suspend;
end
```

To return one row:

```
execute procedure with_in_params(1)
```

To return more than one row:

```
select * from with_in_params(1)
```

Statement example

An InterClient application can call a select procedure in place of a table or view inside a SELECT statement. For example, the stored procedure *multiplyby10* multiplies all the rows in the NUMBERS table (visible only to the stored procedure) by 10, and returns the values in the result set. The following example uses the *Statement.executeQuery()* method to call the *multiplyby10* stored procedure, assuming that you have already created the *Connection* and *Statement* objects:

```
//multiplyby10 multiplies the values in the resultOne, resultTwo,
//resultThree columns of each row of the NUMBERS table by 10
//create a string object
String sql= new String ("SELECT resultone, resulttwo, resultthree FROM
multiplyby10");

//Execute a SELECT statement and store results in resultSet:
java.sql.ResultSet resultSet = statement.executeQuery(sql);

//Step through the result rows
System.out.println("Got results:");
while (resultSet.next ()){

//get the values for the current row
int result1 = resultSet.getInt(1);
int result2 = resultSet.getInt(2);
int result3 = resultSet.getInt(3);

//print the values
System.out.print(" result one =" + result1);
System.out.print(" result two =" + result2);
System.out.print(" result three =" + result3);
System.out.print("\n");
}
```

PreparedStatement example

In the example below, the *multiply* stored procedure is not selectable. Therefore, you have to call the procedure with the *PreparedStatement* class. The procedure arguments are the scale factor and the value of KEYCOL that uniquely identifies the row to be multiplied in the NUMBERS table.

```
//Define a PreparedStatement object type
java.sql.PreparedStatement preparedStatement;

//Create a new string object
java.sql.String sql = new String ("EXECUTE PROCEDURE multiply 10, 1");

//Create the PreparedStatement object
preparedStatement = connection.prepareStatement(sql);

//execute the stored procedure with preparedStatement
java.sql.ResultSet resultSet = preparedStatement.executeQuery(sql);

//step through the result set and print out as in Statement example
```

Troubleshooting InterClient programs

This section covers troubleshooting InterClient installation and debugging applications.

Handling installation problems

Call *interbase.interclient.InstallTest* to test an InterClient installation. *InstallTest* provides two static methods for testing the installation. A static *main* is provided as a command line test that prints to *System.out. main()* uses the other public static methods that test specific features of the installation. These methods can be used by a GUI application as they return strings, rather than writing the diagnostics to *System.out* as *main()* does. *InstallTest* allows you to:

- determine InterClient driver version information
- determine installed packages
- check basic network configuration
- test making a connection directly without the *DriverManager* with *driver.connect()*
- test making a connection with *DriverManager.getConnection()*
- get SQL Exception error messages

Debugging your application

You can tailor your own driver instances by using a class called *interbase.interclient.Monitor*. This is a public InterClient extension to JDBC. The *Monitor* class contains user-configurable switches that enable you to call a method and trace what happens on a per-driver basis. Types of switches that you can enable include: *enableDriverTrace*, *enableConnectionTrace*, *enableStatementTrace*, and so forth.

Every driver instance has one and only one monitor instance associated with it. The initial monitor for the default driver instance that is implicitly registered with the *DriverManager* has no logging/tracing enabled. Enabling tracing for the default driver is not recommended. However, if you create your own driver instance, you can tailor the tracing and logging for your driver without affecting the default driver registered with the *DriverManager*.

Note If you want to use the *Monitor* to trace connections and statements, you must create the original objects using the *connect()* method of the tailored driver. You cannot create a connection with *DriverManager.getConnection()* method and then try to trace that connection. Since tracing is disabled for the default driver, there will be no data.

The following example shows calls to *getMonitor()* trace methods:

```
//Open the driver manager's log stream
DriverManager.setLogStream(System.out);
//Create the driver object
java.sql.Driver icDriver = new interbase.interclient.Driver();

//Trace method invocations by printing messages to this monitor's
//trace stream
((interbase.interclient.Driver)icDriver).getMonitor().setTraceStream
   (System.out);
((interbase.interclient.Driver)icDriver).getMonitor().enableAllTraces (true);
```

After running the program and executing some SQL statements, you can print out the trace messages associated with the driver, connection, and statement methods. The tracing output distinguishes between implicit calls, such as the garbage collector or InterClient driver calling *close()* versus user-explicit calls. This can be used to test application code, since it would show if result sets or statements aren't being cleaned up when they should.

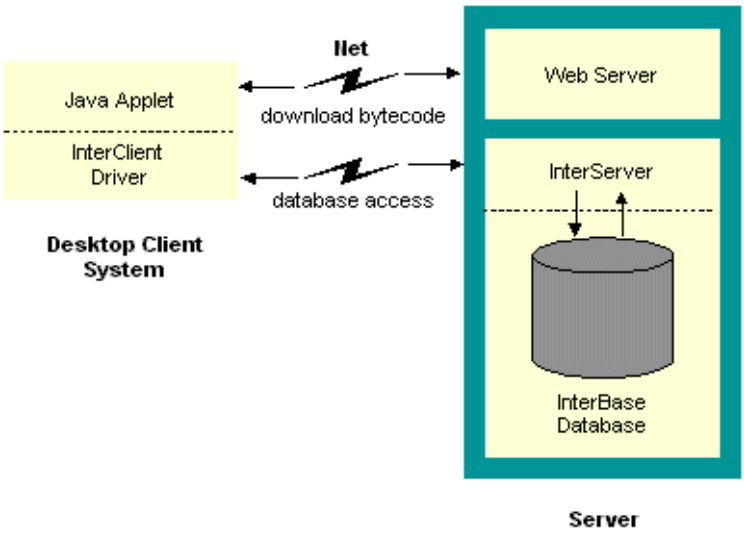
Deploying InterClient programs

Once you have developed your InterClient programs, there are two ways to deploy them: as Java applets embedded on an HTML page, or as stand-alone all-Java applications running on a client system.

Deploying InterClient programs as applets

InterClient programs can be implemented as Java applets that are downloaded over the Internet as part of an HTML web page.

FIGURE 4.4 Using applets to access InterBase



An InterClient applet uses JDBC to provide access to a remote InterBase server in the following manner:

1. A user accesses the HTML page on which the InterClient applet resides.
2. The applet bytecode is downloaded to the client machine from the Web server.
3. The applet code executes on the client machine, downloading the InterClient package (that is, the InterClient classes and the InterClient driver) from the Web server.
4. The InterClient driver communicates with the InterServer process, which in turn establishes a connection to the InterBase server.
5. The InterBase server executes SQL statements and returns the results to the InterServer, which then passes on the results to the user running the InterClient applet.
6. When the applet is finished executing, the applet itself and the InterClient driver and classes disappear from the client machine.

Note In order to use the applet deployment method, the InterServer process and the InterBase server process must be running on the same system as the Web server. Because an applet can communicate only with the server that it was downloaded from, you cannot use an applet to access data from more than one machine/server.

Required software for applets

In order to run InterClient applets, the client and server machines must have the following software loaded:

Client side	Server side
• Java-enabled browser	• InterServer process
• JDBC Driver Manager , which is part of the Java Developer's Kit (JDK)	• InterBase server process
	• Web server process
	• program applets
	• InterClient classes

Pros and cons of applet deployment

The following table displays some of the pros and cons of applet deployment.

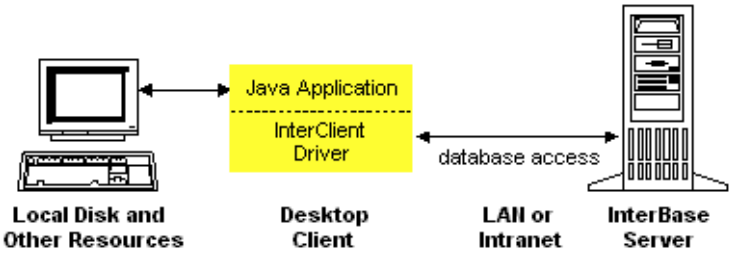
Pros	Cons
• The applet is platform-independent; the program is available to everyone	• An applet cannot open network connections to arbitrary hosts; it can only communicate with the server from which it was deployed (the Web server). Therefore, you could not use an applet if your program needs to access data from more than one server
• All code resides on the server, so if code changes, it needs to be updated only in one place	• Applets cannot access local files, so you could not, for example, use applet code to read or write from your local file system
	• Response time for database applets on the Internet will be slower than for database applications on a LAN

TABLE 4.1 Pros and cons of applet development

Deploying InterClient programs as applications

InterClient programs can also be deployed as stand-alone Java applications. These applications both reside on and execute from the client machine; they're not downloaded from a server. The most common use for these types of Java applications is within a company or corporate Intranet, where the application can access corporate database servers on a local or wide area network. However, you can also use Java applications to access databases via the Internet.

FIGURE 4.5 Using standalone Java applications to access InterBase



Note If your program needs to access data from more than one server/machine, you must develop a stand-alone InterClient application, since you cannot use an applet to do this.

Required software for applications

In order to run InterClient applications, the client and server machines must have the following software loaded:

Client side	Server side
• Java programs (compiled bytecode)	• InterServer process
• InterClient package, including the driver and all of the classes	• InterBase server process
• JDBC Driver Manager , which is part of the Java Developer's Kit (JDK)	

InterClient/JDBC compliance specifications

The following section includes information on:

- **InterClient extensions to the JDBC API**
- **JDBC features not implemented in InterClient**
- **InterClient implementation of JDBC features**
- **InterBase features not available through InterClient or JDBC**
- **Java SQL datatype support**
- **SQL-to-Java type conversions**
- **Java-to-SQL type conversion**
- **InterClient class references**

InterClient extensions to the JDBC API

The following table lists the extensions provided by InterClient that are not part of the JDBC API:

InterClient Subclass	Feature	Description
ErrorCodes		A class defining all error codes returned by InterClient in SQLWarnings and SQLExceptions
PreparedStatement	getParameterMetaData()	Returns a ParameterMetaData object, which provides information about the parameters to a PreparedStatement.
ParameterMetaData		A ParameterMetaData object provides information about the parameters to a PreparedStatement
ResultSet	isNull()	Returns a Boolean value indicating whether the column contains a NULL value. Unlike wasNull(), isNull() does not require the application to read the value first.
	SQL Escape processing: Outer join syntax	InterClient allows you to associate a label with a table name.

TABLE 4.2 InterClient extensions to JDBC

JDBC features not implemented in InterClient

Although all JDBC classes and methods must be implemented in order to create a JDBC-compliant driver, some features are not actually supported.

Note Unsupported features throw an SQLException error message.

The following table lists the JDBC classes, methods, and features not supported by this version of InterClient.

java.sql Subclass	Feature	Description
CallableStatement	OUT parameters	InterBase does not support OUT parameters in stored procedures.
	Escape processing for stored procedures: {? = call procedure_name[]}	InterClient does not support escape syntax with a result parameter.
Statement, PreparedStatement, CallableStatement	Escape processing: Scalar functions	InterClient does not support. Keywords fn user(), fn now(), fn curdate() ARE supported. All other scalar functions are NOT supported unless they are user-defined
Statement, PreparedStatement, CallableStatement	Escape processing: time literals: {t 'hh:mm:ss}	time escape clause not supported
	.f… of TIMESTAMP clause	fractional seconds not supported
Connection	getCatalog()	InterBase does not support catalogs.
	TRANSACTION_READ_UNCOMMITTED	Not supported. Borland recommends using the TRANSACTION_SERIALIZABLE transaction isolation level.
	getLoginTimeout() setLoginTimeout()	Login timeouts are not supported in this release.

TABLE 4.3 Unsupported JDBC features

java.sql Subclass	Feature	Description
	setQueryTimeout() setQueryTimeout() cancel()	Asynchronous cancels are not supported in this release.
Types	BIT TINYINT BIGINT	InterBase does not support these datatypes.
DatabaseMetaData	getCatalogs() getCatalogSeparator() getCatalogTerm() getMaxCatalogNameLength() getMaxSchemaNameLength() getSchemas() getSchemaTerm() isCatalogAtStart()	InterBase does not support catalogs or schemas.
PreparedStatement	setUnicodeStream()	InterClient does not support UNICODE.
ResultSetMetaData	getCatalogName() getSchemaName()	InterBase does not support catalogs or schemas.

TABLE 4.3 Unsupported JDBC features

InterClient implementation of JDBC features

The following lists unique aspects of InterClient's implementation of the JDBC API.

java.sql Subclass	Feature	Description
Driver	connect()	Requires two Properties values: "user" and "password", specifying the database user login and password.
DriverManager	getConnection()	Requires two Properties values: "user" and "password", specifying the database user login and password.

TABLE 4.4 InterClient implementation of JDBC features

InterBase features not available through InterClient or JDBC

The following table lists InterBase features that are currently unavailable to InterClient developers.

Unsupported InterBase Feature	Description
Arrays	InterClient doesn't support arrays
Events and Triggers	InterClient does not support InterBase events and triggers. A trigger or stored procedure posts an event to signal a database change (i.e., inserts, updates, deletes)
Generators	Used to produce unique values to insert into a column as a primary key. InterClient doesn't allow setting of generator values.
Multiple transactions	InterClient does not allow more than one transaction on a single connection.
BLOB filters	A BLOB (binary large object) is used to store large amounts of data of various types. A BLOB filter is a routine that translates BLOB data from one user-defined subtype to another.
Query plan	InterBase uses a query optimizer to determine the most efficient plan for retrieving data. InterClient doesn't allow you to view a query plan.
International character sets	InterClient doesn't support multiple international character sets. *JDBC does support some.
Transaction locking	InterClient doesn't support transaction options such as: two-lock resolution modes, explicit table-level locks,etc.

TABLE 4.5 InterBase features not supported by InterClient

Java SQL datatype support

The following table lists the supported and unsupported Java SQL datatypes.

Supported Java SQL datatypes	Unsupported Java SQL datatypes
VARCHAR, LONGVARCHAR	BIT
VARBINARY, LONGVARBINARY	TINYINT
NUMERIC	BIGINT
SMALLINT	
INTEGER	
FLOAT	
DOUBLE	
DATE	
TIME	
TIMESTAMP	

TABLE 4.6 Java SQL datatype support

SQL-to-Java type conversions

The following table shows the SQL-to-Java type conversion mapping.

DBC SQL Type	…maps to Java Type	…or maps to Java objects returned/used by get/setObject methods
CHAR	java.lang.String	
VARCHAR	java.lang.String	
LONGVARCHAR	java.lang.String	
NUMERIC	java.lang.Bignum	
DECIMAL	java.lang.Bignum	

TABLE 4.7 SQL to Java type conversions

DBC SQL Type	…maps to Java Type	…or maps to Java objects returned/used by get/setObject methods
SMALLINT	short	java.lang.Integer
INTEGER	int	java.lang.Integer
REAL	float	java.lang.Float
FLOAT	double	java.lang.Double
DOUBLE	double	java.lang.Double
BINARY	byte[]	
VARBINARY	byte[]	
LONGVARBINARY	byte[]	
DATE	java.sql.Date	
TIME	java.sql.Time	
TIMESTAMP	java.sql.Timestamp	

TABLE 4.7 SQL to Java type conversions

Java-to-SQL type conversion

The following table shows the Java-to-SQL type conversion mapping.

Java Type maps to>	getObject/setObject	JDBC SQL Type
java.lang.String		VARCHAR, LONGVARCHAR
java.lang.Bignum		NUMERIC
short	java.lang.Integer	SMALLINT
int	java.lang.Integer	INTEGER
float	java.lang.Float	REAL

TABLE 4.8 Java-to-SQL type conversions

Java Type maps to>	getObject/setObject	JDBC SQL Type
double	java.lang.Double	DOUBLE
byte[]		VARBINARY, LONGVARBINARY
java.sql.Date		DATE
java.sql.Time		TIME
java.sql.Timestamp		TIMESTAMP

TABLE 4.8 Java-to-SQL type conversions

InterClient class references

The reference information for the InterClient classes will be included in the documentation set provided to each client.

CHAPTER

5

Programming Applications with ODBC

This chapter discusses how to program InterBase applications with ODBC, including:

- ODBC and OLE DB
- Programming with the ODBC driver
- Configuring and using ODBC datasources

Overview of ODBC

Microsoft's standard, similar in intent to the BDE, is called Open Database Connectivity (ODBC). One standard API provides a unified interface for applications to access data from any data source for which an ODBC driver is available. The InterBase client for Windows NT and Windows 95 includes a 32-bit client library for developing and executing applications that access data via ODBC. The driver is in the file **iscdrv32.dll**.

Similarly to BDE, you configure a data source using the ODBC Administrator tool. If you need to access InterBase databases from third party products that do not have InterBase drivers, you need to install this ODBC driver. The install program then asks you if you want to configure any ODBC data sources. "Configuring" means providing the complete path to any databases that you know you will need to access from non-InterBase-aware products, along with the name of the ODBC driver for InterBase.

ODBC is the common language of data-driven client software. Some software products make use of databases, but do not yet have specific support for InterBase. In such cases, they issue data queries that conform to a current SQL standard. This guarantees that these requests can be understood by any compliant database. The ODBC driver then translates these generic requests into InterBase-specific code. Other ODBC drivers access other vendors' databases.

Microsoft Office, for example, does not have the technology to access InterBase databases directly, but it can use the ODBC driver that is on the InterBase CDROM.

You do not need to install an ODBC driver if you plan to access your InterBase databases only from InterBase itself or from products such as Delphi, C++Builder, and JBuilder that use either native InterBase programming components or Borland SQL-Links components to query InterBase data.

JDBC, InterClient, and InterServer are covered in **Chapter 4, "Programming Applications with JBuilder."**

Configuring an ODBC driver

To access the ODBC Administrator on Windows machines, display the Control Panel and choose ODBC (in some cases, it appears as "32-Bit ODBC Administrator").

Programming with the ODBC driver

Configuring and using ODBC data sources

Configuring data sources

CHAPTER

6

Working with UDFs and Blob Filters

This chapter describes how to create and use UDFs to perform data manipulation tasks that are not directly supported by InterBase. Topics include:

- Writing and compiling a UDF
- Creating a UDF library
- Declaring a UDF to a database
- Calling a UDF
- Writing a Blob UDF
- A description of each UDF in the InterBase UDF library

About UDFs

Just as InterBase has built-in SQL functions such as MIN(), MAX(), and CAST(), it also supports libraries of user-defined functions (UDFs). User-defined functions (UDFs) are host-language programs for performing customized, often-used tasks in applications. UDFs enable the programmer to modularize an application by separating it into more reusable and manageable units. Possibilities include statistical, string, and date functions. UDFs are extensions to the InterBase server and execute as part of the server process.

InterBase provides a library of UDFs, documented in the **"The InterBase UDF library"** section of this chapter on page 90.

You can access UDFs and BLOB filters through **isql** or a host-language program. You can also access UDFs in stored procedures and trigger bodies.

UDFs can be used in a database application anywhere that a built-in SQL function can be used. This chapter describes how to create UDFs and how to use them in an application.

IMPORTANT UDFs are not supported on NetWare.

UDF overview

Creating a UDF is a three-step process:

1. Write the function in any programming language that can create a shared library. Functions written in Java are not supported.
2. Compile the function and link it to a dynamically-linked library (DLL).
3. Use DECLARE EXTERNAL FUNCTION to declare each individual UDF to each database in which you need to use it.

Writing a function module

To create a user-defined function (UDF), you code the UDF in a host language, then build a shared function library that contains the UDF. You must then use DECLARE EXTERNAL FUNCTION to declare each individual UDF to each database where you need to it. Each UDF needs to be declared to each database only once.

Writing a UDF

In the C language, a UDF is written like any standard function. The UDF can require up to ten input parameters, and can return only a single C data value. A source code module can define one or more functions and can use typedefs defined in the InterBase **ibase.h** header file. You must then include **ibase.h** when you compile.

Specifying parameters

A UDF can accept up to ten parameters corresponding to any InterBase datatype. Array elements cannot be passed as parameters. If a UDF returns a Blob, the number of input parameters is restricted to nine. All parameters are passed to the UDF by reference.

Programming language datatypes specified as parameters must be capable of handling corresponding InterBase datatypes. For example, the C function declaration for FN_ABS() accepts one parameter of type double. The expectation is that when FN_ABS() is called, it will be passed a datatype of DOUBLE PRECISION by InterBase.

UDFs that accept Blob parameters require special data structure for processing. A Blob is passed by reference to a Blob UDF structure. For more information about the Blob UDF structure, see **"Writing a Blob UDF" on page 87**.

Specifying a return value

A UDF can return values that can be translated into any InterBase datatype, including a Blob, but it cannot return arrays of datatypes. For example, the C function declaration for FN_ABS() returns a value of type double, which corresponds to the InterBase DOUBLE PRECISION datatype.

By default, return values are passed by reference. Numeric values can be returned by reference or by value. To return a numeric parameter by value, include the optional BY VALUE keyword after the return value when declaring a UDF to a database.

A UDF that returns a Blob does not actually define a return value. Instead, a pointer to a structure describing the Blob to return must be passed as the last input parameter to the UDF.

Character datatypes

UDFs are written in a host language and therefore take host-language datatypes for both their parameters and their return values. However, when a UDF is declared, InterBase must translate them to SQL datatypes or to a CSTRING type of a specified maximum byte length. CSTRING is used to translate parameters of CHAR and VARCHAR datatypes into a null-terminated C string for processing, and to return a variable-length, null-terminated C string to InterBase for automatic conversion to CHAR or VARCHAR.

When you declare a UDF that returns a C string, CHAR or VARCHAR, you must include the FREE_IT keyword in the declaration in order to free the memory used by the return value.

Calling conventions

The calling convention determines how a function is called and how the parameters are passed. The callee function must match the caller function's calling convention. InterBase uses the *STDCALL* calling convention so all UDFs written must use the same calling convention. To make a function use the *STDCALL* calling convention the *__stdcall* keywords must be added to the function declaration.

Here is an example function that specifies the *STDCALL* calling convention:

```
ISC_TIMESTAMP* __stdcall addmonth(ISC_TIMESTAMP *preTime)
{
  // body of function here
}
```

Thread-safe UDFs

In SuperServer implementations of InterBase, the server runs as a single multi-threaded process. This means that you must take some care in the way you allocate and release memory when coding UDFs and in the way you declare UDFs. This section describes how to write UDFs that handle memory correctly in the new single-process environment.

There are several issues to consider when handling memory in the single-process, multi-thread architecture:

- UDFs must avoid static variables in order to be thread safe. You can use static variables only if you can guarantee that only one user at a time will be accessing UDFs, since users running UDFs concurrently will conflict in their use of the same static memory space. If you do return a pointer to static data, you must *not* use FREE_IT.
- UDFs must allocate memory using *ib_util_malloc* rather than static arrays in order to be thread-safe.
- Memory allocated dynamically is not automatically released, since the process does not end. You must use the FREE_IT keyword when you declare the UDF to the database (DECLARE EXTERNAL FUNCTION).

In the following example for user-defined function FN_LOWER(), the array must be global to avoid going out of context:

Multi-process version (not thread-safe: do not use for SuperServer)

```
char buffer[256];
char *fn_lower(char *ups)
{
. . .
    return (buffer);
}
```

In the following version, the InterBase engine will free the buffer if the UDF is declared using the FREE_IT keyword:

Thread-safe version

Notice that this example uses InterBase's *ib_util_malloc*() function to allocate memory.

```
char *fn_lower(char *ups)
{
  char *buffer =  (char *) ib_util_malloc(256);
  ...
  return (buffer);
}
```

The procedure for allocating and freeing memory for return values in a fashion that is both thread safe and compiler independent is as follows:

1. In the UDF code, use InterBase's *ib_util_malloc*() function to allocate memory for return values. This function is in ***interbase_home*/lib/ib_util.dll** on Windows, ***interbase_home*/lib/ib_util.so** on Solaris, and ***interbase_home*/lib/ib_util.sl** on HP-UX.
2. Use the FREE_IT keyword in the RETURNS clause when declaring a function that returns dynamically allocated objects. For example:

   ```
   DECLARE EXTERNAL FUNCTION lowers VARCHAR(256)
   RETURNS CSTRING(256) FREE_IT
   ENTRY POINT 'fn_lower' MODULE_NAME 'ib_udf.dll'
   ```

   InterBase's FREE_IT keyword allows InterBase users to write thread-safe UDF functions without memory leaks.
3. Memory must be released by the same runtime library that allocated it.

Compiling and linking a function module

After a UDF module is complete, you can compile it in a normal fashion into object or library format. You then declare the UDFs in the resulting object or library module to the database using the DECLARE EXTERNAL FUNCTION statement. Once declared to the database, the library containing all the UDFs is automatically loaded at run time from a shared library or dynamic link library.

- Include **ibase.h** in the source code if you use typedefs defined in the InterBase **ibase.h** header file. All "include" (***.h**) libraries are in the ***interbase_home*/SDK/include** directory.
- Link to **gds32.dll** if you use calls to InterBase library functions.
- Linking and compiling:

  Microsoft Visual C/C++ Link with ***interbase_home*/SDK/lib/ib_util_ms.lib** and include ***interbase_home*/SDK/include/ib_util.h**

  Use the following options when compiling applications with Microsoft C++:

Option	Action
c	Compile without linking (DLLs only)
Zi	Generate complete debugging information
DWIN32	Defines "WIN32"
D_MT	Use a multi-thread, statically-linked library

TABLE 6.1 Microsoft C compiler options

  Borland C++ Link with ***interbase_home*/SDK/lib/ib_util.lib** and include ***interbase_home*/SDK/include/ib_util.h**

  Delphi Use ***interbase_home*/SDK/include/ib_util.pas**.

Examples The following commands use the Microsoft compiler to build a DLL that uses InterBase:

```
cl -c -Zi -DWIN32 -D_MT -LD udf.c
   lib -out:udf.dll -def:funclib.def -machine:i586 -subsystem:console
   link -DLL -out:funclib.dll -DEBUG:full,mapped -DEBUGTYPE:CV
   -machine:i586 -entry:_DllMainCRTStartup@12 -subsystem:console
   -verbose udf.obj udf.exp gds32_ms.lib ib_util_ms.lib crtdll.lib
```

This command builds an InterBase executable using the Microsoft compiler:

```
cl -Zi -DWIN32 -D_MT -MD udftest.c udf.lib gds32_ms.lib
   ib_util_ms.lib crtdll.lib
```

See the makefiles (**makefile.bc** and **makefile.msc** on Wintel, **makefile** on UNIX) in the InterBase **examples** subdirectory for details on how to compile a UDF library.

Examples For examples of how to write thread-safe UDFs, see **interbase_home/examples/UDFib_udf.c**. This file contains the source code for the InterBase UDF library.

Creating a UDF library

UDF libraries are standard shared libraries that are dynamically loaded by the database at runtime. You can create UDF libraries on any platform—except NetWare—that is supported by InterBase. To use the same set of UDFs with databases running on different platforms, create separate libraries on each platform where the databases reside. UDFs run on the server where the database resides.

Note A *library*, in this context, is a shared object that typically has a **.dll** extension on Wintel platforms, a **.so** extension on Solaris, and a **.sl** extension on HP-UX.

The InterBase **examples** directory contains sample makefiles (**makefile.bc** and **makefile.msc** on Wintel, **makefile** on UNIX) that build a UDF function library from **ib_udf.c**.

Modifying a UDF library

To add a UDF to an existing UDF library on a platform:

- Compile the UDF according to the instructions for the platform.
- Include all object files previously included in the library and the newly-created object file in the command line when creating the function library.

Note On some platforms, object files can be added directly to existing libraries. For more information, consult the platform-specific compiler and linker documentation.

To delete a UDF from a library, follow the linker's instructions for removing an object from a library. Deleting a UDF from a library does not eliminate references to it in the database.

Declaring a UDF to a database

Once a UDF has been written and compiled into a library, you must use the DECLARE EXTERNAL FUNCTION statement to declare each function to each database where you want to use it. Each function in a library must be declared separately, but needs to be declared only once to each database.

Declaring a UDF to a database informs the database about its location and properties:

- The UDF name as it will be used in embedded SQL statements
- The number and datatypes of its arguments
- The return datatype
- The name of the function as it exists in the UDF module or library
- The name of the library that contains the UDF

You can use **isql**, IBConsole, or a script to declare your UDFs.

```
DECLARE EXTERNAL FUNCTION name [datatype | CSTRING (int)
    [, datatype | CSTRING (int) ...]]
    RETURNS {datatype [BY VALUE] | CSTRING (int)} [FREE_IT]
    [RETURNS PARAMETER n]
    ENTRY_POINT 'entryname'
    MODULE_NAME 'modulename';
```

Table 6.2 lists the arguments to DECLARE EXTERNAL FUNCTION:

Argument	Description
name	Name of the UDF to use in SQL statements; can be different from the name of the function specified after the ENTRY_POINT keyword
datatype	Datatype of an input or return parameter • All input parameters are passed to a UDF by reference • Return parameters can be passed by value • Cannot be an array element
RETURNS	Specifies the return value of a function
BY VALUE	Specifies that a return value should be passed by value rather than by reference
CSTRING (*int*)	Specifies a UDF that returns a null-terminated string *int* bytes in length
FREE_IT	Frees memory of the return value after the UDF finishes running • Use only if the memory is allocated dynamically in the UDF • See also the UDF chapter in the *Developer's Guide*
RETURNS PARAMETER *n*	Specifies that the function returns the *n*th input parameter; required for returning Blobs

TABLE 6.2 Arguments to DECLARE EXTERNAL FUNCTION

Argument	Description
'*entryname*'	Quoted string specifying the name of the UDF in the source code and as stored in the UDF library
'*modulename*'	Quoted file specification identifying the library that contains the UDF • The library must reside on the server • On any platform, the module can be referenced with no path name if it is in ***interbase_home*/UDF** or ***interbase_home*/intl** • If you do not place the library in ***interbase_home*/UDF** or ***interbase_home*/intl**, you *must* specify its location in InterBase's configuration file using the EXTERNAL_FUNCTION_DIRECTORY parameter • It is not necessary to supply the extension to the module name • See **"UDF library placement"** for more about how the operating system finds the UDF library

TABLE 6.2 Arguments to DECLARE EXTERNAL FUNCTION

Declaring UDFs with FREE_IT

InterBase's FREE_IT keyword allows InterBase users to write thread-safe UDF functions without memory leaks.

Whenever a UDF returns a value by reference to dynamically allocated memory, you must declare it using the FREE_IT keyword in order to free the allocated memory.

Note You *must not* use FREE_IT with UDFs that return a pointer to static data, as in the "multi-process version" example on page 79.

The following code shows how to use this keyword:

```
DECLARE EXTERNAL FUNCTION lowers VARCHAR(256)
    RETURNS CSTRING(256) FREE_IT
    ENTRY POINT 'fn_lower' MODULE_NAME 'ib_udf.dll'
```

UDF library placement

The rules for placing UDF libraries have changed since InterBase version 5. In InterBase 6 and later, InterBase finds a UDF library *only* if one of the following conditions is met:

- The library is in ***interbase_home*/UDF**
- The library in a directory other than ***interbase_home*/UDF** and the complete pathname to the directory, including a drive letter in the case of a Windows server, is listed in the InterBase configuration file.

InterBase finds the functions once you have declared them with DECLARE EXTERNAL FUNCTION. You do not need to specify a path in the declaration.

The InterBase configuration file is called **ibconfig** on Windows machines and **isc_config** on UNIX machines.

To specify a location for UDF libraries in a configuration file, enter a line of the following form for Windows platforms:

```
EXTERNAL_FUNCTION_DIRECTORY "D:\Mylibraries\InterBase"
```

For UNIX, the line does not include a drive letter:

```
EXTERNAL_FUNCTION_DIRECTORY "/usr/local/lib/Mylibraries/InterBase"
```

Note that it is no longer sufficient to include a complete path name for the module in the DECLARE EXTERNAL FUNCTION statement. You *must* list the path in the InterBase configuration file if it is other than ***interbase_home*/UDF**.

IMPORTANT For security reasons, InterBase strongly recommends that you place your compiled libraries in directories that are dedicated to InterBase libraries. Placing InterBase libraries in directories such as **C:\winnt40\system32** or **/usr/lib** permits access to all libraries in those directories and is a serious security hole.

Example The following statement declares the TOPS() UDF to a database:

```
DECLARE EXTERNAL FUNCTION TOPS
   CHAR(256), INTEGER, BLOB
   RETURNS INTEGER BY VALUE
   ENTRY_POINT 'TE1' MODULE_NAME 'TM1.DLL';
```

This example does not need the FREE_IT keyword because only cstrings, CHAR, and VARCHAR return types require memory allocation. The module must be in InterBase's **UDF** directory or in a directory that is named in the configuration file.

Example The following **isql** script declares three UDFs, ABS(), DATEDIFF(), and TRIM(), to the **employee.gdb** database:

```
CONNECT 'employee.gdb';
DECLARE EXTERNAL FUNCTION ABS
    DOUBLE PRECISION
    RETURNS DOUBLE BY VALUE
    ENTRY_POINT 'fn_abs' MODULE_NAME 'ib_udf';

DECLARE EXTERNAL FUNCTION DATEDIFF
    DATE, DATE
    RETURNS INTEGER
    ENTRY_POINT 'fn_datediff' MODULE_NAME 'ib_udf';

DECLARE EXTERNAL FUNCTION TRIM
    SMALLINT, CSTRING(256), SMALLINT
    RETURNS CSTRING(256) FREE_IT
    ENTRY_POINT 'fn_trim' MODULE_NAME 'ib_udf';
COMMIT;
```

Note that no extension is supplied for the module name. This creates a portable module. Windows machines add a **.dll** extension automatically.

Calling a UDF

After a UDF is created and declared to a database, it can be used in SQL statements wherever a built-in function is permitted. To use a UDF, insert its name in an SQL statement at an appropriate location, and enclose its input arguments in parentheses.

For example, the following DELETE statement calls the ABS() UDF as part of a search condition that restricts which rows are deleted:

```
DELETE FROM CITIES
    WHERE ABS (POPULATION - 100000) > 50000;
```

UDFs can also be called in stored procedures and triggers. For more information, see "Working with Stored Procedures" and "Working with Triggers" in the *Data Definition Guide*.

Calling a UDF with SELECT

In SELECT statements, a UDF can be used in a select list to specify data retrieval, or in the WHERE clause search condition.

The following statement uses ABS() to guarantee that a returned column value is positive:

```
SELECT ABS (JOB_GRADE) FROM PROJECTS;
```

The next statement uses DATEDIFF() in a search condition to restrict rows retrieved:

```
SELECT START_DATE FROM PROJECTS
    WHERE DATEDIFF (:today, START_DATE) > 10;
```

Calling a UDF with INSERT

In INSERT statements, a UDF can be used in the comma-delimited list in the VALUES clause.

The following statement uses TRIM() to remove leading blanks from *firstname* and trailing blanks from *lastname* before inserting the values of these host variables into the EMPLOYEE table:

```
INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, EMP_NO, DEPT_NO, SALARY)
    VALUES (TRIM (0, ' ',:firstname), TRIM (1, ' ', :lastname),
        :empno, :deptno, greater(30000, :est_salary));
```

Calling a UDF with UPDATE

In UPDATE statements, a UDF can be used in the SET clause as part of the expression assigning column values. For example, the following statement uses TRIM() to ensure that update values do not contain leading or trailing blanks:

```
UPDATE COUNTRIES
    SET COUNTRY = TRIM (2, ' ', COUNTRY);
```

Calling a UDF with DELETE

In DELETE statements, a UDF can be used in a WHERE clause search condition:

```
DELETE FROM COUNTRIES
    WHERE ABS (POPULATION - 100000) < 50000;
```

Writing a Blob UDF

A Blob UDF differs from other UDFs, because pointers to Blob control structures are passed to the UDF instead of references to actual data. A Blob UDF cannot open or close a Blob, but instead invokes functions to perform Blob access.

Creating a Blob control structure

A Blob control structure is a C struct, declared within a UDF module as a typedef. Programmers must provide the control structure definition, which should be defined as follows:

```
typedef struct blob {
   void (*blob_get_segment) ();
   isc_blob_handle blob_handle;
   long number_segments;
   long max_seglen;
   long total_size;
   void (*blob_put_segment) ();
} *Blob;
```

Field	Description
blob_get_segment	The first field in the Blob struct, *blob_get_segment*, is a pointer to a function that is called to read a segment from a Blob if one is passed to the UDF. The function takes four arguments: a Blob handle, the address of a buffer into which to place Blob a segment of data that is read, the size of that buffer, and the address of variable into to store the size of the segment that is read. If Blob data is not read by the UDF, set *blob_get_segment* to NULL.
blob_handle	• The second field in the Blob struct is required. It is a Blob handle that uniquely identifies a Blob passed to a UDF or returned by it. • The type *isc_blob_handle* is new in InterBase 6
number_segments	For Blob data passed to a UDF, *number_segments* specifies the total number of segments in the Blob. Set this value to NULL if Blob data is not passed to a UDF.

TABLE 6.3 Fields in the Blob struct

Field	Description
max_seglen	For Blob data passed to a UDF, *max_seglen* specifies the size, in bytes, of the largest single segment passed. Set this value to NULL if Blob data is not passed to a UDF.
total_size	For Blob data passed to a UDF, *total_size* specifies the actual size, in bytes, of the Blob as a single unit. Set this value to NULL if Blob data is not passed to a UDF.
blob_put_segment	The last field in the Blob struct, *blob_put_segment*, is a pointer to a function that is called to write a segment to a Blob if one is being returned by the UDF. The function takes three arguments: a Blob handle, the address of a buffer containing the data to write into the Blob, and the size, in bytes, of the data to write. If Blob data is not read by the UDF, set *blob_put_segment* to NULL.

TABLE 6.3 Fields in the Blob struct

Declaring a Blob UDF

A Blob UDF is declared to the database using DECLARE EXTERNAL FUNCTION like any non-Blob UDF. Use the RETURNS PARAMETER *n* statements to specify which input Blob is to be returned. For example, the return the third input Blob, specify RETURNS PARAMETER 3. The following statement declares the Blob UDF, Blob_PLUS_Blob, to a database:

```
DECLARE EXTERNAL FUNCTION Blob_PLUS_Blob
    Blob,
    Blob,
    Blob
    RETURNS PARAMETER 3
    ENTRY_POINT 'blob_concatenate' MODULE_NAME 'ib_udf.dll';
COMMIT;
```

A Blob UDF example

The following code creates a UDF, BLOB_CONCATENATE(), that appends data from one Blob to the end of another Blob to return a third Blob consisting of concatenated Blob data. Notice that it is okay to use *malloc*() rather than *ib_util_malloc*() when you free the memory in the same function where you allocate it.

```
/* Blob control structure */
typedef struct blob {
    void (*blob_get_segment) ();
    int *blob_handle;
    long number_segments;
    long max_seglen;
    long total_size;
    void (*blob_put_segment) ();
} *Blob;

extern char *isc_$alloc();
#define MAX(a, b) (a > b) ? a : b
#define DELIMITER "----------------------------"

void blob_concatenate(Blob from1, Blob from2, Blob to)
/* Note Blob to, as final input parameter, is actually for output! */
{
    char *buffer;
    long length, b_length;

    b_length = MAX(from1->max_seglen, from2->max_seglen);
    buffer = malloc(b_length);

    /* write the from1 Blob into the return Blob, to */
    while ((*from1->blob_get_segment) (from1->blob_handle, buffer,
        b_length, &length))
            (*to->blob_put_segment) (to->blob_handle, buffer, length);

    /* now write a delimiter as a dividing line in the blob */
    (*to->blob_put_segment) (to->blob_handle, DELIMITER,
        sizeof(DELIMITER) - 1);

    /* finally write the from2 Blob into the return Blob, to */
    while ((*from2->blob_get_segment) (from2->blob_handle, buffer,
        b_length, &length))
```

```
            (*to->blob_put_segment) (to->blob_handle, buffer, length);

    /* free the memory allocated to the buffer */
    free(buffer);
}
```

The InterBase UDF library

InterBase provides a number of frequently needed functions in the form of a UDF library, named **ib_udf.dll** on Windows platforms and **ib_udf** on UNIX platforms. These UDFs are located in ***interbase_home*/lib** and are all implemented using the standard C library. This section describes each UDF and provides its declaration.

There is a script, **ib_udf.sql**, in the ***interbase_home*/examples/udf** directory that declares all of the functions listed below. If you want to declare only a subset of these, copy and edit the script file.

IMPORTANT Several of these UDFs must be called using the new FREE_IT keyword if—and only if—they are written in thread-safe form, using *malloc* to allocate dynamic memory.

Note When trigonometric functions are passed inputs that are out of bounds, they return zero rather than NaN.

Below is a list of the functions supplied in the InterBase UDF library. See the UDF chapter of the *Language Reference* for more details about these functions.

Function name	Description	Inputs	Outputs
ABS( )	Absolute value	Double precision	Double precision
ACOS( )	Arc cosine	Double precision	Double precision
ASCII_CHAR( )	Return character based on ASCII code	Integer	Char(1)
ASCII_VAL( )	Return ASCII code for given character	Char(1)	Integer
ASIN( )	Arc sine	Double precision	Double precision
ATAN( )	Arc tangent	Double precision	Double precision
ATAN2( )	Arc tangent divided by second argument	Double precision, Double precision	Double precision
BIN_AND( )	Bitwise AND operation	Integer	Integer
BIN_OR( )	Bitwise OR operation	Integer	Integer
BIN_XOR( )	Bitwise XOR operation	Integer	Integer

Function name	Description	Inputs	Outputs
CEILING()	Round up to nearest whole value	Double precision	Double precision
COS( )	Cosine	Double precision	Double precision
COSH( )	Hyperbolic cosine	Double precision	Double precision
COT( )	Cotangent	Double precision	Double precision
DIV( )	Integer division	Integer	Integer
FLOOR( )	Round down to nearest whole value	Double precision	Double precision
LN( )	Natural logarithm	Double precision	Double precision
LOG( )	Logarithm of the first argument, by the base of the second argument	Double precision, Double precision	Double precision
LOG10( )	Logarithm base 10	Double precision	Double precision
LOWER( )	Reduce all upper-case characters to lower-case	Cstring(80)	Cstring(80)
LTRIM( )	Strip preceding blanks	Cstring(80)	Cstring(80)
MOD( )	Modulus operation between the two arguments	Integer, Integer	Integer
PI( )	Return the value of π	—	Double precision
RAND( )	Return a random value	—	Double precision
RTRIM( )	Strip trailing blanks	Cstring(80)	Cstring(80)
SIGN( )	Return -1, 0, or 1	Double precision	Integer
SIN( )	Sine	Double precision	Double precision
SINH( )	Hyperbolic sine	Double precision	Double precision
SQRT( )	Square root	Double precision	Double precision
STRLEN( )	Length of string	Cstring(32767)	Integer
SUBSTR( )	Substring, starting at position equal to second argument, with length equal to third argument	Cstring(80), Smallint, Smallint	Cstring(80)
TAN( )	Tangent	Double precision	Double precision
TANH( )	Hyperbolic tangent	Double precision	Double precision

Declaring Blob filters

You can use BLOB filters to convert data from one BLOB subtype to another. You can access BLOB filters from any program that contains SQL statements.

BLOB filters are user-written utility programs that convert data in columns of BLOB datatype from one InterBase or user-defined subtype to another. Declare the filter to the database with the DECLARE FILTER statement. For example:

```
DECLARE FILTER BLOB_FORMAT
    INPUT_TYPE 1 OUTPUT_TYPE -99
    ENTRY_POINT 'Text_filter' MODULE_NAME 'Filter_99.dll';
```

InterBase invokes BLOB filters in either of the following ways:

- SQL statements in an application
- interactively through **isql**.

isql automatically uses a built-in ASCII BLOB filter for a BLOB defined without a subtype, when asked to display the BLOB. It also automatically filters BLOB data defined with subtypes to text, if the appropriate filters have been defined.

To use BLOB filters, follow these steps:

1. Write the filters and compile them into object code.
2. Create a shared filter library.
3. Make the filter library available to InterBase at run time.
4. Define the filters to the database using DECLARE FILTER.
5. Write an application that requests filtering.

You can use BLOB subtypes and BLOB filters to do a large variety of processing. For example, you can define one BLOB subtype to hold:

- Compressed data and another to hold decompressed data. Then you can write BLOB filters for expanding and compressing BLOB data.
- Generic code and other BLOB subtypes to hold system-specific code. Then you can write BLOB filters that add the necessary system-specific variations to the generic code.
- Word processor input and another to hold word processor output. Then you can write a BLOB filter that invokes the word processor.

For more information about creating and using BLOB filters, see "Working with Blob Data" chapter in the *Embedded SQL Guide*. For the complete syntax of DECLARE FILTER, see the *Language Reference*.

CHAPTER

7

Using the Install and Licensing APIs

This chapter describes how to use the functions in the InterBase Install API as part of an application install. It includes the following topics:

- A description of the Install API and its parts, including a list of the ten functions
- An overview of how to use the API to write an install
- Pseudocode for a typical install
- A reference section with details of each function
- A list of error and warning numbers and their text

About the InterBase Install API

InterBase provides developers with resources that greatly facilitate the process of installing InterBase as part of an application install on the Win32 platform. It provides mechanisms for an install that is completely silent. In addition, it allows you to interact with users if desired, to gather information from them and to report progress and messages back to them.

Using the API functions contained in **ibinstall.dll**, you can integrate the installation of your own product with the deployment of an embedded copy of InterBase. The InterBase portion of the install is *silent*: it does not display billboards and need not require intervention from the end user.

Files in the Install API

The API consists of following files:

File	Description
ibinstall.dll	A library of functions—the "install engine" • An API that contains ten functions plus the full text of all InterBase error messages and warnings • Installed when any InterBase option is installed
ibinstall.h	For C programmers: • A header file that contains function declarations and related values, and a list of error and warning messages and their numbers • Installed with the IBDEV option
ibinstall.lib	For Borland C++ Builder programmers: • A library file that contains the list of functions in **ibinstall.dll** • Installed with the IBDEV option

TABLE 7.1 Install API files required for writing an InterBase install

File	Description
ibinstall_ms.lib	For Microsoft Visual C programmers: • A header file that contains function declarations and related values, and a list of error and warning messages and their numbers • Installed with the IBDEV option
ibinstall.pas	For Delphi programmers: • An Object Pascal sourcefile that contains function declarations and related values • Installed with the IBDEV option

TABLE 7.1 Install API files required for writing an InterBase install

These files are all available on the InterBase CDROM. They are also copied as part of the InterBase install when the DEV option is chosen at install time.

What the Install API does

The functions in the InterBase Install API perform many of the steps that were previously the responsibility of the developer:

- Performs preinstall checks: check for valid operating system, correct user permissions, existing copies of InterBase, disk space, source and destination directories
- Logs all actions to a file called **ib_install.log**
- Creates the destination directory if necessary (and possible)
- Checks for option dependencies
- Copies all files, performing necessary version checks to avoid copying over newer versions
- Creates needed registry entries and increases reference count of shared files
- On NT, installs the InterBase Server and InterBase Guardian as services that start automatically; on Windows 95/98, adds the Guardian to the Run section of the Registry
- Modifies the TCP/IP **Services** file if necessary
- Writes the selected options into the uninstall file

What the Install API does not do

The InterBase Install API does not provide functions for starting the server after it is installed or for creating shortcuts. Licensing functions are handled by the Licensing API

The install handle

Each install instance has a unique handle that identifies it. This handle is a variable of type *OPTION_HANDLE* (see page "Datatypes defined for the Install API" on page 99) that you initialize to zero at the beginning of the InterBase install. Throughout this chapter, this variable is referred to as *handle*, and its address is *phandle*. You are, of course, free to name it what you will. Once you have passed it to *isc_install_set_option*(), it references a data area where all the options for the current install are stored. You need not and should not dereference *handle* directly. The install data is all maintained by the install engine. You need only pass *handle* or a pointer to it—depending on the syntax of the function you are calling—and the install engine does all the work for you.

You must pass *handle* to *isc_install_set_option*() before passing it to any of the other functions, since *isc_install_set_option*() is the only function that accepts *handle* when its value is zero. The others return an error.

Error handling

Each of the functions in the InterBase Install API returns a message number as follows:

- If the function executes successfully, it returns zero (*isc_install_success*).
- If it completes with warnings, it returns a negative number that corresponds to a specific warning message.
- If an error occurs, it returns a positive number that corresponds to a specific error message.

You should check the return each time you call a function. If the return is nonzero, call *isc_install_get_message*() to get the text of the error or warning. For example:

```
error = isc_install_precheck(handle, source_path, dest_path)
    if(error)
    isc_install_get_message(error, message, length(message))
```

The steps in "Overview of the process" do not explicitly remind you to do this. It is assumed that you will do so as necessary.

Callback functions

The *isc_install_execute*() and *isc_uninstall_execute*() functions permit you to pass in pointers to an error-handling function and to a status function, both of which are supplied by you.

- You can use the error-handing function to specify a response to an error or warning and to display message text to the end user.
- The status function can pass status information to the end user and pass back a "cancel" request from the user.

The prototype of these functions must be as follows:

fp_status()

```
int (*fp_status)(int status, void *status_arg, const TEXT* act_desc)
```

fp_status() is a callback function supplied by you, the developer. It accepts an integer, *status*, indicating percent of install/uninstall completed. If you pass a pointer to *fp_status*() to either *isc_install_execute*() or *isc_uninstall_execute*(), they call *fp_status*() at intervals and pass it a number indicating percent completion so that you can display a status bar or other indicator to the end user.

fp_status() also passes back text containing the action being performed, such as "Copy Server Files."

Parameter	Type	Description
status	INT	Accepts an interger from zero to 100 from either *isc_install_execute*() or *isc_uninstall_execute*(). The integer passed in indicates the percent of the install/uninstall completed.
status_arg	VOID*	A pointer to optional user-defined data passed to *isc_install_execute*() or *isc_uninstall_execute*()
act_desc	TEXT*	Provides text that can be displayed as part of the progress indicator

Return Value The *fp_status*() function must return either *isc_install_fp_continue* or *isc_install_fp_abort*.

fp_error()

```
int (*fp_error)(MSG_NO msg_no, void *error_arg, const TEXT* context)
```

fp_error() is a callback function supplied by you, the developer. It accepts an error number, *msg_no*, when a pointer to it is passed to either *isc_install_execute*() or *isc_uninstall_execute*() as a parameter.

Parameter	Type	Description
msg_no	MSG_NO	Accepts an error number from either *isc_install_execute*() or *isc_uninstall_execute*().
error_arg	VOID*	A pointer to optional user-defined data passed to *isc_install_execute*() or *isc_uninstall_execute*()
context	TEXT*	Provides additional information about the nature of the error that can be passed on to the end user

Return Value *fp_error*() processes the error message and returns one of three values: *isc_install_fp_retry*, *isc_install_fp_continue*, or *isc_install_fp_abort*.

fp_error() returns	Effect on calling function
isc_install_ fp_abort	Action fails and calling function returns with the same error
isc_install_ fp_retry	Action is retried but will probably fail again unless user has intervened
isc_install_ fp_continue	Function ignores the error and continues from the point where the error occurred

IMPORTANT These callback functions can make calls only to *isc_install_get_message*(). The result is undetermined if they attempt to call any other Install API function.

Datatypes defined for the Install API

The following datatypes are defined for the Install API functions:

Datatype	Definition
OPTIONS_HANDLE	void*
TEXT	char
MSG_NO	long
OPT	unsigned long
FP_STATUS	function pointer of type int (*fp_status)(int status, void *status_arg, const TEXT* description)
FP_ERROR	function pointer of type int (*fp_error)(MSG_NO msg_no, void *status_arg, const TEXT* description)

TABLE 7.2 Datatypes defined for the InterBase Install API

Writing an InterBase install

"Overview of the process" below lists the steps to follow and issues to consider when writing an InterBase install. The steps you use depend on whether you are writing a silent install or an interactive install. Some steps are merely recommended rather than required, such as Calling *isc_clear_options*() before proceeding with the rest of the install. Others vary depending on whether you are also performing tasks such as writing an uninstall program, creating icons, adding authorization codes, and starting the server.

Following the list of functions there is pseudocode that provides much of the same information in code form.

IMPORTANT There must be only one InterBase server per machine. It is particularly important to avoid putting a SuperServer version of InterBase (V 4.2 and later on Wintel platforms) on a machine where a Classic server is still installed.

Overview of the process

1. The files that you need to develop and compile your application are in the ***ib_install_dir*\SDK** directory if you installed InterBase on your development system with the IB_DEV option. They are also on the InterBase CDROM in the **\SDK** directory. Collect the following files:

   - For C/C++ programmers: **ibinstall.dll**, **ibinstall.lib**, **ibinstall.h**

   - For Delphi programmers: **ibinstall.dll**, **ibinstall.pas**

   Place **ibinstall.dll** in the directory that will contain your executable after it is compiled. Place the other files where your compiler can find them.

2. Declare a variable of type *OPTIONS_HANDLE* for *handle* and initialize it to 0 (a long INT). If you are writing a companion uninstall program, allocate a text buffer for the uninstall file name.

3. If you need messages in a language other than English, call *isc_load_external_text*() to load the error and warning messages.

4. *For interactive installs only* The next steps temporarily select a group of options in order to check that there is a valid operating system, that no Classic server is present, and that there is no InterBase server running. This prevents the case where the end user answers several questions and then finds that the install cannot be performed because of an invalid OS or the presence of the Classic server:

   - Call *isc_install_set_option*() with the following parameters:

   ```
   isc_install_set_option(handle, INTERBASE)
   ```

   If you are installing a client but no server, substitute *IB_CLIENT* for *INTERBASE*.

   - Call *isc_install_precheck*(*handle*, NULL, NULL)

   - Call *isc_install_clear_options*().

5. In an interactive install, query users for a destination and desired options.

6. Call *isc_install_set_option*() once for each option to install. This is the mechanism you use to process user input.

7. Call *isc_install_precheck*() a second time. This time, provide the source and destination path and selected options. *isc_install_precheck*() checks that the destination directory exists and is writable. If the directory does not exist and cannot be created, the function exits with an error. It also checks the dependencies of the selected options and issues a warning if the selections are incompatible or require options not selected. See page 109 for a further description of this function.
8. Call *isc_install_execute*(), passing in *handle*, the source path, and the destination path. If you have written functions to handle errors and display status, you pass in pointers to these functions and optionally pointers to context data as well. The last parameter is an optional pointer to a buffer where the uninstall file name can be stored. If you are providing a companion uninstall program, you must declare a text buffer for the name of the uninstall file and pass in a pointer to it as the final argument for this function. *isc_install_execute*() then performs the actual install.

The next steps are all optional.

9. When the install is complete, you can enable licensed functionality for the product by calling functions in the Licensing API (**iblicense.dll**) and providing certificate IDs and keys. If you do not do this, the end user must enter certificate ID and key pairs (authorization codes) before starting the server.
10. Create shortcuts on the Start menu.
11. Start the InterBase Guardian. You can do this only after providing valid certificate IDs and keys.

A real-world example

The source code for the InterBase **setup.exe** is in ***ib_install_dir*\examples\install**. Since it uses the InterBase Install API, it serves as an example of how to make use of the functions to write an install program.

The Install API functions

The InterBase Install API, **ibinstall.dll**, is a library of functions that facilitate the process of installing and deploying InterBase as part of the developer's own application. The table below lists each entry point in **ibinstall.dll** with a brief description.

Following the table is a list of datatypes that are defined for these functions and a detailed description of each function.

Function	Description
isc_install_clear_options()	Clears all options set by *isc_install_set_option*()
isc_install_execute()	Performs the actual install, including file copying, registry entries, saving uninstall options, and modifying the Services file if necessary
isc_install_get_info()	Returns the requested information in human-readable form: a suggested install directory, required disk space, an option name, or option description
isc_install_get_message()	Returns the text of the requested error or warning message number
isc_install_load_external_text()	Loads the messages from the specified message file
isc_install_precheck()	Performs a number of necessary checks on the install environment, such as checking for existing servers, disk space and access, user permissions, and option dependencies
isc_install_set_option()	Creates a handle to a list of selected install options; must be called once for each option
isc_install_unset_option()	Removes an option from the list of selected options obtained from *isc_install_set_option*()
isc_uninstall_execute()	Removes installed InterBase files (but see exceptions on page 114), updates the registry, removes shares files that have a reference count less than 1, uninstalls the InterBase Guardian and Server services
isc_uninstall_precheck()	Checks for running server, correct user permission, and validity of the uninstall file

TABLE 7.3 Entry points in **ibinstall.dll**

isc_install_clear_options()

Syntax

```
MSG_NO isc_install_clear_options(OPTIONS_HANDLE *phandle)
```

Parameter	Type	Description
phandle	OPTIONS_HANDLE*	• Pointer to the handle of the list of options for the current install • You must initialize this to zero before first use • Handle is maintained by the install engine; you do not need to and should not dereference it

Description *isc_install_clear_options*() clears all the options and other install data stored in *handle* and sets *handle* to zero. It returns a warning if *handle* is zero.

It is good practice to call this function both at the beginning and at the end of an install to free all resources. After calling *isc_install_clear_options*(), you must pass *handle* to *isc_install_set_option*() at least once before passing it to any of the other install functions.

Example [To come]

Return Value Returns *isc_install_success* if the function executes successfully, a number larger than *isc_install_success* if an error occurs, and a number smaller than *isc_install_success* if the function completes with warnings.

Call *isc_install_get_message*() to obtain the error message when the result is not equal to *isc_install_success*.

isc_install_execute()

Syntax

```
MSG_NO isc_install_execute(OPTIONS_HANDLE handle, TEXT *source_path,
   TEXT *dest_path, FP_STATUS *fp_status, void *status_arg,
   FP_ERROR *fp_error, void *error_arg, TEXT *uninst_file_name )
```

Parameter	Type	Description
handle	OPTIONS_HANDLE	• The handle to the list of options created by *isc_install_set_option*() • *isc_install_execute*() returns an error if the value of *handle* is NULL or zero
source_path	TEXT*	• The path where the files to be installed are located, typically located on a CDROM • *isc_install_execute*() returns an error if *source_path* is NULL or an empty string
dest_path	TEXT*	• The path to the desired install location • *isc_install_execute*() returns an error if *dest_path* is NULL or an empty string
fp_status	FP_STATUS*	• A pointer to a callback function that accepts an integer from 0 to 100; see page 97 for more information • May be NULL if no status information is required by the end user
status_arg	void*	• User-defined data to be passed to *fp_status*() • Value is often NULL
fp_error	FP_ERROR*	• A pointer to a callback function that accepts an error number and returns a mnemonic specifying whether *isc_install_execute*() should abort, continue, or retry
error_arg	void*	• User-defined data to be passed to *fp_error*() • Value is often NULL
uninst_file_name	TEXT*	• A pointer to a buffer containing the name of the uninstall file • Can be set to NULL

Description *isc_install_execute*() performs the actual install, including the following operations:

- Calls *isc_install_precheck*() to ensure that the install can be performed; if *isc_install_precheck*() returns an error the install aborts
- Logs all actions to a temporary file called **ib_install.log**
- Creates the destination directory if it does not already exist
- Copies the files using all the correct version checks and delayed copying methods if necessary
- Creates the required registry entries
- Increments UseCount entries in the Registry for shared files
- Installs the Guardian and Server as services on Windows NT, or adds the Guardian to the Run section of the registry on Windows 95/98.
- If necessary, adds *gds_db* to the Services file
- Streams the selected options into **ib_uninst.*nnn*** (where *nnn* is a sequence number) for use at uninstall time
- Frees the options list from memory
- Upon completion, moves **ib_install.log** to the install directory
- Calls *fp_status*() at regular intervals to pass information on the install progress (percent complete)
- Attempts to clean up if at any point the install is canceled by the user or by an error

If you choose to write functions for displaying status and handling errors, you pass in pointers to these functions as the *fp_status* and *fp_error* parameters. In addition, you can pass context information or data to these functions by passing in values for *status_arg* and *error_arg*, although these last two parameters are more commonly NULL. See page 97 for more about these callback functions.

Example [To come]

Return Value Returns zero if the function executes successfully, a positive number if an error occurs, and a negative number if the function completes with warnings.

Call *isc_install_get_message*() to obtain the error message when the result is nonzero.

isc_install_get_info()

Syntax

```
MSG_NO isc_install_get_info(OPT option, int info_type, void *info_buf,
    unsigned int buf_len)
```

Parameter	Type	Description
option	OPT	• The option for which information is requested if *info_type* is 2 through 4 • Returns an error if *option* is not one of thefollowing tokens: INTERBASE, IB_SERVER, IB_CLIENT, IB_CMD_TOOLS, IB_GUI_TOOLS, IB_DOC, IB_EXAMPLES, IB_EXAMPLE_API, IB_EXAMPLE_DB, IB_DEV, IB_CONNECTIVITY_SERVER, IB_ODBC_CLIENT, IB_OLEDB_CLIENT, IB_JDBC_CLIENT, IB_CONNECTIVITY See *isc_install_set_option*() on page 111 for a description of each option.
info_type	int	Specifies the type of information requested; can be any one of the following values
	isc_install_info_destination	• Returns a suggested destination • Ignores any value passed for *option*
	isc_install_info_opspace	• Returns the disk space required to install a particular option • Requires a valid value for *option*
	isc_install_info_opname	• Returns a human-readable option name for the specified option • Requires a valid value for *option*
	isc_install_info_opdescription	• Returns a human-readable description for the specified option • Requires a valid value for *option*

Parameter	Type	Description
info_buf	void*	• *isc_install_get_info*() writes the requested information to this buffer • Returns an error if *info_buf* is NULL • If disk space information is requested, the result is an unsigned long
buf_len	unsigned int	• The length in bytes of *info_buf* • Returns an error if *buf_len* is NULL • Value should be al least ISC_INSTALL_MAX_MESSAGE_LEN bytes • If a destination suggestion is requested, the recommended buffer size is ISC_INSTALL_MAX_PATH

Description *isc_install_get_info*() returns the information requested by *info_type* into *info_buf* location. The *info_buf* and *buf_len* parameters cannot be NULL.

Example [To come]

Return Value Returns zero if the function executes successfully, a positive number if an error occurs, and a negative number if the function completes with warnings.

Call *isc_install_get_message*() to obtain the error message when the result is nonzero.

The contents of *info_buf* are undetermined if *isc_install_get_message*() returns anything other than zero, so the caller should always check the return from this function.

isc_install_get_message()

Syntax

```
MSG_NO isc_install_get message(MSG_NO msg_no, TEXT *msg, int msg_len)
```

Parameter	Type	Description
msg_no	MSG_NO	• Message number for which text is requested • This is the return from all the Install API functions
msg	TEXT*	• A pointer to the buffer in which the message will be returned • The message is always NULL-terminated
msg_len	int	• The length of *msg* in bytes • Value should be al least ISC_INSTALL_MAX_MESSAGE_LEN bytes

Description *isc_install_get_message*() converts the error or warning value stored in *msg_no* and returns the corresponding message text to the programmer.

Example [To come]

Return Value Returns zero if the function executes successfully, a positive number if an error occurs, and a negative number if the function completes with warnings.

Call *isc_install_get_message*() to obtain the error message when the result is nonzero.

isc_install_load_external_text()

Syntax

```
MSG_NO isc_install_load_external_text(TEXT *external_path)
```

Parameter	Type	Description
external_path	TEXT*	A pointer to a buffer that contains the full path and filename of a file containing error and warning messages in a language other than English

Description *isc_install_load_external_text*() loads the message file from the named path. This file contains the text of the install error and warning messages as well as option names and descriptions, and action text, description, and status messages.

If you are using English-language messages, there is no need to call this function. For messages in other languages, you can purchase translations from some InterBase VARs and use this function to load them. You must initialize this buffer with the path and filename to use.

Example [To come]

Return Value Returns zero if the function executes successfully, a positive number if an error occurs, and a negative number if the function completes with warnings.

isc_install_precheck()

Syntax

```
MSG_NO isc_install_precheck(OPTIONS_HANDLE handle, TEXT *source_path,
    TEXT *dest_path)
```

Parameter	Type	Description
handle	OPTIONS_HANDLE	• The handle to the list of options created by *isc_install_set_option*() • *isc_install_precheck*() returns an error if the value of *handle* is NULL or zero
source_path	TEXT*	• The path where the files to be installed are located, typically located on a CDROM • This check is skipped if *source_path* is NULL
dest_path	TEXT*	• The path to the desired install location • Disk space check is skipped if *dest_path* is NULL

Description *isc_install_precheck*() performs the following checks to ensure that installation is possible:

- Checks for a valid operating system. These are currently Windows 95, Windows 98, and Windows NT 4.
- Checks that an InterBase Classic server (version 4.1 or earlier) is not present. The InterBase server (SuperServer) is a multithreaded architecture and cannot coexist with the Classic server.
- Checks that *source_path* exists and is a directory readable by the user. No check is performed if *source_path* is NULL or an empty string.
- Checks that *dest_path* is a directory writable by the user and that the drive contains enough space to install the selected components. No check is performed if *dest_path* is NULL or an empty string.
- If the *IB_SERVER* option is specified, checks whether any existing newer or older version of the SuperServer is already running.
- On NT, if the *IB_SERVER* option is specified, checks that the user performing the install has administrative privileges.

- Checks the dependencies of the options required. The dependencies between the options are as follows:

If any of these are specified	These must also be installed
IB_CMD_TOOLS, *IB_GUI_TOOLS*, *IB_DEV*, and *IB_ODBC_CLIENT*, *IB_OLEDB_CLIENT*, *IB_JDBC_CLIENT*, *IB_CONNECTIVITY*	*IB_CLIENT*
IB_EXAMPLES	*IB_SERVER*, *IB_CLIENT*, and *IB_DEV*
IB_EXAMPLE_API3	*IB_CLIENT* and *IB_DEV*
IB_EXAMPLE_DB	*IB_SERVER*

isc_install_precheck() returns an error if any of the checks besides option dependencies fails. It returns a warning if necessary options have not been specified.

Example [To come]

Return Value Returns *isc_install_success* if the function executes successfully, a number larger than *isc_install_success* if an error occurs, and a number smaller than *isc_install_success* if the function completes with warnings.

Call *isc_install_get_message*() to obtain the error message when the result is not equal to *isc_install_success*.

isc_install_set_option()

Syntax

```
MSG_NO isc_install_set_option(OPTIONS_HANDLE *phandle,
    OPT option)
```

Parameter	Type	Description
phandle	OPTIONS_HANDLE*	• Pointer to the handle of the list of options for the current install • You must initialize this to zero before first use • Handle is maintained by the install engine; you do not need to and should not dereference it
option	OPT	*option* can be any one of the following values:
		INTERBASE — • Installs all interbase components and their related files; same as specifying IB_SERVER, IB_CLIENT, IB_CMD_TOOLS, IB_GUI_TOOLS, IB_DOC, IB_EXAMPLES, and IB_DEV
		IB_SERVER — • Installs the Server component of InterBase: the server, the license file if present, the message file, the Guardian, the server configuration tool, **gstat**, **gds_lock_print**/**iblockpr**, the UDF library, the international character set library, and the help files • Makes all necessary additions to the registry • Creates the InterBase service on NT • On NT, modifies the Services file, if necessary, to add the gds_db service
		IB_CLIENT — • Installs the InterBase Client: the client library, the license file, and the message file • Makes all necessary additions to the registry (Windows only) • On NT, modifies the Services file, if necessary, to add the gds_db service

Parameter	Type	Description	
		IB_CMD_TOOLS	• Installs all the command line tools for InterBase on Windows platforms: **gbak**, **gfix**, **gsec**, **gstat**, **iblockpr**, and **isql** • Issues a warning if the IB_CLIENT option has not been specified
		IB_GUI_TOOLS	• Installs IBConsole and its related help files • Issues a warning if the IB_CLIENT option has not been specified
		IB_DOC	• Installs the InterBase documentation
		IB_EXAMPLES	• Installs all the InterBase examples; has the same effect as specifying IB_EXAMPLE_API and IB_EXAMPLE_DB • Issues a warning if the IB_SERVER, IB_CLIENT, and IB_DEV options have not been specified
		IB_EXAMPLE_API	• Installs all API, SQL, DSQL, and ESQL example files • Issues a warning if the IB_CLIENT and the IB_DEV options have not been specified
		IB_EXAMPLE_DB	• Installs all example databases • Issues a warning if the IB_SERVER option has not been specified
		IB_DEV	• Installs **gpre**, the import libraries, and the header files

Description *isc_install_set_option*() creates and maintains a handle to a list of requested option values. You must call *ib_install_set_option*() once for each option to be installed. In an interactive install, the function would typically be invoked by a mouse click in a check box.

You must initialize *handle* to zero before calling *isc_install_set_option*() for the first time.

Example [To come]

Return Value Returns *isc_install_success* if the function executes successfully, a number larger than *isc_install_success* if an error occurs, and a number smaller than *isc_install_success* if the function completes with warnings.

Call *isc_install_get_message*() to obtain the error message when the result is not equal to *isc_install_success*.

isc_install_unset_option()

Syntax

```
MSG_NO isc_install_unset_option(OPTIONS_HANDLE *phandle, OPT option)
```

Parameter	Type	Description
phandle	OPTIONS_HANDLE	• Pointer to the handle of the list of options for the current install • You must initialize this to zero before first use • Handle is maintained by the install engine; you do not need to and should not dereference it
option	OPT	• *option* can be any of the values listed for *isc_install_set_option*() • If *option* is the only member of the list, sets *handle* to zero

Description *isc_install_unset_option*() removes the option specified by *option* from the list maintained by *handle*. You must call this function once for each option to be removed. If *handle* is zero when this function is called, the function generates a warning.

Example [To come]

Return Value Returns *isc_install_success* if the function executes successfully, a number larger than *isc_install_success* if an error occurs, and a number smaller than *isc_install_success* if the function completes with warnings.

Call *isc_install_get_message*() to obtain the error message when the result is not equal to *isc_install_success*.

isc_uninstall_execute()

Syntax

```
MSG_NO isc_uninstall_execute(TEXT *uninstall_file_name,
   FP_STATUS *fpstatus, void *status_arg, FP_ERROR *fp_error,
   void *error_arg)
```

Parameter	Type	Description
uninstall_file_name	TEXT*	• The name of the file containing the options that were installed • Cannot be NULL
fp_status	FP_STATUS*	• A pointer to a callback function that accepts an integer from 0 to 100; see page 97 for more information • Can be NULL if no status information is required by the end user
status_arg	void*	• User-defined data to be passed to *fp_status*() • Value is often NULL
fp_error	FP_ERROR*	• A pointer to a callback function that accepts an error number and returns a mnemonic specifying whether *isc_install_execute*() should abort, continue, or retry
error_arg	void*	• User-defined data to be passed to *fp_error*() • Value is often NULL

Description *isc_uninstall_execute*() performs the actual uninstall, including the following steps:

- Calls *isc_uninstall_precheck*() to ensure that the uninstall can be performed.
- Decrements UseCount entries in the Registry for shared files and remove any files that have a reference count less than one, except for files that have a value of zero preassigned by Microsoft, such as **msvcrt.dll**.
- Removes all InterBase files named in **ib_uninst.*nnn*** except for **isc4.gdb**, **isc4.gbk**, and **ib_license.dat**.
- Removes all registry entries in **ib_uninst.*nnn***.
- On Windows NT, uninstalls the Guardian and Server services. On Windows 95/98, removes the Run registry entries for them.
- Calls *fp_status*() at regular intervals to keep caller informed of uninstall status.
- Cleans up if uninstall is cancelled by the user if by an error.

Example [To come]

Return Value Returns zero if the function executes successfully, a positive number if an error occurs, and a negative number if the function completes with warnings.

Call *isc_install_get_message*() to obtain the error message when the result is nonzero.

isc_uninstall_precheck()

Syntax

```
MSG_NO isc_uninstall_precheck(TEXT *uninstall_file_name)
```

Parameter	Type	Description
uninstall_file_name	TEXT*	• A pointer to the name of the uninstall file that was created by *isc_install_execute*() • Cannot be NULL

Description *isc_uninstall_precheck*() performs several checks to determine that an uninstall is possible. It checks that:

- The operating system is valid: Windows NT 4, Windows 95/98
- The uninstall file (**ib_uninst.*nnn***) is valid and contains the streamed list of options.
- The server, if installed, is not running.
- The user performing the uninstall is a member of either the administrator or poweruser groups when the platform is Windows NT; no equivalent check is performed on Windows 95/98.

Example [To come]

Return Value Returns zero if the function executes successfully, a positive number if an error occurs, and a negative number if the function completes with warnings.

Call *isc_install_get_message*() to obtain the error message when the result is nonzero.

Call *isc_install_get_message*() to obtain the text of an error message or warning when the result of one of the Install API functions is nonzero.

Using the License API

The InterBase server functionality must be activated by installing *authorization codes* that are provided on software activation certificates obtained from InterBase. Each authorization code consists of a Certificate ID and Certificate key. You can activate the server as part of your install by using functions provided in the InterBase License API. If you do not activate the server as part of the install, it will be inactive until the end user provides authorization codes using IBConsole.

The InterBase License API (**iblicense.dll**) provides five functions that allow you to check, add, remove, and view certificate ID and key pairs (authorization codes). The fifth function retrieves and displays messages associated with the return values from the other four functions.

Loading the License API

You cannot statically load **iblicense.dll** during an install process. Use the Windows *LoadLibrary*() API call or other language-specific equivalent to load it dynamically when you need it and free the library immediately after use.

Typically, you would load the License API at the beginning of an install in order to check that your desired certificate ID/key pairs can indeed be added. Call *isc_license_check*() and then free the library. Later, when you have completed the install portion and are ready to add authorization codes, load **iblicense.dll** again and add the authentication codes. This sequence avoids the case in which an install is completed and then must be uninstalled because the authentication codes cannot be added for some reason.

Preparing the ib_license.dat file

InterBase authorization codes are stored in the **ib_license.dat** file in the InterBase root directory. This file contains authorization codes from previous installs. Authorization codes for previous versions of InterBase do not work with the current version, but you should retain them in case you need them for older versions. If you delete the file, InterBase cannot replace the codes.

There is also an **ib_license.dat** file on the InterBase CD-ROM, which contains the client activation code for the current client version. Following the steps in this section ensures that you are using the most recent client authorizations and that no prior authorization codes are lost:

- Check for the existence of **ib_license.dat** in the InterBase install directory.

- If the file is found, concatenate it with the **ib_license.dat** that is on the CD-ROM to add the current client capability.
- If the file is not found, copy **ib_license.dat** from the CD-ROM to the InterBase install directory.

These steps ensure that you have retained any existing licensed server functionality while providing functionality for the latest client.

The capabilities activated on the server are the union of the capabilities activated by each line.

Adding server functionality

There are five functions available for manipulating authorization codes in **ib_license.dll**:

- *isc_license_add*() adds a line to **ib_license.dat**. Use only authorization codes that you have been given expressly as deployment codes from Inprise Corp.
- *isc_license_check*() checks to see whether an authorization code *could* be added to **ib_license.dat**. This function performs all the same tasks as *isc_license_add*(), without actually modifying **ib_license.dat**.
- *isc_license_remove*() removes a line from **ib_license.dat**.
- *isc_license_display*() displays the authorization codes that are currently in **ib_license.dat**.
- *isc_license_get_msg*() returns the text of error messages that correspond to error codes returned by the other four licensing functions.

isc_license_add()

Syntax `int isc_license_add(char *cert_id, char *cert_key)`

Parameter	Type	Description
cert_id	char*	Pointer to a NULL-terminated character buffer containing the certificate ID to be added
cert_key	char*	Pointer to a NULL-terminated character buffer containing the certificate key to be added

Description Adds a line containing the specified certificate ID and key pair to the **ib_license.dat** file in the InterBase install directory. This ID/key pair must be a valid authorization code obtained from Inprise sales. InterBase might require several authorization codes to run and you must call the function once for ID/key pair you need to add.

Example [To come]

Return Value *isc_license_add*() returns *isc_license_msg_restart* if it successfully adds the authorization code. If it returns an error, pass the return value to *isc_license_get_msg*() to obtain the exact error message. The possible return values are:

Return	Description
isc_license_msg_restart	Authorization code was successfully added
isc_license_msg_writefailed	The authorization code could not be written
isc_license_msg_dupid	The authorization code was not added to the license file because it is a duplicate of one already present in the file
isc_license_msg_convertfailed	The ID/key combination is invalid

TABLE 7.4 Error codes from *isc_license_add*()

isc_license_check()

Syntax

```
int isc_license_check(char *cert_id, char *cert_key)
```

Parameter	Type	Description
cert_id	char*	Pointer to a NULL-terminated character buffer containing the certificate ID to be checked
cert_key	char*	Pointer to a NULL-terminated character buffer containing the certificate key to be checked

Description Checks whether the specified ID/key pair is valid and could be added to **iblicense.dat**. Calling this function does not actually add anything to the file.

Example [To come]

Return Value *isc_license_check*() returns *isc_license_success* if it determines that the authorization code could be added. If it returns an error, pass the return value to *isc_license_get_msg*() to obtain the exact error message. The possible return values are:

Return	Description
isc_license_success	Authorization code could be successfully added
isc_license_msg_dupid	The authorization code was not added to the license file because it is a duplicate of one already present in the file
isc_license_msg_convertfailed	The ID/key combination is invalid

TABLE 7.5 Error codes from *isc_license_check*()

isc_license_remove()

Syntax

```
int isc_license_remove(char *cert_key)
```

Parameter	Type	Description
cert_key	char*	Pointer to a NULL-terminated character buffer containing the certificate key to be added

Description Removes the line specified by *cert_key* from **ib_license.dat**.

Example [To come]

Return Value *isc_license_remove*() has the following return values:

Return	Description
isc_license_msg_restart	Authorization code was successfully removed
isc_license_msg_notremoved	The authorization code could not be removed; possible reasons are: • The key specified by *cert_key* does not exist in **ib_license.dat** • *cert_key* identifies an evaluation license

TABLE 7.6 Returns codes from *isc_license_remove*()

isc_license_display()

Syntax

```
unsigned short isc_license_display(char *buf, unsigned short buf_len)
```

Parameter	Type	Description
buf	*char**	• A character buffer for the result • Must be allocated by the programmer • *isc_license_get_message*() returns an error if *buf* is not long enough • Must be NULL-terminated
buf_len	short	• Length of *buf*

Description Places all certificate ID/key pairs that are currently in **iblicense.dat** into *buf*, separated by commas and NULL-terminated.

Example [To come]

Return Value Returns zero if it succeeds. Otherwise, it returns the length that *buf* must have in order to contain the message text, and *buf* itself contains NULL.

isc_license_get_msg()

Syntax

```
unsigned short isc_get_msg(short msg_no, char *msg,
    unsigned short msg_len)
```

Parameter	Type	Description
msg_no	short	A message number returned by one of the other *isc_license_**() functions
msg	char*	• A character buffer for the message that corresponds to *msg_no* • Must be allocated by the programmer • Recommended length is ISC_LICENSE_MAX_MESSAGE_LEN
msg_len	short	The length of *msg*

Description When passed an error code from one of the other four functions in the License API, *isc_license_get_msg*() returns the text of the corresponding error message in the *msg* buffer.

Example [To come]

Return Value *isc_license_get_msg*() returns zero if it succeeds. Otherwise, it returns the length that *msg* must have in order to contains the message text.

Pseudocode for a typical install

The following code indicates the steps you would typically take in writing an install. Calls to functions in the Install API and related specific code are in bold.

```
begin
   OPTIONS_HANDLE   handle;
   boolean          done = false;
   LANG_TYPE        language;

/* Get user preference if desired. This is if you created translated
 * ibinstall.msg files in different directories */
language = get_language_choice();
if (language <> english)
   isc_install_load_external_text(lang_dirs[language]);

/* Query install for all the possible option names */
while(not all options)
begin
   isc_install_get_info(isc_install_info_opname, option, opname buffer,
     ISC_INSTALL_MAX_MESSAGE_LEN);
   isc_install_get_info(isc_install_info_opdescription, option, opdesc buffer,
     ISC_INSTALL_MAX_MESSAGE_LEN);
   isc_install_get_info(isc_install_info_opspace, option, opspace buffer,
     sizeof(unsigned long));
end;

/* Get a suggested destination directory */
isc_install_get_info(isc_install_info_destination, 0, dest_buffer,
     ISC_INSTALL_MAX_PATH);

/* Present the user his choices and interact with them */
interact_with_user();
```

```
/* Use isc_install_set_option and isc_install_unset_option either when
 * interacting with the user or after the user pushes Install button.
 * Zero the handle the very first time. */
handle = 0L;
while (not all options)
begin
   if(option is selected)
      isc_install_set_option(&handle, option); // Check for errors.
end;

/* You can check for source_dir and dest_dir. In this case no check
 * is performed on directories. Also not all of the checks are performed
 * on the dest_path if it does not exist. */
error = isc_install_precheck(handle, source_path, dest_path);
if (error > isc_install_success) then
   begin
        /* if a classic server is installed, or any server is running
         * then give error and exit */
   isc_install_get_message(error, message, length(message));
        user_choice = display(message);
        do_user_choice();/* For example, terminate, return to options
                         * selection screen */
   end
else if (error < isc_install_success) then
   begin
      /* Some warning has occured, display it and continue */
      isc_install_get_message(error, message, length(message))
      display(message)
   end

display_file(install.txt)
display_file(license.txt)

/* You can supply no callback functions but it is not recommended because install
 * will abort on any error. Some of the errors might be ignored. Some problems
 * might be fixed by hand after the install. If you do not use callbacks you will
 * not be able to appraise the user of the status */
error = isc_install_execute(&handle, source_path, dest_path, NULL, NULL, NULL,
     NULL, NULL)
```

```
if (error < 0) then
    begin
       isc_install_get_message(error, message, length(message))
        display(message)
        exit()
     end
else
    if (error > 0) then
       begin
          isc_install_get_message(error, message, length(message))
          display(message)
       end
display_file(readme.txt)

/* This is mandatory. Not clearing options results in memory leaks */
isc_install_clear_options(&handle)
display_done()
end
```

CHAPTER

8

Introduction to IBX

InterBase Express (IBX) is a set of data access components that provide a means of building applications with Borland Delphi 5 which can access, administer, monitor, and run the InterBase Services on InterBase databases.

This document assumes that you are familiar with the Delphi development environment and know how to use the Standard, Data Access, and Data Control components.

Though many IBX components are similar to the Delphi data access components in name, they do not use the Borland Database Engine (BDE). For each IBX component with a BDE counterpart, the differences are described below.

There is no simple migration from BDE to IBX applications. Generally, you must replace the BDE components with the comparable IBX components, and then recompile your applications. However, the speed you gain, along with the access you get to the powerful InterBase features make migration well worth your time. Migration is discussed in more depth in **Chapter 22, "Migrating to InterBase Express."**

The InterBase tab

The InterBase tab in Delphi 5 contains the IBX equivalents of the Delphi components on the Data Access tab, along with additional monitoring and events components.

FIGURE 8.1 The InterBase tab

The InterBase tab contains the following components, from left to right:

- **TIBTable**
- **TIBQuery**
- **TIBStoredProc**
- **TIBDatabase**
- **TIBTransaction**
- **TIBUpdateSQL**
- **TIBDataSet**
- **TIBSQL**
- **TIBDatabaseInfo**
- **TIBSQLMonitor**
- **TIBEvents**

These components are discussed in the following sections.

TIBTable Component

TIBTable

Use a *TIBTable* component to set up a live dataset on a table or a view without having to enter any SQL statements.

The *TIBTable* component is discussed more fully in **Chapter 13, "Working with Tables."**

TIBQuery Component

TIBQuery

Use a *TIBQuery* component to execute any InterBase dynamic SQL statement, restrict the result set to particular columns and rows, use aggregate functions, and join multiple tables.

The *TIBQuery* component is discussed more fully in **Chapter 14, "Working with Queries."**

TIBStoredProc

TIBStoredProc Component

Use a *TIBStoredProc* component for InterBase executable procedures; procedures that return, at most, one row of information. For stored procedures that return more than one row of data, or for SELECT procedures, use either *TIBQuery* or *TIBDataset* components.

The *TIBStoredProc* component is discussed more fully in **Chapter 15, "Working with Stored Procedures."**

TIBDatabase

TIBDatabase Component

Use a *TIBDatabase* component to establish connections to databases, which can involve one or more concurrent transactions. Unlike the BDE, IBX has a separate transaction component, which allows you to separate transactions and database connections.

The *TIBDatabase* component is discussed more fully in **Chapter 11, "Connecting to Databases."**

TIBTransaction

TIBTransaction Component

Use a *TIBTransaction* component to handle transaction contexts, which might involve one or more database connections. In most cases, a simple one database/one transaction model will do. Having a separate transaction component allows you to take advantage of the InterBase two-phase commit functionality (transactions that span multiple connections) and multiple concurrent transactions using the same connection

The *TIBTransaction* component is discussed more fully in **"Using transactions" on page 150**.

TIBUpdateSQL

TIBUpdateSQL Component

Use a *TIBUpdateSQL* component to update read-only datasets or *TIBQuery* output.

The *TIBUpdateSQL* component is discussed more fully in **"Updating a read-only result set" on page 209**

TIBDataSet

TIBDataSet Component

Use a *TIBDataSet* component to execute any InterBase dynamic SQL statement, restrict the result set to particular columns and rows, use aggregate functions, and join multiple tables. *TIBDataSet* components are similar to *TIBQuery* components, except that they support live datasets without a *TIBUpdateSQL* component.

The *TIBDataSet* component is discussed more fully in **Chapter 14, "Working with Queries."**

TIBSQL

TIBSQL Component

Use a *TIBSQL* component for data operations that need to be fast and lightweight. Operations such as data definition and pumping data from one database to another are suitable for *TIBSQL* components.

The *TIBSQL* component is discussed more fully in **Chapter 14, "Working with Queries."**

TIBDatabaseInfo

TIBDatabaseInfo Component

Use a *TIBDatabaseInfo* component to retrieve information about a particular database, such as the sweep interval, ODS version, and the user names of those currently attached to the database.

The *TIBDatabaseInfo* component is discussed more fully in **"Requesting information about an attachment" on page 162**.

TIBSQLMonitor

TIBSQLMonitor Component

Use a *TIBSQLMonitor* component to develop diagnostic tools to monitor the communication between your application and the InterBase server. With the *TraceFlags* property of a *TIBDatabase* component turned on, active *TIBSQLMonitor* components can keep track of the connection's activity and send the output to a file or control.

The *TIBSQLMonitor* component is discussed more fully in **Chapter 17, "Debugging with SQL Monitor."**

TIBEvents

TIBEvents Component

Use a TIBEvents component to register interest in, and asynchronously handle, events posted by an InterBase server.

The TIBEvents component is discussed more fully in **Chapter 20, "Programming with Database Events."**

The InterBase Admin tab

The InterBase Admin tab in Delphi 5 contains the IBX services components, along with the install and uninstall components.

FIGURE 8.2 InterBase Admin tab

The InterBase Admin tab contains the following components, from left to right:

- **TIBConfigService**
- **TIBBackupService**
- **TIBRestoreService**
- **TIBValidationService**
- **TIBStatisticalService**
- **TIBLogService**
- **TIBSecurityService**
- **TIBLicensingService**
- **TIBServerProperties**
- **TIBInstall**
- **TIBUnInstall**

These components are discussed in the following sections.

ConfigService Component

TIBConfigService

Use a *TIBConfigService* component to configure database parameters, including page buffers, access mode, and sweep interval.

The *TIBConfigService* component is discussed more fully in **"Setting database properties" on page 272**.

BackupService Component

TIBBackupService

Use the *TIBBackService* component to back up databases. With a *TIBBackupService* component in your application, you can set such parameters as the blocking factor, backup file name, and database backup options.

The *TIBBackupService* component is discussed more fully in **"Backing up and restoring databases" on page 277**.

RestoreService Component

TIBRestoreService

Use the *TIBRestoreService* component to restore a database. With a *TIBRestoreService* component in your application, you can set such parameters as page buffers, page size, and database restore options.

The *TIBRestoreService* component is discussed more fully in **"Backing up and restoring databases" on page 277**.

Validation Service Component

TIBValidationService

Use the *TIBValidationService* component to validate you database and reconcile your database transactions. With the *TIBValidationService*, you can set the default transaction action, return limbo transaction information, and set other database validation options.

The *TIBValidationService* component is discussed more fully in **"Performing database maintenance" on page 285**.

StatisticalService Component

TIBStatisticalService

Use the *TIBStatisticalService* component to view database statistics such as data pages, database log, header pages, index pages, and system relations.

The *TIBStatisticalService* component is discussed more fully in **“Requesting database and server status reports” on page 288**.

TIBLogService

LogService Component

Use the *TIBLogService* component to create an InterBase log file for your application.

The *TIBLogService* component is discussed more fully in **“Using the log service” on page 290**.

TIBSecurityService

SecurityService Component

Use the *TIBSecurityService* component to manage user access to the InterBase server. With a *TIBSecurityService* component in your application, you can create, delete, and modify user accounts, display user information, and set up work groups using SQL roles.

The *TIBSecurityService* component is discussed more fully in **“Configuring users” on page 291**.

TIBLicensingService

LicensingService Component

Use the *TIBLicensingService* component to add, view, or remove InterBase software activation certificates.

The *TIBLicensingService* component is discussed more fully in **“Administering software activation certificates” on page 294**.

TIBServerProperties

ServerProperties Component

Use the *TIBServerProperties* component to return database server information, including configuration parameters, and version and license information.

The *TIBServerProperties* component is discussed more fully in **“Displaying server properties” on page 296**.

TIBInstall

Install Component

Use the *TIBInstall* component to set up an InterBase installation component, including the installation source and destination directories, and the components to be installed.

The *TIBInstall* component is discussed more fully in **Chapter 21, "Writing Installation Wizards."**

TIBUnInstall

UnInstall Component

Use the *TIBUnInstall* component to set up an InterBase uninstall component.

The *TIBUnInstall* component is discussed more fully in **Chapter 21, "Writing Installation Wizards."**

CHAPTER

9

Designing Database Applications

Database applications allow users to interact with information that is stored in databases. Databases provide structure for the information, and allow it to be shared among different applications.

The InterBase Express (IBX) components in Delphi provide support for relational database applications. Relational databases organize information into tables, which contain rows (records) and columns (fields). These tables can be manipulated by simple operations known as relational calculus.

When designing a database application, you must understand how the data is structured. Based on that structure, you can then design a user interface to display data to the user and allow the user to enter new information or modify existing data.

This chapter introduces some common considerations for designing a database application and the decisions involved in designing a user interface.

The following topics introduce common considerations when designing a database application:

- **Using databases**
- **Database architecture**
- **Designing the user interface**

Using databases

The components on the InterBase page of the Component palette allow your application to read from and write to databases. These components access database information which they make available to the data-aware controls in your user interface.

Local InterBase databases

Local databases reside on your local drive or on a local area network. They use the InterBase proprietary APIs for accessing the data. Often, they are dedicated to a single system. When they are shared by several users, they use file-based locking mechanisms. Because of this, they are sometimes called file-based databases.

Local databases can be faster than remote database servers because they often reside on the same system as the database application.

Because they are file-based, local databases are more limited than remote database servers in the amount of data they can store. Therefore, in deciding whether to use a local database, you must consider how much data the tables are expected to hold.

Applications that use local databases are called single-tiered applications because the application and the database share a single file system.

Remote InterBase database servers

Remote database servers usually reside on a remote machine. They use Structured Query Language (SQL) to enable clients to access the data. Because of this, they are sometimes called SQL servers. (Another name is Remote Database Management system, or RDBMS.)

Remote database servers are designed for access by several users at the same time. Instead of a file-based locking system such as those employed by local databases, they provide more sophisticated multi-user support, based on transactions.

Remote database servers hold more data than local databases. Sometimes, the data from a remote database server does not even reside on a single machine, but is distributed over several servers.

Applications that use remote database servers are called two-tiered applications or multi-tiered applications because the application and the database operate on independent systems (or tiers).

Database security

Databases often contain sensitive information. When users try to access protected tables, they are required to provide a password. Once users have been authenticated, they can see only those fields (columns) for which they have permission.

For access to InterBase databases on a server, a valid user name and password is required. Once the user has logged in to the database, that username and password (and sometimes, role) determine which tables can be used. For information on providing passwords to InterBase servers, see **Controlling server login on page 159**, or "Database security" in the *InterBase 6 Operations Guide*.

If you are requiring your user to supply a password, you must consider when the password is required. If you are using a local database but intend to scale up to a larger SQL server later, you may want to prompt for the password before you access the table, even though it is not required until then.

If your application requires multiple passwords because you must log in to several protected systems or databases, you can have your users provide a single master password which is used to access a table of passwords required by the protected systems. The application then supplies passwords programmatically, without requiring the user to provide multiple passwords.

In multi-tiered applications, you may want to use a different security model altogether. You can use CORBA or MTS to control access to middle tiers, and let the middle tiers handle all details of logging into database servers.

Transactions

A transaction is a group of actions that must all be carried out successfully on one or more tables in a database before they are committed (made permanent). If any of the actions in the group fails, then all actions are rolled back (undone).

Client applications can start multiple simultaneous transactions. InterBase provides full and explicit transaction control for starting, committing, and rolling back transactions. The statements and functions that control starting a transaction also control transaction behavior.

InterBase transactions can be isolated from changes made by other concurrent transactions. For the life of these transactions, the database appears to be unchanged except for the changes made by the transaction. Records deleted by another transaction exist, newly stored records do not appear to exist, and updated records remain in the original state.

For details on using transactions in database applications, see **Using transactions on page 150**. For details on using transactions in multi-tiered applications, see "Creating multi-tiered applications" in the *Delphi 5 Developer's Guide*.

The Data Dictionary

No matter what type of database you use, your application has access to the Data Dictionary. The Data Dictionary provides a customizable storage area, independent of your applications, where you can create extended field attribute sets that describe the content and appearance of data.

For example, if you frequently develop financial applications, you may create a number of specialized field attribute sets describing different display formats for currency. When you create datasets for your application at design time, rather than using the Object Inspector to set the currency fields in each dataset by hand, you can associate those fields with an extended field attribute set in the data dictionary. Using the data dictionary ensures a consistent data appearance within and across the applications you create.

In a Client/Server environment, the Data Dictionary can reside on a remote server for additional sharing of information.

To learn how to create extended field attribute sets from the Fields editor at design time, and how to associate them with fields throughout the datasets in your application, see "Creating attribute sets for field components" in the *Delphi 5 Developer's Guide*. To learn more about creating a data dictionary and extended field attributes with the SQL and Database Explorers, see their respective online help files.

A programming interface to the Data Dictionary is available in the *drintf* unit (located in the **lib** directory). This interface supplies the following methods:

Routine	Use
DictionaryActive	Indicates if the data dictionary is active
DictionaryDeactivate	Deactivates the data dictionary
IsNullID	Indicates whether a given ID is a null ID
FindDatabaseID	Returns the ID for a database given its alias
FindTableID	Returns the ID for a table in a specified database
FindFieldID	Returns the ID for a field in a specified table

TABLE 9.1 Data Dictionary interface

Routine	Use
FindAttrID	Returns the ID for a named attribute set
GetAttrName	Returns the name an attribute set given its ID
GetAttrNames	Executes a callback for each attribute set in the dictionary
GetAttrID	Returns the ID of the attribute set for a specified field
NewAttr	Creates a new attribute set from a field component
UpdateAttr	Updates an attribute set to match the properties of a field
CreateField	Creates a field component based on stored attributes
UpdateField	Changes the properties of a field to match a specified attribute set
AssociateAttr	Associates an attribute set with a given field ID
UnassociateAttr	Removes an attribute set association for a field ID
GetControlClass	Returns the control classs for a specified attribute ID
QualifyTableName	Returns a fully qualified table name (qualified by user name)
QualifyTableNameByName	Returns a fully qualified table name (qualified by user name)
HasConstraints	Indicates whether the dataset has constraints in the dictionary
UpdateConstraints	Updates the imported constraints of a dataset
UpdateDataset	Updates a dataset to the current settings and constraints in the dictionary

TABLE 9.1 Data Dictionary interface

Referential integrity, stored procedures, and triggers

All relational databases have certain features in common that allow applications to store and manipulate data. InterBase also provides other database-specific, features that can prove useful for ensuring consistent relationships between the tables in a database. These include:

- *Referential integrity.* Referential integrity provides a mechanism to prevent master/detail relationships between tables from being broken. When the user attempts to delete a field in a master table which would result in orphaned detail records, referential integrity rules prevent the deletion or automatically delete the orphaned detail records.

- *Stored procedures.* Stored procedures are sets of SQL statements that are named and stored on an SQL server. Stored procedures usually perform common database-related tasks on the server, and return sets of records (datasets).
- *Triggers.* Triggers are sets of SQL statements that are automatically executed in response to a particular command.

Database architecture

Database applications are built from user interface elements, components that manage the database or databases, and components that represent the data contained by the tables in those databases (datasets). How you organize these pieces is the architecture of your database application.

By isolating database access components in data modules, you can develop forms in your database applications that provide a consistent user interface. By storing links to well-designed forms and data modules in the Object Repository, you and other developers can build on existing foundations rather than starting over from scratch for each new project. Sharing forms and modules also makes it possible for you to develop corporate standards for database access and application interfaces.

Many aspects of the architecture of your database application depend on the number of users who will be sharing the database information and the type of information you are working with.

When writing applications that use information that is not shared among several users, you may want to use a local database in a *single-tiered application*. This approach can have the advantage of speed (because data is stored locally), and does not require the purchase of a separate database server and expensive site licences. However, it is limited in how much data the tables can hold and the number of users your application can support.

Writing a *two-tiered application* provides more multi-user support and lets you use large remote databases that can store far more information.

Note Support for two-tiered applications requires SQL Links.

When the database information includes complicated relationships between several tables, or when the number of clients grows, you may want to use a *multi-tiered application*. Multi-tiered applications include middle tiers that centralize the logic which governs your database interactions so that there is centralized control over data relationships. This allows different client applications to use the same data while

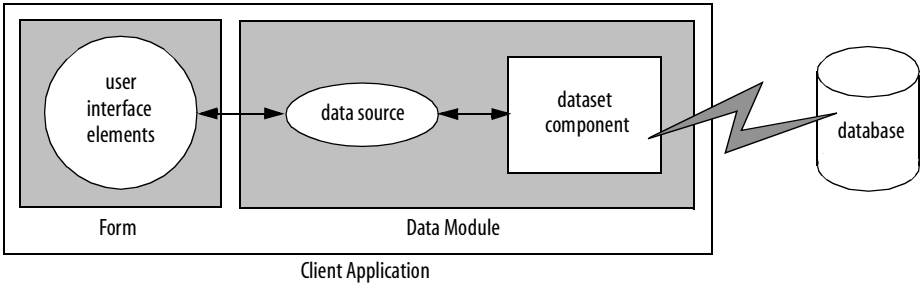
ensuring that the data logic is consistent. They also allow for smaller client applications because much of the processing is off-loaded onto middle tiers. These smaller client applications are easier to install, configure, and maintain because they do not include the database connectivity software. Multi-tiered applications can also improve performance by spreading the data-processing tasks over several systems.

Planning for scalability

The development process can get more involved and expensive as the number of tiers increases. Because of this, you may wish to start developing your application as a single-tiered application. As the amount of data, the number of users, and the number of different applications accessing the data grows, you may later need to scale up to a multi-tiered architecture. By planning for scalability, you can protect your development investment when writing a single- or two-tiered application so that the code can be reused as your application grows.

The VCL data-aware components make it easy to write scalable applications by abstracting the behavior of the database and the data stored by the database. Whether you are writing a single-tiered, two-tiered, or multi-tiered application, you can isolate your user interface from the data access layer as illustrated in **FIGURE 9.1**.

FIGURE 9.1 User-interface to dataset connections in all database applications



A form represents the user interface, and contains data controls and other user interface elements. The data controls in the user interface connect to datasets which represent information from the tables in the database. A data source links the data controls to these datasets. By isolating the data source and datasets in a data module, the form can remain unchanged as you scale your application up. Only the datasets must change.

A flat-file database application is easily scaled to the client in a multi-tiered application because both architectures use the same client dataset component. In fact, you can write an application that acts as both a flat-file application and a multi-tiered client (see **Using the briefcase model on page 153**).

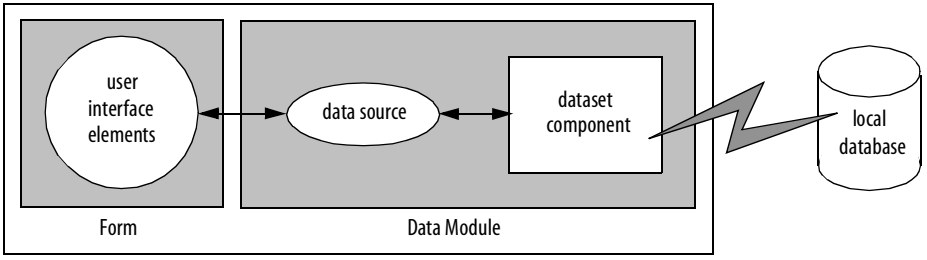
If you plan to scale your application up to a three-tiered architecture eventually, you can write your one- or two-tiered application with that goal in mind. In addition to isolating the user interface, isolate all logic that will eventually reside on the middle tier so that it is easy to replace at a later time. You can even connect your user interface elements to client datasets (used in multi-tiered applications), and connect them to local versions of the datasets in a separate data module that will eventually move to the middle tier. If you do not want to introduce this artifice of an extra dataset layer in your one- and two-tiered applications, it is still easy to scale up to a three-tiered application at a later date. See **Scaling up to a three-tiered application on page 154** for more information.

Single-tiered database applications

In single-tiered database applications, the application and the database share a single file system. They use local databases or files that store database information in a flat-file format.

A single application comprises the user interface and incorporates the data access mechanism. The type of dataset component used to represent database tables is in a flat file. **FIGURE 9.2** illustrates this:

FIGURE 9.2 Single-tiered database application architecture

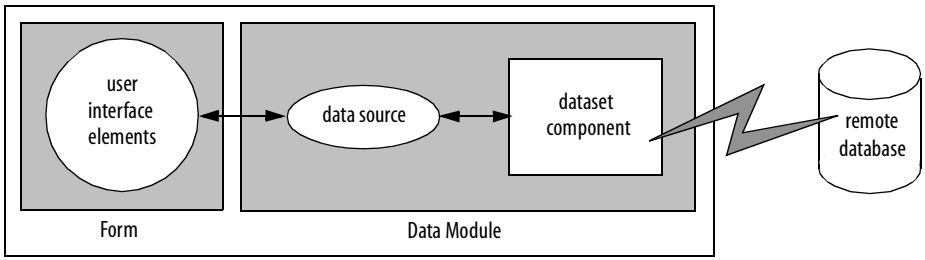


For more information on building single-tiered database applications, see **Chapter 10, "Building One- and Two-tiered Applications."**

Two-tiered database applications

In two-tiered database applications, a client application provides a user interface to data, and interacts directly with a remote database server. **FIGURE 9.3** illustrates this relationship.

FIGURE 9.3 Two-tiered database application architecture

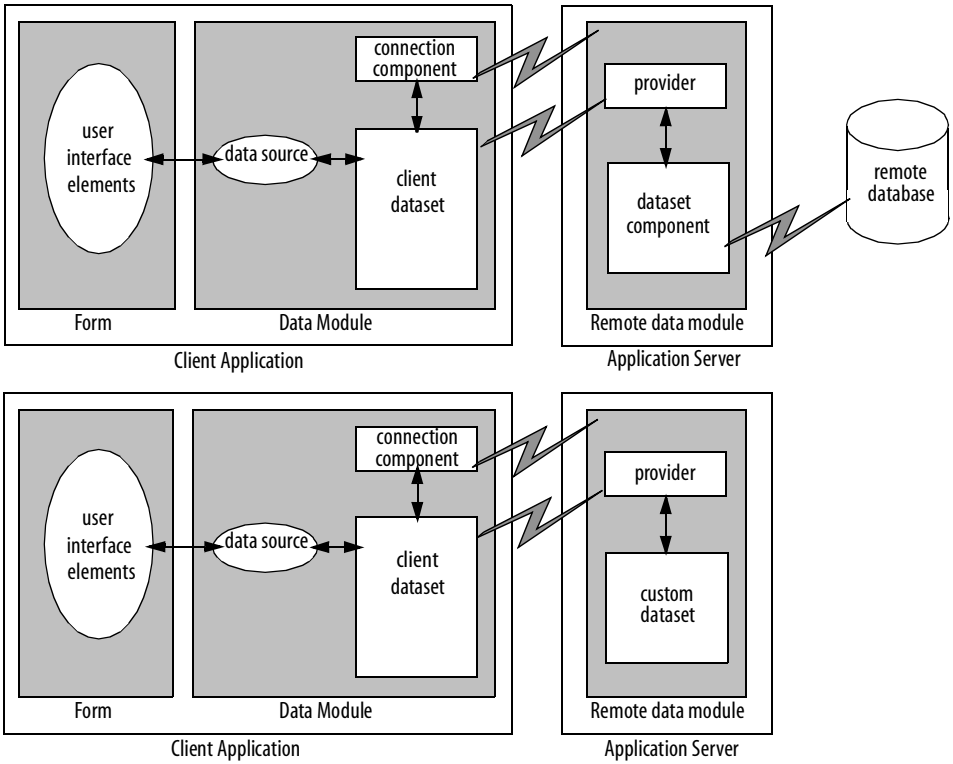


In this model, all applications are database clients. A *client* requests information from and sends information to a database server. A server can process requests from many clients simultaneously, coordinating access to and updating of data.

Multi-tiered database applications

In multi-tiered database applications, an application is partitioned into pieces that reside on different machines. A client application provides a user interface to data. It passes all data requests and updates through an application server (also called a "remote data broker"). The application server, in turn, communicates directly with a remote database server or some other custom dataset. Usually, in this model, the client application, the application server, and the remote database server are on separate machines. **FIGURE 9.4** illustrates these relationships for multi-tiered applications.

FIGURE 9.4 Multi-tiered database architectures



You can use Delphi to create both client applications and application servers. The client application uses standard data-aware controls connected through a data source to one or more client dataset components in order to display data for viewing and editing. Each client dataset communicates with an application server through an *IProvider* interface that is part of the application server's remote data module. The client application can use a variety of protocols (TCP/IP, DCOM, MTS, or CORBA) to establish this communication. The protocol depends on the type of connection component used in the client application and the type of remote data module used in the server application.

The application server creates the *IProvider* interfaces in one of two ways. If the application server includes any provider objects, then these objects are used to create the *IProvider* interface. This is the method illustrated in the previous figure. Using a provider component gives an application more control over the interface. All data is passed between the client application and the application server through the interface. The interface receives data from and sends updates to conventional datasets, and these components communicate with a database server.

Usually, several client applications communicate with a single application server in the multi-tiered model. The application server provides a gateway to your databases for all your client applications, and it lets you provide enterprise-wide database tasks in a central location, accessible to all your clients. For more information about creating and using a multi-tiered database application, see "Creating multi-tiered applications" in the in the Delphi 5 Developer's Guide.

Designing the user interface

The Data Controls page of the Component palette provides a set of data-aware controls that represent data from fields in a database record, and can permit users to edit that data and post changes back to the database. Using data-aware controls, you can build your database application's user interface (UI) so that information is visible and accessible to users. For more information on data-aware controls see "Using Data Controls" in the *Delphi 5 Developer's Guide*.

Data-aware controls get data from and send data to a data source component, *TDataSource*. A data source component acts as a conduit between the user interface and a dataset component which represents a set of information from the tables in a database. Several data-aware controls on a form can share a single data source, in which case the display in each control is synchronized so that as the user scrolls through records, the corresponding value in the fields for the current record is displayed in each control. An application's data source components usually reside in a data module, separate from the data-aware controls on forms.

The data-aware controls you add to your user interface depend on what type of data you are displaying (plain text, formatted text, graphics, multimedia elements, and so on). In addition, your choice of controls is determined by how you want to organize the information and how (or if) you want to let users navigate through the records of datasets and add or edit data.

The following sections introduce the components you can use for various types of user interface:

- **Displaying a single record**

- **Displaying multiple records**
- **Analyzing data**
- **Selecting what data to show**

Displaying a single record

In many applications, you may only want to provide information about a single record of data at a time. For example, an order-entry application may display the information about a single order without indicating what other orders are currently logged. This information probably comes from a single record in an orders dataset.

Applications that display a single record are usually easy to read and understand, because all database information is about the same thing (in the previous case, the same order). The data-aware controls in these user interfaces represent a single field from a database record. The Data Controls page of the Component palette provides a wide selection of controls to represent different kinds of fields. For more information about specific data-aware controls, see "Controls that represent a single field" in the "Using data controls" chapter of the *Delphi 5 Developer's Guide*.

Displaying multiple records

Sometimes you want to display many records in the same form. For example, an invoicing application might show all the orders made by a single customer on the same form.

To display multiple records, use a grid control. Grid controls provide a multi-field, multi-record view of data that can make your application's user interface more compelling and effective. They are discussed in "Viewing and editing data with TDBGrid" and "Creating a grid that contains other data-aware controls" in the "Using data controls" chapter of the *Delphi 5 Developer's Guide*.

You may want to design a user interface that displays both fields from a single record and grids that represent multiple records. There are two models that combine these two approaches:

- *Master-detail forms:* You can represent information from both a master table and a detail table by including both controls that display a single field and grid controls. For example, you could display information about a single customer with a detail grid that displays the orders for that customer. For information about linking the underlying tables in a master-detail form, see **Creating master/detail forms on page 189**.

- *Drill-down forms:* In a form that displays multiple records, you can include single field controls that display detailed information from the current record only. This approach is particularly useful when the records include long memos or graphic information. As the user scrolls through the records of the grid, the memo or graphic updates to represent the value of the current record. Setting this up is very easy. The synchronization between the two displays is automatic if the grid and the memo or image control share a common data source.

  Note It is generally not a good idea to combine these two approaches on a single form. While the result can sometimes be effective, it is usually confusing for users to understand the data relationships.

Analyzing data

Some database applications do not present database information directly to the user. Instead, they analyze and summarize information from databases so that users can draw conclusions from the data.

The *TDBChart* component on the Data Controls page of the Component palette lets you present database information in a graphical format that enables users to quickly grasp the import of database information.

In addition, some versions of Delphi include a Decision Cube page on the Component palette. It contains six components that let you perform data analysis and cross-tabulations on data when building decision support applications. For more information about using the Decision Cube components, see "Using decision support components" in the *Delphi 5 Developer's Guide*.

If you want to build your own components that display data summaries based on various grouping criteria, you can use maintained aggregates with a client dataset. For more information about using maintained aggregates, see "Using maintained aggregates" in the "Creating and using a client dataset" chapter of the *Delphi 5 Developer's Guide*.

Selecting what data to show

Often, the data you want to surface in your database application does not correspond exactly to the data in a single database table. You may want to use only a subset of the fields or a subset of the records in a table. You may want to combine the information from more than one table into a single joined view.

The data available to your database application is controlled by your choice of dataset component. Datasets abstract the properties and methods of a database table, so that you do not need to make major alterations depending on whether the data is stored in a database table or derived from one or more tables in the database. For more information on the common properties and methods of datasets, see **Chapter 12, "Understanding Datasets."**

Your application can contain more than one dataset. Each dataset represents a logical table. By using datasets, your application logic is buffered from restructuring of the physical tables in your databases. You might need to alter the type of dataset component, or the way it specifies the data it contains, but the rest of your user interface can continue to work without alteration.

You can use any of the following types of dataset:

- *Table components:* Tables (*TIBTable*) correspond directly to the underlying tables in the database. You can adjust which fields appear (including adding lookup fields and calculated fields) by using persistent field components. You can limit the records that appear using ranges or filters. Tables are described in more detail in **Chapter 13, "Working with Tables."** Persistent fields are described in "Persistent field components" in the *Delphi 5 Developer's Guide*. Ranges and filters are described in **Working with a subset of data on page 185**.
- *Query components:* Queries (*TIBQuery*, *TIBDataSet*, and *TIBSQL*) provide the most general mechanism for specifying what appears in a dataset. You can combine the data from multiple tables using joins, and limit the fields and records that appear based on any criteria you can express in SQL. For more information on queries, see **Chapter 14, "Working with Queries."**
- *Stored procedures:* Stored procedures (*TIBStoredProc*) are sets of SQL statements that are named and stored on an SQL server. If your database server defines a remote procedure that returns the dataset you want, you can use a stored procedure component. For more information on stored procedures, see **Chapter 15, "Working with Stored Procedures."**
- *Client datasets:* Client datasets cache the records of the logical dataset in memory. Because of that, they can only hold a limited number of records. Client datasets are populated with data in one of two ways: from an application server or from flat-file data stored on disk. When using a client dataset to represent flat-file data, you must create the underlying table programmatically. For more information about client datasets, see "Creating and using a client dataset" in the *Delphi 5 Developer's Guide*.

- *Custom datasets:* You can create your own custom descendants of *TDataSet* to represent a body of data that you create or access in code you write. Writing custom datasets allows you the flexibility of managing the data using any method you choose, while still letting you use the VCL data controls to build your user interface. For more information about creating custom components, see "Overview of component creation" in the *Delphi 5 Developer's Guide*.

CHAPTER

10

Building One- and Two-tiered Applications

One- and two-tiered applications include the logic that manipulates database information in the same application that implements the user interface. Because the data manipulation logic is not isolated in a separate tier, these types of applications are most appropriate when there are no other applications sharing the same database information. Even when other applications share the database information, these types of applications are appropriate if the database is very simple, and there are no data semantics that must duplicated by all applications that use the data.

You may want to start by writing a one- or two-tiered application, even when you intend to eventually scale up to a multi-tiered model as your needs increase. This approach lets you avoid having to develop data manipulation logic up front so that the application server can be available while you are writing the user interface. It also allows you to develop a simpler, cheaper prototype before investing in a large, multi-system development project. If you intend to eventually scale up to a multi-tiered application, you can isolate the data manipulation logic so that it is easy to move it to a middle tier at a later date.

Understanding databases and datasets

Databases contain information stored in tables. They may also include tables of information about what is contained in the database, objects such as indexes that are used by tables, and SQL objects such as stored procedures. See **Chapter 11, "Connecting to Databases"** for more information about databases.

The InterBase page of the Component palette contains various dataset components that represent the tables contained in a database or logical tables constructed out of data stored in those database tables. See **Selecting what data to show on page 145** for more information about these dataset components. You must include a dataset component in your application to work with database information.

Each dataset component on the InterBase page has a published *Database* property that specifies the database which contains the table or tables that hold the information in that dataset. When setting up your application, you must use this property to specify the database before you can bind the dataset to specific information contained in that database. What value you specify depends on whether or not you are using explicit database components. Database components (*TIBDatabase*) represent a database in your application. If you do not add a database component explicitly, a temporary one is created for you automatically, based on the value of the *Database* property. If you are using explicit database components, *Database* is the value of the *Database* property of the database component. See **Understanding persistent and temporary database components on page 157** for more information about using database components.

Using transactions

A *transaction* is a group of actions that must all be carried out successfully on one or more tables in a database before they are *committed* (made permanent). If one of the actions in the group fails, then all actions are *rolled back* (undone). By using transactions, you ensure that the database is not left in an inconsistent state when a problem occurs completing one of the actions that make up the transaction.

For example, in a banking application, transferring funds from one account to another is an operation you would want to protect with a transaction. If, after decrementing the balance in one account, an error occurred incrementing the balance in the other, you want to roll back the transaction so that the database still reflects the correct total balance.

By default, implicit transaction control is provided for your applications. When an application is under implicit transaction control, a separate transaction is used for each record in a dataset that is written to the underlying database. Implicit transactions guarantee both a minimum of record update conflicts and a consistent view of the database. On the other hand, because each row of data written to a database takes place in its own transaction, implicit transaction control can lead to excessive network traffic and slower application performance. Also, implicit transaction control will not protect logical operations that span more than one record, such as the transfer of funds described previously.

If you explicitly control transactions, you can choose the most effective times to start, commit, and roll back your transactions. When you develop applications in a multi-user environment, particularly when your applications run against a remote SQL server, you should control transactions explicitly.

Note You can also minimize the number of transactions you need by caching updates. For more information about cached updates, see **Chapter 16, "Working with Cached Updates."**

Using a transaction component

When you start a transaction, all subsequent statements that read from and write to the database occur in the context of that transaction. Each statement is considered part of a group. Changes must be successfully committed to the database, or every change made in the group must be undone.

Ideally, a transaction should only last as long as necessary. The longer a transaction is active, the more simultaneous users that access the database, and the more concurrent, simultaneous transactions that start and end during the lifetime of your transaction, the greater the likelihood that your transaction will conflict with another when you attempt to commit your changes.

When using a transaction component, you code a single transaction as follows:

1. Start the transaction by calling the transaction's *StartTransaction* method:

   ```
   IBTransaction.StartTransaction;
   ```

2. Once the transaction is started, all subsequent database actions are considered part of the transaction until the transaction is explicitly terminated. You can determine whether a transaction is in process by checking the transaction component's *InTransaction* property.

3. When the actions that make up the transaction have all succeeded, you can make the database changes permanent by using the transaction component's *Commit* method:

```
IBTransaction.Commit;
```

Alternately, you can commit the transaction while retaining the current transaction context using the *CommitRetaining* method:

```
IBTransaction.CommitRetaining;
```

Commit is usually attempted in a **try...except** statement. That way, if a transaction cannot commit successfully, you can use the **except** block to handle the error and retry the operation or to roll back the transaction.

If an error occurs when making the changes that are part of the transaction, or when trying to commit the transaction, you will want to discard all changes that make up the transaction. To discard these changes, use the database component's *Rollback* method:

```
IBTransaction.Rollback;
```

You can also rollback the transaction while retaining the current transaction context using the *RollbackRetaining* method:

```
IBTransaction.RollbackRetaining;
```

Rollback usually occurs in

- Exception handling code when you cannot recover from a database error.
- Button or menu event code, such as when a user clicks a Cancel button.

Caching updates

InterBase Express (IBX) provides support for caching updates. When you cache updates, your application retrieves data from a database, makes all changes to a local, cached copy of the data, and applies the cached changes to the dataset as a unit. Cached updates are applied to the database in a single transaction.

Caching updates can minimize transaction times and reduce network traffic. However, cached data is local to your application and is not under transaction control. This means that while you are working on your local, in-memory, copy of the data, other applications can be changing the data in the underlying database table. They also cannot see any changes you make until you apply the cached updates. Because of this, cached updates may not be appropriate for applications that work with volatile data, as you may create or encounter too many conflicts when trying to merge your changes into the database.

You can tell datasets to cache updates using the *CachedUpdates* property. When the changes are complete, they can be applied by the dataset component, by the database component, or by a special update object. When changes cannot be applied to the database without additional processing (for example, when working with a joined query), you must use the *OnUpdateRecord* event to write changes to each table that makes up the joined view.

For more information on caching updates, see **Chapter 16, "Working with Cached Updates."**

Note If you are caching updates, you may want to consider moving to a multi-tiered model to have greater control over the application of updates. For more information about the multi-tiered model, see "Creating multi-tiered applications" in the *Delphi 5 Developer's Guide*.

Creating and restructuring database tables

You can use the *TIBTable* component to create new database tables and to add indexes to existing tables.

You can create tables either at design time, in the Forms Designer, or at runtime. To create a table, you must specify the fields in the table using the *FieldDefs* property, add any indexes using the *IndexDefs* property, and call the *CreateTable* method (or select the Create Table command from the table's context menu). For more detailed instructions on creating tables, see **"Creating a table" on page 186**.

You can add indexes to an existing table using the *AddIndex* method of *TIBTable*.

Note To create and restructure tables on remote servers at design time, use the SQL Explorer and restructure the table using SQL.

Using the briefcase model

Most of this chapter has described creating and using a client dataset in a one-tiered application. The one-tiered model can be combined with a multi-tiered model to create what is called the briefcase model. In this model, a user starts a client application on one machine and connects over a network to an application server on a remote machine. The client requests data from the application server, and sends updates to it. The updates are applied by the application server to a database that is presumably shared with other clients throughout an organization.

Note The briefcase model is sometimes called the disconnected model, or mobile computing.

Suppose, however, that your on-site company database contains valuable customer contact data that your sales representatives can use and update in the field. In this case, it would be useful if your sales reps could download some or all of the data from the company database, work with it on their laptops as they fly across the country, and even update records at existing or new customer sites. When the sales reps return on-site, they could upload their data changes to the company database for everyone to use. The ability to work with data off-line and then apply updates online at a later date is known as the "briefcase" model.

By using the briefcase model, you can take advantage of the client dataset component's ability to read and write data to flat files to create client applications that can be used both online with an application server, and off-line, as temporary one-tiered applications.

To implement the briefcase model, you must

1. Create a multi-tiered server application as described in "Creating multi-tiered applications" in the *Delphi 5 Developer's Guide*.
2. Create a flat-file database application as your client application. Add a connection component and set the *RemoteServer* property of your client datasets to specify this connection component. This allows them to talk to the application server created in step 1. For more information about connection components, see "Connecting to the application server" in the *Delphi 5 Developer's Guide*.
3. In the client application, try on start-up to connect to the application server. If the connection fails, prompt the user for a file and read in the local copy of the data.
4. In the client application, add code to apply updates to the application server. For more information on sending updates from a client application to an application server, see "Updating records" in the *Delphi 5 Developer's Guide*.

Scaling up to a three-tiered application

In a two-tiered client/server application, the application is a client that talks directly to a database server. Even so, the application can be thought of as having two parts: a database connection and a user interface. To make a two-tiered client/server application into a multi-tiered application you must:

- Split your existing application into an application server that handles the database connection, and into a client application that contains the user interface.

- Add an interface between the client and the application server.

  There are a number of ways to proceed, but the following sequential steps may best keep your translation work to a minimum:

  1. Create a new project for the application server, duplicate the relevant database connection portions of your former two-tiered application, and for each dataset, add a provider component that will act as a data conduit between the application server and the client. For more information on using a provider component, see "Creating a data provider for the application server" in the *Delphi 5 Developer's Guide*.
  2. Copy your existing two-tiered project, remove its direct database connections, add an appropriate connection component to it. For more information about creating and using connection components, see "Connecting to the application server" in the *Delphi 5 Developer's Guide*.
  3. Substitute a client dataset for each dataset component in the original project. For general information about using a client dataset component, see "Creating and using a client dataset" in the *Delphi 5 Developer's Guide*.
  4. In the client application, add code to apply updates to the application server. For more information on sending updates from a client application to an application server, see "Updating records" in the *Delphi 5 Developer's Guide*.
  5. Move the dataset components to the application server's data modules. Set the *DataSet* property of each provider to specify the corresponding datasets. For more information about linking a dataset to a provider component, see "Creating a data provider for the application server" in the *Delphi 5 Developer's Guide*.

Creating multi-tiered applications

A multi-tiered client/server application is partitioned into logical units which run in conjunction on separate machines. Multi-tiered applications share data and communicate with one another over a local-area network or even over the Internet. They provide many benefits, such as centralized business logic and thin client applications. For information on how to build multi-tiered applications, refer to "Creating multi-tiered applications" in the *Delphi 5 Developers Guide*.

CHAPTER

11

Connecting to Databases

When an InterBase Express (IBX) application connects to a database, that connection is encapsulated by a *TIBDatabase* component. A database component encapsulates the connection to a single database in an application. This chapter describes database components and how to manipulate database connections.

Another use for database components is applying cached updates for related tables. For more information about using a database component to apply cached updates, see **"Applying cached updates with a database component method" on page 234**.

Understanding persistent and temporary database components

Each database connection in an application is encapsulated by a database component whether or not you explicitly provide a database component at design time or create it dynamically at runtime. When an application attempts to connect to a database, it either uses an explicitly instantiated, or *persistent*, database component, or it generates a temporary database component that exists only for the lifetime of the connection.

Temporary database components are created as necessary for any datasets in a data module or form for which you do not create yourself. Temporary database components provide broad support for many typical desktop database applications without requiring you to handle the details of the database connection. For most client/server applications, however, you should create your own database components instead of relying on temporary ones. You gain greater control over your databases, including the ability to

- Create persistent database connections
- Customize database server logins
- Control transactions and specify transaction isolation levels

Using temporary database components

Temporary database components are automatically generated as needed. For example, if you place a *TIBTable* component on a form, set its properties, and open the table without first placing and setting up a *TIBDatabase* component and associating the table component with it, Delphi creates a temporary database component for you behind the scenes.

The default properties created for temporary database components provide reasonable, general behaviors meant to cover a wide variety of situations. For complex, mission-critical client/server applications with many users and different requirements for database connections, however, you should create your own database components to tune each database connection to your application's needs.

Creating database components at design time

The InterBase page of the Component palette contains a database component you can place in a data module or form. The main advantages to creating a database component at design time are that you can set its initial properties and write *OnLogin* events for it. *OnLogin* offers you a chance to customize the handling of security on a database server when a database component first connects to the server. For more information about managing connection properties, see **"Connecting to a database server" on page 160**. For more information about server security, see **"Controlling server login" on page 159**.

Controlling connections

Whether you create a database component at design time or runtime, you can use the properties, events, and methods of *TIBDatabase* to control and change its behavior in your applications. The following sections describe how to manipulate database components. For details about all *TIBDatabase* properties, events, and methods, see TIBDatabase in the online *InterBase Express Reference*.

Controlling server login

InterBase servers include security features to prohibit unauthorized access. The server requires a user name and password login before permitting database access.

At design time, a standard Login dialog box prompts for a user name and password when you first attempt to connect to the database.

At runtime, there are three ways you can handle a server's request for a login:

- Set the *LoginPrompt* property of a database component to *True* (the default). Your application displays the standard Login dialog box when the server requests a user name and password.
- Set the *LoginPrompt* to *False*, and include hard-coded USER_NAME and PASSWORD parameters in the *Params* property for the database component. For example:

```
USER_NAME=SYSDBA
PASSWORD=masterkey
```

IMPORTANT Note that because the *Params* property is easy to view, this method compromises server security, so it is not recommended.

- Write an *OnLogin* event for the database component, and use it to set login parameters at runtime. *OnLogin* gets a copy of the database component's *Params* property, which you can modify. The name of the copy in *OnLogin* is *LoginParams*. Use the *Values* property to set or change login parameters as follows:

```
LoginParams.Values['USER_NAME'] := UserName;
LoginParams.Values['PASSWORD'] := PasswordSearch(UserName);
```

On exit, *OnLogin* passes its *LoginParams* values back to *Params*, which is used to establish a connection.

Connecting to a database server

There are two ways to connect to a database server using a database component:

- Call the *Open* method.
- Set the *Connected* property to *True*.

Setting *Connected* to *True* executes the *Open* method. *Open* verifies that the database specified by the *Database* or *Directory* properties exists, and if an *OnLogin* event exists for the database component, it is executed. Otherwise, the default Login dialog box appears.

Note When a database component is not connected to a server and an application attempts to open a dataset associated with the database component, the database component's *Open* method is first called to establish the connection. If the dataset is not associated with an existing database component, a temporary database component is created and used to establish the connection.

Once a database connection is established the connection is maintained as long as there is at least one active dataset. If a dataset is later opened which uses the database, the connection must be reestablished and initialized.

Working with network protocols

As part of configuring the appropriate SQL Links or ODBC driver, you may need to specify the network protocol used by the server, such as SPX/IPX or TCP/IP, depending on the driver's configuration options. In most cases, network protocol configuration is handled using a server's client setup software. For ODBC it may also be necessary to check the driver setup using the ODBC driver manager.

Establishing an initial connection between client and server can be problematic. The following troubleshooting checklist should be helpful if you encounter difficulties:

- Is your server's client-side connection properly configured?
- If you are using TCP/IP:
  - Is your TCP/IP communications software installed? Is the proper **WINSOCK.DLL** installed?
  - Is the server's IP address registered in the client's **HOSTS** file?
  - Is the Domain Name Service (DNS) properly configured?
  - Can you ping the server?
- Are the DLLs for your connection and database drivers in the search path?

For more troubleshooting information, see the online *SQL Links User's Guide*.

Using ODBC

An application can use ODBC data sources (for example, Btrieve). An ODBC driver connection requires

- A vendor-supplied ODBC driver.
- The Microsoft ODBC Driver Manager.

Disconnecting from a database server

There are two ways to disconnect a server from a database component:

- Set the *Connected* property to *False*.
- Call the *Close* method.

Setting *Connected* to *False* calls *Close*. *Close* closes all open datasets and disconnects from the server. For example, the following code closes all active datasets for a database component and drops its connections:

```
IBDatabase1.Connected := False;
```

Iterating through a database component's datasets

A database component provides two properties that enable an application to iterate through all the datasets associated with the component: *DataSets* and *DataSetCount*.

DataSets is an indexed array of all active datasets (*TIBDataSet*, *TIBSQL*, *TIBTable*, *TIBQuery*, and *TIBStoredProc*) for a database component. An active dataset is one that is currently open. *DataSetCount* is a read-only integer value specifying the number of currently active datasets.

You can use *DataSets* with *DataSetCount* to cycle through all currently active datasets in code. For example, the following code cycles through all active datasets to set the *CachedUpdates* property for each dataset of type *TIBTable* to *True*:

```
var
  I: Integer;
begin
  for I := 0 to DataSetCount - 1 do
```

```
    if DataSets[I] is TIBTable then
      DataSets[I].CachedUpdates := True;
end;
```

Requesting information about an attachment

Use a *TIBDatabaseInfo* component in your application to query InterBase for attachment information, such as the version of the on-disk structure (ODS) used by the attachment, the number of database cache buffers allocated, the number of database pages read from or written to, or write ahead log information.

After attaching to a database, you can use the *TIBDatabaseInfo* properties to return information on:

- Database characteristics
- Environmental characteristics
- Performance statistics
- Database operation counts

Database characteristics

Several properties are available for determining database characteristics, such as size and major and minor ODS numbers. The following table lists the properties that can be passed, and the information returned in the result buffer for each property type:

Property	Returns
Allocation	The number of pages allocated as a long integer
BaseLevel	The database version number as a long integer
DBFileName	The database file name as a string
DBImplementationClass	The database implementation class number as a long integer; either 1 or 12
DBImplementationNo	The database implementation number as a long integer
DBSiteName	The database site name as a string

TABLE 11.1 TIBDatabaseInfo database characteristic properties

Property	Returns
DBSQLDialect	The database SQL dialect as a long integer
Handle	The database handle
NoReserve	0 to indicate that space is reserved on each database page for holding backup version of modified records (the default) or 1 to indicate that no space is reserved
ODSMajorVersion	The on disk structure (ODS) major version number as a long integer
ODSMinorVersion	The ODS minor version number as a long integer
PageSize	The number of bytes per page as a long integer
Version	The database version as a string

TABLE 11.1 TIBDatabaseInfo database characteristic properties

Environmental characteristics

Several properties are provided for determining environmental characteristics, such as the amount of memory currently in use, or the number of database cache buffers currently allocated. These properties are described in the following table:

Property	Returns
CurrentMemory	The amount of server memory currently in use (in bytes) as along integer

TABLE 11.2 TIBDatabaseInfo environmental characteristic properties

Property	Returns
ForcedWrites	0 for asynchronous (forced) database writes, or 1 for synchronous writes
MaxMemory	The maximum amount of memory used at one time since the first process attached to database as a long integer
NumBuffers	The number of memory buffers currently allocated as a long integer
SweepInterval	The number of transactions that are committed between sweeps as a long integer
UserNames	The names of all users currently attached to the database as a TStringList

TABLE 11.2 TIBDatabaseInfo environmental characteristic properties

Performance statistics

There are four properties that request performance statistics for a database. The statistics accumulate for a database from the moment it is first attached by any process until the last remaining process detaches from the database. For example, the value returned for the *Reads* property is the number of reads since the current database was first attached, that is, an aggregate of all reads done by all attached processes, rather than the number of reads done for the calling program since it attached to the database:

Property	Returns
Fetches	The number of reads from the memory buffer cache as a long integer
Marks	The number of writes to the memory buffer cache as a long integer
Reads	The number of pages reads from the database since the current database was first attached; returned as a long integer
Writes	The number of page writes to the current database since it was first attached by any process; returned as long integer

TABLE 11.3 TIBDataBaseInfo performance properties

Database operation counts

Several information properties are provided for determining the number of various database operations performed by the currently attached calling program. These values are calculated on a per-table basis.

The following table describes the properties which return count values for operations on the database:

Property	Returns
BackoutCount	The number of removals of a version of a record as a long integer
DeleteCount	The number of database deletes since the database was last attached; returned as long integer
ExpungeCount	The number of removals of a record and all of its ancestors as a long integer
InsertCount	The number of inserts into the database since the database was last attached; returned as a long integer
PurgeCount	The number of removals of fully mature records from the database; returned as a long integer
ReadIdxCount	The number of sequential database reads done via an index since the database was last attached; returned as a long integer
ReadSeqCount	The number of sequential database reads done on each table since the database was last attached; returned as a long integer
UpdateCount	The number of updates since the database was last attached; returned as a long integer

TABLE 11.4 TIBDatabaseInfo database operation count properties

Requesting database information

This section gives an example on how to use the *TIBDatabaseInfo* component.

To set up a simple *TIBDatabaseInfo* component:

1. Drop a *TIBDatabase* component and a *TIBDatabaseInfo* component on a Delphi form.
2. Using either the Object Inspector or the Database Component Editor, set up the database connection. For more information, see **"Connecting to a database server" on page 160**.
3. Set the *TIBDatabaseInfo* component's *Database* property to the name of the *TIBDatabase* component.
4. Connect the *TIBDatabase* component to the database by setting the *Connected* property to *True*.
5. Drop a *Button* component and a *Memo* component on the form.
6. Double-click the *Button* component to bring up the code editor, and set any of the *TIBDatabaseInfo* properties described above. For example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
 with IBDatabaseInfo1 do
 begin
   for I := 0 to UserNames.Count - 1 do
   Memo1.Lines.Add(UserNames[i]);
   Memo1.Lines.Add(DBFileName);
   Memo1.Lines.Add(IntToStr(Fetches));
   Memo1.Lines.Add(IntToStr(CurrentMemory));
  end;
end;
```

CHAPTER

12

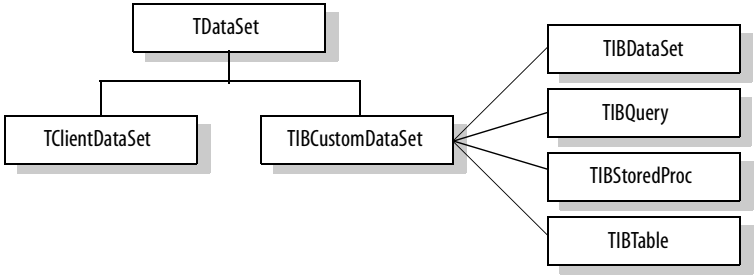
Understanding Datasets

In Delphi, the fundamental unit for accessing data is the dataset family of objects. Your application uses datasets for all database access. Generally, a dataset object represents a specific table belonging to a database, or it represents a query or stored procedure that accesses a database.

All dataset objects that you will use in your database applications descend from the virtualized dataset object, *TDataSet*, and they inherit data fields, properties, events, and methods from *TDataSet*. This chapter describes the functionality of *TDataSet* that is inherited by the dataset objects you will use in your database applications. You need to understand this shared functionality to use any dataset object.

FIGURE 12.1 illustrates the hierarchical relationship of all the dataset components:

FIGURE 12.1 InterBase database component dataset hierarchy



What is TDataSet?

TDataSet is the ancestor for all dataset objects you use in your applications. It defines a set of data fields, properties, events, and methods shared by all dataset objects. *TDataSet* is a virtualized dataset, meaning that many of its properties and methods are *virtual* or *abstract*. A *virtual method* is a function or procedure declaration where the implementation of that method can be (and usually is) overridden in descendant objects. An *abstract method* is a function or procedure declaration without an actual implementation. The declaration is a prototype that describes the method (and its parameters and return type, if any) that must be implemented in all descendant dataset objects, but that might be implemented differently by each of them.

Because *TDataSet* contains *abstract* methods, you cannot use it directly in an application without generating a runtime error. Instead, you either create instances of *TDataSet*'s descendants, such as *TIBCustomDataSet*, *TIBDataSet*, *TIBTable*, *TIBQuery*, *TIBStoredProc*, and *TClientDataSet*, and use them in your application, or you derive your own dataset object from *TDataSet* or its descendants and write implementations for all its *abstract* methods.

Nevertheless, *TDataSet* defines much that is common to all dataset objects. For example, *TDataSet* defines the basic structure of all datasets: an array of *TField* components that correspond to actual columns in one or more database tables, lookup fields provided by your application, or calculated fields provided by your application. For more information about *TField* components, see "Working with field components" in the *Delphi 5 Developer's Guide*.

The following topics are discussed in this chapter:

- **Opening and closing datasets**
- **Determining and setting dataset states**
- **Navigating datasets**
- **Searching datasets**
- **Modifying data**
- **Using dataset events**

Opening and closing datasets

To read or write data in a table or through a query, an application must first open a dataset. You can open a dataset in two ways:

- Set the *Active* property of the dataset to *True*, either at design time in the Object Inspector, or in code at runtime:

```
IBTable.Active := True;
```

- Call the *Open* method for the dataset at runtime:

```
IBQuery.Open;
```

You can close a dataset in two ways:

- Set the *Active* property of the dataset to *False*, either at design time in the Object Inspector, or in code at runtime:

```
IBQuery.Active := False;
```

- Call the *Close* method for the dataset at runtime:

```
IBTable.Close;
```

You may need to close a dataset when you want to change certain of its properties, such as *TableName* on a *TIBTable* component. At runtime, you may also want to close a dataset for other reasons specific to your application.

Determining and setting dataset states

The *state*—or *mode*—of a dataset determines what can be done to its data. For example, when a dataset is closed, its state is *dsInactive*, meaning that nothing can be done to its data. At runtime, you can examine a dataset's read-only *State* property to determine its current state. The following table summarizes possible values for the *State* property and what they mean:

Value	State	Meaning
dsInactive	Inactive	DataSet closed; its data is unavailable
dsBrowse	Browse	DataSet open; its data can be viewed, but not changed
dsEdit	Edit	DataSet open; the current row can be modified
dsInsert	Insert	DataSet open; a new row is inserted or appended

TABLE 12.1 Values for the dataset *State* property

Value	State	Meaning
dsCalcFields	CalcFields	DataSet open; indicates that an *OnCalcFields* event is under way and prevents changes to fields that are not calculated
dsCurValue	CurValue	Internal use only
dsNewValue	NewValue	Internal use only
dsOldValue	OldValue	Internal use only
dsFilter	Filter	DataSet open; indicates that a filter operation is under way: a restricted set of data can be viewed, and no data can be changed

TABLE 12.1 Values for the dataset *State* property

When an application opens a dataset, it appears by default in *dsBrowse* mode. The state of a dataset changes as an application processes data. An open dataset changes from one state to another based on either the code in your application, or the built-in behavior of data-related components.

To put a dataset into *dsBrowse*, *dsEdit*, or *dsInsert* states, call the method corresponding to the name of the state. For example, the following code puts *IBTable* into *dsInsert* state, accepts user input for a new record, and writes the new record to the database:

```
IBTable.Insert; { Your application explicitly sets dataset state to
Insert }
AddressPromptDialog.ShowModal;
if AddressPromptDialog.ModalResult := mrOK then
  IBTable.Post; { Delphi sets dataset state to Browse on successful
completion }
else
  IBTable.Cancel; {Delphi sets dataset state to Browse on cancel }
```

This example also illustrates that the state of a dataset automatically changes to *dsBrowse* when

- The *Post* method successfully writes a record to the database. (If *Post* fails, the dataset state remains unchanged.)
- The *Cancel* method is called.

Some states cannot be set directly. For example, to put a dataset into *dsInactive* state, set its *Active* property to *False*, or call the *Close* method for the dataset. The following statements are equivalent:

```
IBTable.Active := False;
IBTable.Close;
```

The remaining states (*dsCalcFields*, *dsCurValue, dsNewValue*, *dsOldValue*, and *dsFilter*) cannot be set by your application. Instead, the state of the dataset changes automatically to these values as necessary. For example, *dsCalcFields* is set when a dataset's *OnCalcFields* event is called. When the *OnCalcFields* event finishes, the dataset is restored to its previous state.

Whenever a dataset's state changes, the *OnStateChange* event is called for any data source components associated with the dataset. For more information about data source components and *OnStateChange*, see "Using data sources" in the "Using data controls" chapter of the *Delphi 5 Developer's Guide.*

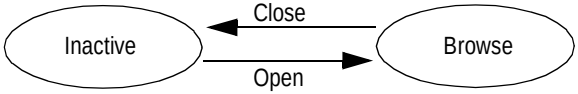
The following sections provide overviews of each state, how and when states are set, how states relate to one another, and where to go for related information, if applicable.

Deactivating a dataset

A dataset is inactive when it is closed. You cannot access records in a closed dataset. At design time, a dataset is closed until you set its *Active* property to *True*. At runtime, a dataset is initially closed until an application opens it by calling the *Open* method, or by setting the *Active* property to *True*.

When you open an inactive dataset, its state automatically changes to the *dsBrowse* state. **FIGURE 12.2** illustrates the relationship between these states and the methods that set them.

FIGURE 12.2 Relationship of Inactive and Browse states



To make a dataset inactive, call its *Close* method. You can write *BeforeClose* and *AfterClose* event handlers that respond to the *Close* method for a dataset. For example, if a dataset is in *dsEdit* or *dsInsert* modes when an application calls *Close*, you should prompt the user to post pending changes or cancel them before closing the dataset. The following code illustrates such a handler:

```
procedure IBTable.VerifyBeforeClose(DataSet: TIBCustomDataSet)
begin
  if (IBTable.State = dsEdit) or (IBTable.State = dsInsert) then
  begin
```

```
    if MessageDlg('Post changes before closing?', mtConfirmation,
mbYesNo, 0) = mrYes then
      IBTable.Post;
    else
      IBTable.Cancel;
  end;
end;
```

To associate a procedure with the *BeforeClose* event for a dataset at design time:

1. Select the table in the data module (or form).
2. Click the Events page in the Object Inspector.
3. Enter the name of the procedure for the *BeforeClose* event (or choose it from the drop-down list).

Browsing a dataset

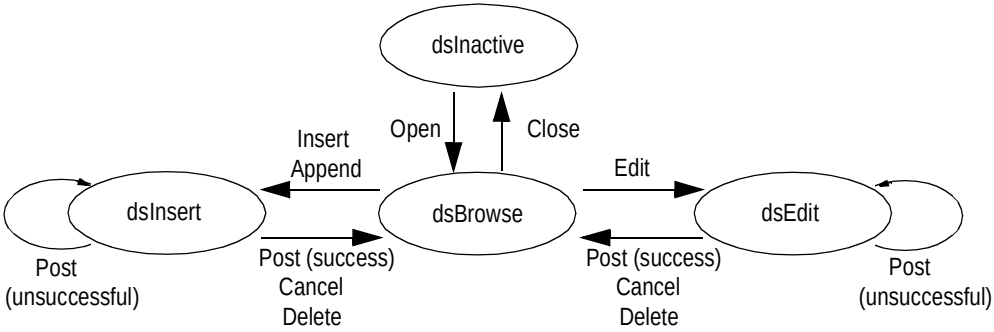
When an application opens a dataset, the dataset automatically enters *dsBrowse* state. Browsing enables you to view records in a dataset, but you cannot edit records or insert new records. You mainly use *dsBrowse* to scroll from record to record in a dataset. For more information about scrolling from record to record, see **"Navigating datasets" on page 175**.

From *dsBrowse* all other dataset states can be set. For example, calling the *Insert* or *Append* methods for a dataset changes its state from *dsBrowse* to *dsInsert* (note that other factors and dataset properties, such as *CanModify*, may prevent this change). For more information about inserting and appending records in a dataset, see **"Modifying data" on page 176**.

Two methods associated with all datasets can return a dataset to *dsBrowse* state. *Cancel* ends the current edit, insert, or search task, and always returns a dataset to *dsBrowse* state. *Post* attempts to write changes to the database, and if successful, also returns a dataset to *dsBrowse* state. If *Post* fails, the current state remains unchanged.

FIGURE 12.3 illustrates the relationship of *dsBrowse* both to the other dataset modes you can set in your applications, and the methods that set those modes.

FIGURE 12.3 Relationship of Browse to other dataset states



Enabling dataset editing

A dataset must be in *dsEdit* mode before an application can modify records. In your code you can use the *Edit* method to put a dataset into *dsEdit* mode if the read-only *CanModify* property for the dataset is *True*. *CanModify* is *True* if the database underlying a dataset permits read and write privileges.

On forms in your application, some data-aware controls can automatically put a dataset into *dsEdit* state if:

- The control's *ReadOnly* property is *False* (the default),
- The *AutoEdit* property of the data source for the control is *True*, and
- *CanModify* is *True* for the dataset.

IMPORTANT For *TIBTable* components, if the *ReadOnly* property is *True*, *CanModify* is *False*, preventing editing of records.

Note Even if a dataset is in *dsEdit* state, editing records will not succeed for InterBase databases if your application user does not have proper SQL access privileges.

You can return a dataset from *dsEdit* state to *dsBrowse* state in code by calling the *Cancel*, *Post,* or *Delete* methods. *Cancel* discards edits to the current field or record. *Post* attempts to write a modified record to the dataset, and if it succeeds, returns the dataset to *dsBrowse*. If *Post* cannot write changes, the dataset remains in *dsEdit* state. *Delete* attempts to remove the current record from the dataset, and if it succeeds, returns the dataset to *dsBrowse* state. If *Delete* fails, the dataset remains in *dsEdit* state.

Data-aware controls for which editing is enabled automatically call *Post* when a user executes any action that changes the current record (such as moving to a different record in a grid) or that causes the control to lose focus (such as moving to a different control on the form).

For a complete discussion of editing fields and records in a dataset, see **"Modifying data" on page 176**.

Enabling insertion of new records

A dataset must be in *dsInsert* mode before an application can add new records. In your code you can use the *Insert* or *Append* methods to put a dataset into *dsInsert* mode if the read-only *CanModify* property for the dataset is *True*. *CanModify* is *True* if the database underlying a dataset permits read and write privileges.

On forms in your application, the data-aware grid and navigator controls can put a dataset into *dsInsert* state if

- The control's *ReadOnly* property is *False* (the default),
- The *AutoEdit* property of the data source for the control is *True*, and
- *CanModify* is *True* for the dataset.

IMPORTANT For *TIBTable* components, if the *ReadOnly* property is *True*, *CanModify* is *False*, preventing editing of records.

Note Even if a dataset is in *dsInsert* state, inserting records will not succeed for InterBase databases if your application user does not have proper SQL access privileges.

You can return a dataset from *dsInsert* state to *dsBrowse* state in code by calling the *Cancel*, *Post*, or *Delete* methods. *Delete* and *Cancel* discard the new record. *Post* attempts to write the new record to the dataset, and if it succeeds, returns the dataset to *dsBrowse*. If *Post* cannot write the record, the dataset remains in *dsInsert* state.

Data-aware controls for which inserting is enabled automatically call *Post* when a user executes any action that changes the current record (such as moving to a different record in a grid).

For more discussion of inserting and appending records in a dataset, see **"Modifying data" on page 176**.

Calculating fields

Delphi puts a dataset into *dsCalcFields* mode whenever an application calls the dataset's *OnCalcFields* event handler. This state prevents modifications or additions to the records in a dataset except for the calculated fields the handler is designed to modify. The reason all other modifications are prevented is because *OnCalcFields* uses the values in other fields to derive values for calculated fields. Changes to those other fields might otherwise invalidate the values assigned to calculated fields.

When the *OnCalcFields* handler finishes, the dataset is returned to *dsBrowse* state.

For more information about creating calculated fields and *OnCalcFields* event handlers, see **"Using OnCalcFields" on page 177**.

Updating records

When performing cached update operations, Delphi may put the dataset into *dsNewValue*, *dsOldValue*, or *dsCurValue* states temporarily. These states indicate that the corresponding properties of a field component (*NewValue*, *OldValue*, and *CurValue*, respectively) are being accessed, usually in an *OnUpdateError* event handler. Your applications cannot see or set these states. For more information about using cached updates, see **Chapter 16, "Working with Cached Updates."**

Navigating datasets

For information on navigating datasets, refer to "Navigating datasets" in the *Delphi 5 Developer's Guide*.

Searching datasets

For information on searching datasets, refer to "Searching datasets" in the *Delphi 5 Developer's Guide*.

Modifying data

For information on modifying data, refer to "Modifying data" in the *Delphi 5 Developer's Guide*.

Using dataset events

Datasets have a number of events that enable an application to perform validation, compute totals, and perform other tasks. The events are listed in the following table:

Event	Description
BeforeOpen, AfterOpen	Called before/after a dataset is opened.
BeforeClose, AfterClose	Called before/after a dataset is closed.
BeforeInsert, AfterInsert	Called before/after a dataset enters Insert state.
BeforeEdit, AfterEdit	Called before/after a dataset enters Edit state.
BeforePost, AfterPost	Called before/after changes to a table are posted.
BeforeCancel, AfterCancel	Called before/after the previous state is canceled.
BeforeDelete, AfterDelete	Called before/after a record is deleted.
OnNewRecord	Called when a new record is created; used to set default values.
OnCalcFields	Called when calculated fields are calculated.

TABLE 12.2 Dataset events

For more information about events for the *TIBCustomDataSet* component, see the online VCL Reference.

Aborting a method

To abort a method such as an *Open* or *Insert*, call the *Abort* procedure in any of the *Before* event handlers (*BeforeOpen*, *BeforeInsert*, and so on). For example, the following code requests a user to confirm a delete operation or else it aborts the call to *Delete*:

```
procedure TForm1.TableBeforeDelete (Dataset: TDataset)
begin
```

```
  if MessageDlg('Delete This Record?', mtConfirmation, mbYesNoCancel,
0) <> mrYes then
    Abort;
end;
```

Using OnCalcFields

The *OnCalcFields* event is used to set the values of calculated fields. The *AutoCalcFields* property determines when *OnCalcFields* is called. If *AutoCalcFields* is *True*, then *OnCalcFields* is called when

- A dataset is opened.
- Focus moves from one visual component to another, or from one column to another in a data-aware grid control and the current record has been modified.
- A record is retrieved from the database.

OnCalcFields is always called whenever a value in a non-calculated field changes, regardless of the setting of *AutoCalcFields*.

IMPORTANT *OnCalcFields* is called frequently, so the code you write for it should be kept short. Also, if *AutoCalcFields* is *True*, *OnCalcFields* should not perform any actions that modify the dataset (or the linked dataset if it is part of a master-detail relationship), because this can lead to recursion. For example, if *OnCalcFields* performs a *Post*, and *AutoCalcFields* is *True*, then *OnCalcFields* is called again, leading to another *Post*, and so on.

If *AutoCalcFields* is *False*, then *OnCalcFields* is not called when individual fields within a single record are modified.

When *OnCalcFields* executes, a dataset is in *dsCalcFields* mode, so you cannot set the values of any fields other than calculated fields. After *OnCalcFields* is completed, the dataset returns to *dsBrowse* state.

Using cached updates

Cached updates enable you to retrieve data from a database, cache and edit it locally, and then apply the cached updates to the database as a unit. When cached updates are enabled, updates to a dataset (such as posting changes or deleting records) are stored in an internal cache instead of being written directly to the dataset's underlying table. When changes are complete, your application calls a method that writes the cached changes to the database and clears the cache.

Implementation of cached updating occurs in *TIBCustomDataSet*. The following table lists the properties, events, and methods for cached updating:

Property, event, or method	Purpose
CachedUpdates property	Determines whether or not cached updates are in effect for the dataset. If *True*, cached updating is enabled. If *False*, cached updating is disabled.
UpdateObject property	Indicates the name of the *TUpdateSQL* component used to update datasets based on queries.
UpdatesPending property	Indicates whether or not the local cache contains updated records that need to be applied to the database. *True* indicates there are records to update. *False* indicates the cache is empty.
UpdateRecordTypes property	Indicates the kind of updated records to make visible to the application during the application of cached updates.
UpdateStatus method	Indicates if a record is unchanged, modified, inserted, or deleted.
OnUpdateError event	A developer-created procedure that handles update errors on a record-by-record basis.
OnUpdateRecord event	A developer-created procedure that processes updates on a record-by-record basis.
ApplyUpdates method	Applies records in the local cache to the database.
CancelUpdates method	Removes all pending updates from the local cache without applying them to the database.
FetchAll method	Copies all database records to the local cache for editing and updating.
RevertRecord method	Undoes updates to the current record if updates are not yet applied on the server side.

TABLE 12.3 Properties, events, and methods for cached updates

Using cached updates and coordinating them with other applications that access data in a multi-user environment is an advanced topic that is fully covered in **Chapter 16, "Working with Cached Updates."**

CHAPTER

13 Working with Tables

This chapter describes how to use the *TIBTable* dataset component in your database applications. A table component encapsulates the full structure of and data in an underlying database table. A table component inherits many of its fundamental properties and methods from *TDataSet* and *TIBCustomDataSet*. Therefore, you should be familiar with the general discussion of datasets in **Chapter 12, "Understanding Datasets"** and before reading about the unique properties and methods of table components discussed here.

Using table components

A table component gives you access to every row and column in an underlying database table. You can view and edit data in every column and row of a table. You can work with a range of rows in a table, and you can filter records to retrieve a subset of all records in a table based on filter criteria you specify. You can search for records, copy, rename, or delete entire tables, and create master/detail relationships between tables.

Note A table component always references a single database table. If you need to access multiple tables with a single component, or if you are only interested in a subset of rows and columns in one or more tables, you should use a *TIBQuery* or *TIBDataSet* component instead of a *TIBTable* component. For more information about *TIBQuery* and *TIBDataSet* components, see **Chapter 14, "Working with Queries."**

Setting up a table component

The following steps are general instructions for setting up a table component at design time. There may be additional steps you need to tailor a table's properties to the requirements of your application.

- To create a table component:

1. Place a table component from the InterBase page of the Component palette in a data module or on a form, and set its *Name* property to a unique value appropriate to your application.
2. Set the *Database* property to the name of the database component to access.
3. Set the *Transaction* property to the name of the transaction component.
4. Set the *DatabaseName* property in the Database component to the name of a the database containing the table.
5. Set the *TableName* property to the name of the table in the database. You can select tables from the drop-down list if the *Database* and *Transaction* properties are already specified, and if the *Database* and *Transaction* components are connected to the server.
6. Place a data source component in the data module or on the form, and set its *DataSet* property to the name of the table component. The data source component is used to pass a result set from the table to data-aware components for display.

- To access the data encapsulated by a table component:

1. Place a data source component from the Data Access page of the Component palette in the data module or form, and set its *DataSet* property to the name of the table component.
2. Place a data-aware control, such as *TDBGrid*, on a form, and set the control's *DataSource* property to the name of the data source component placed in the previous step.
3. Set the *Active* property of the table component to *True*.

Tip For more information about database components, see **Chapter 11, "Connecting to Databases."**

Specifying a table name

The *TableName* property specifies the table in a database to access with the table component. To specify a table, follow these steps:

1. Set the table's *Active* property to *False*, if necessary.
2. Set the *DatabaseName* property of the database component to a directory path.

Note You can use the Database Editor to set the database location, login name, password, SQL role, and switch the login prompt on and off. To access the Database Component Editor, right click on the database component and choose Database Editor from the drop-down menu.

3. Set the *TableName* property to the table to access. You are prompted to log in to the database. At design time you can choose from valid table names in the drop-down list for the *TableName* property in the Object Inspector. At runtime, you must specify a valid name in code.

Once you specify a valid table name, you can set the table component's *Active* property to *True* to connect to the database, open the table, and display and edit data.

At runtime, you can set or change the table associated with a table component by:

- Setting *Active* to *False*.
- Assigning a valid table name to the *TableName* property.

For example, the following code changes the table name for the *OrderOrCustTable* table component based on its current table name:

```
with OrderOrCustTable do
begin
  Active := False; {Close the table}
  if TableName = 'CUSTOMER.DB' then
    TableName := 'ORDERS.DB'
  else
    TableName := 'CUSTOMER.DB';
  Active := True; {Reopen with a new table}
end;
```

Opening and closing a table

To view and edit a table's data in a data-aware control such as *TDBGrid*, open the table. There are two ways to open a table. You can set its *Active* property to *True*, or you can call its *Open* method. Opening a table puts it into *dsBrowse* state and displays data in any active controls associated with the table's data source.

To end display and editing of data, or to change the values for a table component's fundamental properties (for example: *Database*, *TableName*, and *TableType*), first post or discard any pending changes. If cached updates are enabled, call the *ApplyUpdates* method to write the posted changes to the database. Finally, close the table.

There are two ways to close a table. You can set its *Active* property to *False*, or you can call its *Close* method. Closing a table puts the table into *dsInactive* state. Active controls associated with the table's data source are cleared.

Controlling read/write access to a table

By default when a table is opened, it requests read and write access for the underlying database table. Depending on the characteristics of the underlying database table, the requested write privilege may not be granted (for example, when you request write access to an SQL table on a remote server and the server restricts the table's access to read only).

The *ReadOnly* property for table components is the only property that can affect an application's read and write access to a table.

ReadOnly determines whether or not a user can both view and edit data. When *ReadOnly* is *False* (the default), a user can both view and edit data. To restrict a user to viewing data, set *ReadOnly* to *True* before opening a table.

Searching for records

You can search for specific records in a table in various ways. The most flexible and preferred way to search for a record is to use the generic search methods *Locate* and *Lookup*. These methods enable you to search on any type of fields in any table, whether or not they are indexed or keyed.

- *Locate* finds the first row matching a specified set of criteria and moves the cursor to that row.
- *Lookup* returns values from the first row that matches a specified set of criteria, but does not move the cursor to that row.

You can use *Locate* and *Lookup* with any kind of dataset, not just *TIBTable*. For a complete discussion of *Locate* and *Lookup*, see **Chapter 12, "Understanding Datasets."**

Table components also support the *Goto* and *Find* methods. While these methods are documented here to allow you to work with legacy applications, you should always use *Lookup* and *Locate* in your new applications. You may see performance gains in existing applications if you convert them to use the new methods.

Sorting records

An index determines the display order of records in a table. In general, records appear in ascending order based on a primary index. This default behavior does not require application intervention. If you want a different sort order, however, you must specify either

- An alternate index.
- A list of columns on which to sort.

Specifying a different sort order requires the following steps:

1. Determining available indexes.
2. Specifying the alternate index or column list to use.

Retrieving a list of available indexes with GetIndexNames

At runtime, your application can call the *GetIndexNames* method to retrieve a list of available indexes for a table. *GetIndexNames* returns a string list containing valid index names. For example, the following code determines the list of indexes available for the *CustomersTable* dataset:

```
var
  IndexList: TList;
{...}
CustomersTable.GetIndexNames(IndexList);
```

Specifying an alternative index with IndexName

To specify that a table should be sorted using an alternative index, specify the index name in the table component's *IndexName* property. At design time you can specify this name in the Object Inspector, and at runtime you can access the property in your code. For example, the following code sets the index for *CustomersTable* to *CustDescending*:

```
CustomersTable.IndexName := 'CustDescending';
```

Specifying sort order for SQL tables

In SQL, sort order of rows is determined by the ORDER BY clause. You can specify the index used by this clause either with the

- *IndexName* property, to specify an existing index, or
- *IndexFieldNames* property, to create a pseudo-index based on a subset of columns in the table.

IndexName and *IndexFieldNames* are mutually exclusive. Setting one property clears values set for the other.

Specifying fields with IndexFieldNames

IndexFieldNames is a string list property. To specify a sort order, list each column name to use in the order it should be used, and delimit the names with semicolons. Sorting is by ascending order only.

The following code sets the sort order for *PhoneTable* based on *LastName*, then *FirstName*:

```
PhoneTable.IndexFieldNames := 'LastName;FirstName';
```

Examining the field list for an index

When your application uses an index at runtime, it can examine the

- *IndexFieldCount* property, to determine the number of columns in the index.
- *IndexFields* property, to examine a list of column names that comprise the index.

IndexFields is a string list containing the column names for the index. The following code fragment illustrates how you might use *IndexFieldCount* and *IndexFields* to iterate through a list of column names in an application:

```
var
  I: Integer;
  ListOfIndexFields: array[0 to 20} of string;
begin
with CustomersTable do
  begin
  for I := 0 to IndexFieldCount - 1 do
     ListOfIndexFields[I] := IndexFields[I];
  end;
end;
```

Note *IndexFieldCount* is not valid for a base table opened on an expression index.

Working with a subset of data

Production tables can be huge, so applications often need to limit the number of rows with which they work. For table components use filters to limit records used by an application. Filters can be used with any kind of dataset, including *TIBDataSet*, *TIBTable*, *TIBQuery*, and *TIBStoredProc* components. Because they apply to all datasets, you can find a full discussion of using filters in **Chapter 12, "Understanding Datasets."**

Deleting all records in a table

To delete all rows of data in a table, call a table component's *EmptyTable* method at runtime. For SQL tables, this method only succeeds if you have DELETE privileges for the table. For example, the following statement deletes all records in a dataset:

```
PhoneTable.EmptyTable;
```

IMPORTANT Data you delete with *EmptyTable* is gone forever.

Deleting a table

At design time, to delete a table from a database, right-click the table component and select Delete Table from the context menu. The Delete Table menu pick will only be present if the table component represents an existing database table (the *Database* and *TableName* properties specify an existing table).

To delete a table at runtime, call the table component's *DeleteTable* method. For example, the following statement removes the table underlying a dataset:

```
CustomersTable.DeleteTable;
```

IMPORTANT When you delete a table with *DeleteTable*, the table and all its data are gone forever.

Renaming a table

You can rename a table by typing over the name of an existing table next to the *TableName* property in the Object Inspector. When you change the *TableName* property, a dialog appears asking you if you want to rename the table. At this point, you can either choose to rename the table, or you can cancel the operation, changing the *TableName* property (for example, to create a new table) without changing the name of the table represented by the old value of *TableName*.

Creating a table

You can create new database tables at design time or at runtime. The Create Table command (at design time) or the *CreateTable* method (at runtime) provides a way to create tables without requiring SQL knowledge. They do, however, require you to be intimately familiar with the properties, events, and methods common to dataset components, *TIBTable* in particular. This is so that you can first define the table you want to create by doing the following:

- Set the *Database* property to the database that will contain the new table.
- Set the *TableName* property to the name of the new table.

- Add field definitions to describe the fields in the new table. At design time, you can add the field definitions by double-clicking the *FieldDefs* property in the Object Inspector to bring up the collection editor. Use the collection editor to add, remove, or change the properties of the field definitions. At runtime, clear any existing field definitions and then use the *AddFieldDef* method to add each new field definition. For each new field definition, set the properties of the *TFieldDef* object to specify the desired attributes of the field.
- Optionally, add index definitions that describe the desired indexes of the new table. At design time, you can add index definitions by double-clicking the *IndexDefs* property in the Object Inspector to bring up the collection editor. Use the collection editor to add, remove, or change the properties of the index definitions. At runtime, clear any existing index definitions, and then use the *AddIndexDef* method to add each new index definition. For each new index definition, set the properties of the *TIndexDef* object to specify the desired attributes of the index.

  Note At design time, you can preload the field definitions and index definitions of an existing table into the *FieldDefs* and *IndexDefs* properties, respectively. Set the *Database* and *TableName* properties to specify the existing table. Right click the table component and choose Update Table Definition. This automatically sets the values of the *FieldDefs* and *IndexDefs* properties to describe the fields and indexes of the existing table. Next, reset the *Database* and *TableName* to specify the table you want to create, cancelling any prompts to rename the existing table. If you want to store these definitions with the table component (for example, if your application will be using them to create tables on user's systems), set the *StoreDefs* property to *True*.

  Once the table is fully described, you are ready to create it. At design time, right-click the table component and choose Create Table. At runtime, call the *CreateTable* method to generate the specified table.

IMPORTANT If you create a table that duplicates the name of an existing table, the existing table and all its data are overwritten by the newly created table. The old table and its data cannot be recovered.

The following code creates a new table at runtime and associates it with the EMPLOYEE.GDB database. Before it creates the new table, it verifies that the table name provided does not match the name of an existing table:

```
var
  NewTable: TIBTable;
  NewIndexOptions: TIndexOptions;
  TableFound: Boolean;
begin
  NewTable := TIBTable.Create;
  NewIndexOptions := [ixPrimary, ixUnique];
```

```
  with NewTable do
  begin
    Active := False;
    Database := 'C:\Interbase\Examples\Database\employee.gdb';
    TableName := Edit1.Text;
    TableType := ttDefault;
    FieldDefs.Clear;
    FieldDefs.Add(Edit2.Text, ftInteger, 0, False);
    FieldDefs.Add(Edit3.Text, ftInteger, 0, False);
    IndexDefs.Clear;
    IndexDefs.Add('PrimaryIndex', Edit2.Text, NewIndexOptions);
  end;
  {Now check for prior existence of this table}
  TableFound := FindTable(Edit1.Text); {code for FindTable not shown}
  if TableFound = True then
   if MessageDlg('Overwrite existing table ' + Edit1.Text + '?',
mtConfirmation,
      mbYesNo, 0) = mrYes then
    TableFound := False;
  if not TableFound then
    CreateTable; { create the table}
  end;
end;
```

Synchronizing tables linked to the same database table

If more than one table component is linked to the same database table through their *Database* and *TableName* properties and the tables do not share a data source component, then each table has its own view on the data and its own current record. As users access records through each table component, the components' current records will differ.

You can force the current record for each of these table components to be the same with the *GotoCurrent* method. *GotoCurrent* sets its own table's current record to the current record of another table component. For example, the following code sets the current record of *CustomerTableOne* to be the same as the current record of *CustomerTableTwo*:

```
CustomerTableOne.GotoCurrent(CustomerTableTwo);
```

Tip If your application needs to synchronize table components in this manner, put the components in a data module and include the header for the data module in each unit that accesses the tables.

If you must synchronize table components on separate forms, you must include one form's header file in the source unit of the other form, and you must qualify at least one of the table names with its form name.

For example:

```
CustomerTableOne.GotoCurrent(Form2.CustomerTableTwo);
```

Creating master/detail forms

A table component's *MasterSource* and *MasterFields* properties can be used to establish one-to-many relationships between two tables.

The *MasterSource* property is used to specify a data source from which the table will get data for the master table. For instance, if you link two tables in a master/detail relationship, then the detail table can track the events occurring in the master table by specifying the master table's data source component in this property.

The *MasterFields* property specifies the column(s) common to both tables used to establish the link. To link tables based on multiple column names, use a semicolon delimited list:

```
Table1.MasterFields := 'OrderNo;ItemNo';
```

To help create meaningful links between two tables, you can use the Field Link designer. For more information about the Field Link designer, see the online *Delphi User's Guide*.

Building an example master/detail form

The following steps create a simple form in which a user can scroll through customer records and display all orders for the current customer. The master table is the *CustomersTable* table, and the detail table is *SalesTable*.

1. Place two *TIBTable*, *TIBDatabase*, *TIBTransaction*, and *TDataSource* components in a data module.
2. Set the properties of the first *TIBDatabase* component as follows:

· *Name*: *CustDatabase*

· *DefaultTransaction*: *CustTransaction*

3. Using the Database Component Editor, set the Database Name to **C:\Program Files\Interbase Corp\Interbase\examples\Database\employee.gdb**, set the User Name to SYSDBA, set the Password to "masterkey", and uncheck the Login Prompt box.

4. Set the properties of the first *TIBTransaction* component as follows:

· *DefaultDatabase*: *CustDatabase*

· *Name*: *CustTransaction*

5. Set the properties of the first *TIBTable* component as follows:

· *Database*: *CustDatabase*

· *Transaction*: *CustTransaction*

· *TableName*: *CUSTOMER*

· *Name*: *CustomersTable*

6. Set the properties of the second *TIBDatabase* component as follows:

· *Name*: *SalesDatabase*

· *DefaultTransaction*: *SalesTransaction*

7. Using the Database Component Editor, set the Database Name to **C:\Program Files\Interbase Corp\Interbase\examples\Database\employee.gdb**, set the User Name to SYSDBA, set the Password to "masterkey", and uncheck the Login Prompt box.

8. Set the properties of the second *TIBTransaction* component as follows:

· *DefaultDatabase*: *SalesDatabase*

· *Name*: *SalesTransaction*

9. Set the properties of the second *TIBTable* component as follows:

· *Database*: *SalesDatabase*

· *Transaction*: *SalesTransaction*

· *TableName*: *SALES*

· *Name*: *SalesTable*

10. Set the properties of the first *TDataSource* component as follows:

· *Name*: *CustSource*

· *DataSet*: *CustomersTable*

11. Set the properties of the second *TDataSource* component as follows:

· *Name*: *SalesSource*

· *DataSet*: *SalesTable*

12. Place two *TDBGrid* components on a form.

13. Choose **File | Use Unit** to specify that the form should use the data module.
14. Set the *DataSource* property of the first grid component to "DataModule2.CustSource", and set the *DataSource* property of the second grid to "DataModule2.SalesSource".
15. Set the *MasterSource* property of *SalesTable* to "CustSource". This links the CUSTOMER table (the master table) to the ORDERS table (the detail table).
16. Double-click the *MasterFields* property value box in the Object Inspector to invoke the Field Link Designer to set the following properties:

- Choose *CustNo* from the IndexFieldNames property's drop-down list to link the two tables by the *CustNo* field.
- Select *CustNo* in both the Detail Fields and Master Fields field lists.
- Click the Add button to add this join condition. In the Joined Fields list, "CustNo -> CustNo" appears.
- Choose OK to commit your selections and exit the Field Link Designer.
- Set the *Active* properties of *CustomersTable* and *SalesTable* to *True* to display data in the grids on the form.
- Compile and run the application.

If you run the application now, you will see that the tables are linked together, and that when you move to a new record in the CUSTOMER table, you see only those records in the SALES table that belong to the current customer.

CHAPTER

14

Working with Queries

This chapter describes the *TIBDataSet* and *TIBQuery* dataset components which enable you to use SQL statements to access data. It assumes you are familiar with the general discussion of datasets and data sources in **Chapter 12, "Understanding Datasets."**

A query component encapsulates an SQL statement that is used in a client application to retrieve, insert, update, and delete data from one or more database tables. Query components can be used with remote database servers (Client/Server Suite and Enterprise edition only) and with ODBC-compliant databases.

Queries for desktop developers

As a desktop developer you are already familiar with the basic table, record, and field paradigm used by Delphi and InterBase Express. You feel very comfortable using a *TIBTable* component to gain access to every field in every data record in a dataset. You know that when you set a table's *TableName* property, you specify the database table to access.

Chances are you have also used a *TIBTable*'s range methods and filter property to limit the number of records available at any given time in your applications. Applying a range temporarily limits data access to a block of contiguously indexed records that fall within prescribed boundary conditions, such as returning all records for employees whose last names are greater than or equal to "Jones" and less than or equal to "Smith." Setting a filter temporarily restricts data access to a set of records that is usually non-contiguous and that meets filter criteria, such as returning only those customer records that have a California mailing address.

A query behaves in many ways very much like a table filter, except that you use the query component's *SQL* property (and sometimes the *Params* property) to identify the records in a dataset to retrieve, insert, delete, or update. In some ways a query is even more powerful than a filter because it lets you access:

- More than one table at a time (called a "join" in SQL).
- A specified subset of rows *and* columns in its underlying table(s), rather than always returning all rows and columns. This improves both performance and security. Memory is not wasted on unnecessary data, and you can prevent access to fields a user should not view or modify.

Queries can be verbatim, or they can contain replaceable parameters. Queries that use parameters are called *parameterized queries*. When you use parameterized queries, the actual values assigned to the parameters are inserted into the query before you execute, or run, the query. Using parameterized queries is very flexible, because you can change a user's view of and access to data on the fly at runtime without having to alter the SQL statement.

Most often you use queries to select the data that a user should see in your application, just as you do when you use a table component. Queries, however, can also perform update, insert, and delete operations instead of retrieving records for display. When you use a query to perform insert, update, and delete operations, the query ordinarily does not return records for viewing. In this way a query differs from a table.

To learn more about using the SQL property to write an SQL statement, see **"Specifying the SQL statement to execute" on page 197**. To learn more about using parameters in your SQL statements, see **"Setting parameters" on page 200**. To learn about executing a query, see **"Executing a query" on page 205** and the *InterBase 6 Language Reference*.

Queries for server developers

As a server developer you are already familiar with SQL and with the capabilities of your database server. To you a query is the SQL statement you use to access data. You know how to use and manipulate this statement and how to use optional parameters with it.

The SQL statement and its parameters are the most important parts of a query component. The query component's *SQL* property is used to provide the SQL statement to use for data access, and the component's *Params* property is an optional array of parameters to bind into the query. However, a query component is much more than an SQL statement and its parameters. A query component is also the interface between your client application and the server.

A client application uses the properties and methods of a query component to manipulate an SQL statement and its parameters, to specify the database to query, to prepare and unprepare queries with parameters, and to execute the query. A query component's methods communicates with the database server, usually through an SQL Links driver for Delphi Client/Server and Delphi Enterprise.

To learn more about using the *SQL* property to write an SQL statement, see **"Specifying the SQL statement to execute" on page 197**. To learn more about using parameters in your SQL statements, see **"Setting parameters" on page 200**. To learn about preparing a query, see **"Preparing a query" on page 207**, and to learn more about executing a query, see **"Executing a query" on page 205**.

When to use TIBDataSet, TIBQuery, and TIBSQL

Both *TIBDataSet*, *TIBQuery,* and *TIBSQL* can execute any valid dynamic SQL statement. However, when you use *TIBSQL* to execute SELECT statements, its results are unbuffered and therefore uni-directional. *TIBDataSet* and *TIBQuery*, on the other hand, are intended primarily for use with SELECT statements. They buffer the result set, so that it is completely scrollable.

Use *TIBDataSet* or *TIBQuery* when you require use of data-aware components or a scrollable result set. In any other case, it is probably best to use *TIBSQL*, which requires much less overhead.

Using a query component: an overview

To use a query component in an application, follow these steps at design time:

1. Place a query component from the InterBase tab of the Component palette in a data module, and set its *Name* property appropriately for your application.
2. Set the *Database* property of the component to the name of the *TIBDatabase* component to query.

3. Set the *Transaction* property of the component to the name of the *TIBTransaction* component to query.
4. Specify an SQL statement in the *SQL* property of the component, and optionally specify any parameters for the statement in the *Params* property. For more information, see **"Specifying the SQL property at design time" on page 198**.
5. If the query data is to be used with visual data controls, place a data source component from the Data Access tab of the Component palette in the data module, and set its *DataSet* property to the name of the query component. The data source component is used to return the results of the query (called a *result set*) from the query to data-aware components for display. Connect data-aware components to the data source using their *DataSource* and *DataField* properties.
6. Activate the query component. For queries that return a result set, use the *Active* property or the *Open* method. For queries that only perform an action on a table and return no result set, use the *ExecSQL* method.

To execute a query for the first time at runtime, follow these steps:

1. Close the query component.
2. Provide an SQL statement in the *SQL* property if you did not set the *SQL* property at design time, or if you want to change the SQL statement already provided. To use the design-time statement as is, skip this step. For more information about setting the *SQL* property, see **"Specifying the SQL statement to execute" on page 197**.
3. Set parameters and parameter values in the *Params* property either directly or by using the *ParamByName* method. If a query does not contain parameters, or the parameters set at design time are unchanged, skip this step. For more information about setting parameters, see **"Setting parameters" on page 200**.
4. Call *Prepare* to bind parameter values into the query. Calling *Prepare* is optional, though highly recommended. For more information about preparing a query, see **"Preparing a query" on page 207**.
5. Call *Open* for queries that return a result set, or call *ExecSQL* for queries that do not return a result set. For more information about opening and executing a query see **"Executing a query" on page 205**.

After you execute a query for the first time, then as long as you do not modify the SQL statement, an application can repeatedly close and reopen or re-execute a query without preparing it again. For more information about reusing a query, see **"Executing a query" on page 205**.

Specifying the SQL statement to execute

Use the *SQL* property to specify the SQL query statement to execute. At design time a query is prepared and executed automatically when you set the query component's *Active* property to *True*. At runtime, a query is prepared with a call to *Prepare*, and executed when the application calls the component's *Open* or *ExecSQL* methods.

The *SQL* property is a *TStrings* object, which is an array of text strings and a set of properties, events, and methods that manipulate them. The strings in *SQL* are automatically concatenated to produce the SQL statement to execute. You can provide a statement in as few or as many separate strings as you desire. One advantage to using a series of strings is that you can divide the SQL statement into logical units (for example, putting the WHERE clause for a SELECT statement into its own string), so that it is easier to modify and debug a query.

The SQL statement can be a query that contains hard-coded field names and values, or it can be a parameterized query that contains replaceable parameters that represent field values that must be bound into the statement before it is executed. For example, this statement is hard-coded:

```
SELECT * FROM Customer WHERE CustNo = 1231
```

Hard-coded statements are useful when applications execute exact, known queries each time they run. At design time or runtime you can easily replace one hard-code query with another hard-coded or parameterized query as needed. Whenever the SQL property is changed the query is automatically closed and unprepared.

Note In queries using local SQL, when column names in a query contain spaces or special characters, the column name must be enclosed in quotes and must be preceded by a table reference and a period. For example, BIOLIFE."Species Name".

A parameterized query contains one or more placeholder parameters, application variables that stand in for comparison values such as those found in the WHERE clause of a SELECT statement. Using parameterized queries enables you to change the value without rewriting the application. Parameter values must be bound into the SQL statement before it is executed for the first time. Query components do this automatically for you even if you do not explicitly call the *Prepare* method before executing a query.

This statement is a parameterized query:

```
SELECT * FROM Customer WHERE CustNo = :Number
```

The variable *Number*, indicated by the leading colon, is a parameter that fills in for a comparison value that must be provided at runtime and that may vary each time the statement is executed. The actual value for *Number* is provided in the query component's *Params* property.

Tip It is a good programming practice to provide variable names for parameters that correspond to the actual name of the column with which it is associated. For example, if a column name is "Number," then its corresponding parameter would be ":Number". Using matching names ensures that if a query uses its *DataSource* property to provide values for parameters, it can match the variable name to valid field names.

Specifying the SQL property at design time

You can specify the *SQL* property at design time using the String List editor. To invoke the String List editor for the *SQL* property:

- Double-click on the *SQL* property value column, or
- Click its ellipsis button.

You can enter an SQL statement in as many or as few lines as you want. Entering a statement on multiple lines, however, makes it easier to read, change, and debug. Choose OK to assign the text you enter to the *SQL* property.

Normally, the *SQL* property can contain only one complete SQL statement at a time, although these statements can be as complex as necessary (for example, a SELECT statement with a WHERE clause that uses several nested logical operators such as AND and OR). InterBase supports "batch" syntax so you can enter multiple statements in the *SQL* property.

Note With the Client/Server Suite or Enterprise edition, you can also use the SQL Builder to construct a query based on a visible representation of tables and fields in a database. To use the SQL Builder, select a query component, right-click it to invoke the context menu, and choose Graphical Query Editor. To learn how to use the SQL Builder, open it and use its online help.

Specifying an SQL statement at runtime

There are three ways to set the *SQL* property at runtime. An application can set the *SQL* property directly, it can call the *SQL* property's *LoadFromFile* method to read an SQL statement from a file, or an SQL statement in a string list object can be assigned to the *SQL* property.

Setting the SQL property directly

To directly set the *SQL* property at runtime,

1. Call *Close* to deactivate the query. Even though an attempt to modify the *SQL* property automatically deactivates the query, it is a good safety measure to do so explicitly.
2. If you are replacing the whole SQL statement, call the *Clear* method for the *SQL* property to delete its current SQL statement.
3. If you are building the whole SQL statement from nothing or adding a line to an existing statement, call the *Add* method for the *SQL* property to insert and append one or more strings to the *SQL* property to create a new SQL statement. If you are modifying an existing line use the *SQL* property with an index to indicate the line affected, and assign the new value.
4. Call *Open* or *ExecSQL* to execute the query.

The following code illustrates building an entire SQL statement from nothing.

```
with CustomerQuery do begin
  Close;                      { close the query if it's active }
  with SQL do begin
    Clear;                { delete the current SQL statement, if any }
    Add('SELECT * FROM Customer');        { add first line of SQL... }
    Add('WHERE Company = "Sight Diver"');      { ... and second line }
  end;
  Open;                                          { activate the query }
end;
```

The code below demonstrates modifying only a single line in an existing SQL statement. In this case, the WHERE clause already exists on the second line of the statement. It is referenced via the *SQL* property using an index of 1.

```
CustomerQuery.SQL[1] := 'WHERE Company = "Kauai Dive Shoppe"';
```

Note If a query uses parameters, you should also set their initial values and call the *Prepare* method before opening or executing a query. Explicitly calling *Prepare* is most useful if the same SQL statement is used repeatedly; otherwise it is called automatically by the query component.

Loading the SQL property from a file

You can also use the *LoadFromFile* method to assign an SQL statement in a text file to the *SQL* property. The *LoadFromFile* method automatically clears the current contents of the *SQL* property before loading the new statement from file. For example:

```
CustomerQuery.Close;
CustomerQuery.SQL.LoadFromFile('c:\orders.txt');
CustomerQuery.Open;
```

Note If the SQL statement contained in the file is a parameterized query, set the initial values for the parameters and call *Prepare* before opening or executing the query. Explicitly calling *Prepare* is most useful if the same SQL statement is used repeatedly; otherwise it is called automatically by the query component.

Loading the SQL property from string list object

You can also use the *Assign* method of the *SQL* property to copy the contents of a string list object into the *SQL* property. The *Assign* method automatically clears the current contents of the *SQL* property before copying the new statement. For example, copying an SQL statement from a *TMemo* component:

```
CustomerQuery.Close;
CustomerQuery.SQL.Assign(Memo1.Lines);
CustomerQuery.Open;
```

Note If the SQL statement is a parameterized query, set the initial values for the parameters and call *Prepare* before opening or executing the query. Explicitly calling *Prepare* is most useful if the same SQL statement is used repeatedly; otherwise it is called automatically by the query component.

Setting parameters

A parameterized SQL statement contains parameters, or variables, the values of which can be varied at design time or runtime. Parameters can replace data values, such as those used in a WHERE clause for comparisons, that appear in an SQL statement. Ordinarily, parameters stand in for data values passed to the statement. For example, in the following INSERT statement, values to insert are passed as parameters:

```
INSERT INTO Country (Name, Capital, Population)
VALUES (:Name, :Capital, :Population)
```

In this SQL statement, *:name*, *:capital*, and *:population* are placeholders for actual values supplied to the statement at runtime by your application. Before a parameterized query is executed for the first time, your application should call the *Prepare* method to bind the current values for the parameters to the SQL statement. Binding means that the server allocates resources for the statement and its parameters that improve the execution speed of the query.

```
with IBQuery1 do begin
  Close;
  Unprepare;
  ParamByName('Name').AsString := 'Belize';
  ParamByName('Capital').AsString := 'Belmopan';
  ParamByName('Population').AsInteger := '240000';
  Prepare;
  Open;
end;
```

Supplying parameters at design time

At design time, parameters in the SQL statement appear in the parameter collection editor. To access the *TParam* objects for the parameters, invoke the parameter collection editor, select a parameter, and access the *TParam* properties in the Object Inspector. If the SQL statement does not contain any parameters, no *TParam* objects are listed in the collection editor. You can only add parameters by writing them in the SQL statement.

To access parameters:

1. Select the query component.
2. Click on the ellipsis button for the *Params* property in Object Inspector.
3. In the parameter collection editor, select a parameter.
4. The *TParam* object for the selected parameter appears in the Object Inspector.
5. Inspect and modify the properties for the *TParam* in the Object Inspector.

For queries that do not already contain parameters (the *SQL* property is empty or the existing SQL statement has no parameters), the list of parameters in the collection editor dialog is empty. If parameters are already defined for a query, then the parameter editor lists all existing parameters.

Note The *TIBQuery* component shares the *TParam* object and its collection editor with a number of different components. While the right-click context menu of the collection editor always contains the Add and Delete options, they are never enabled for *TIBQuery* parameters. The only way to add or delete *TIBQuery* parameters is in the SQL statement itself.

As each parameter in the collection editor is selected, the Object Inspector displays the properties and events for that parameter. Set the values for parameter properties and methods in the Object Inspector.

The *DataType* property lists the data type for the parameter selected in the editing dialog. Initially the type will be *ftUnknown*. You must set a data type for each parameter.

The *ParamType* property lists the type of parameter selected in the editing dialog. Initially the type will be *ptUnknown*. You must set a type for each parameter.

Use the *Value* property to specify a value for the selected parameter at design-time. This is not mandatory when parameter values are supplied at runtime. In these cases, leave *Value* blank.

Supplying parameters at runtime

To create parameters at runtime, you can use the:

- *ParamByName* method to assign values to a parameter based on its name.
- *Params* property to assign values to a parameter based on the parameter's ordinal position within the SQL statement.
- *Params.ParamValues* property to assign values to one or more parameters in a single command line, based on the name of each parameter set. This method uses variants and avoids the need to cast values.

Note In dialect 3, parameter names passed to functions are case-sensitive.

For all of the examples below, assume the *SQL* property contains the SQL statement below. All three parameters used are of data type *ftString*.

```
INSERT INTO "COUNTRY.DB"
(Name, Capital, Continent)
VALUES (:Name, :Capital, :Continent)
```

The following code uses *ParamByName* to assign the text of an edit box to the Capital parameter:

```
IBQuery1.ParamByName('Capital').AsString := Edit1.Text;
```

The same code can be rewritten using the *Params* property, using an index of 1 (the Capital parameter is the second parameter in the SQL statement):

```
IBQuery1.Params[1].AsString := Edit1.Text;
```

The command line below sets all three parameters at once, using the *Params.ParamValues* property:

```
IBQuery1.Params.ParamValues['Country;Capital;Continent'] :=
  VarArrayOf([Edit1.Text, Edit2.Text, Edit3.Text]);
```

Using a data source to bind parameters

If parameter values for a parameterized query are not bound at design time or specified at runtime, the query component attempts to supply values for them based on its *DataSource* property. *DataSource* specifies a different table or query component that the query component can search for field names that match the names of unbound parameters. This search dataset must be created and populated before you create the query component that uses it. If matches are found in the search dataset, the query component binds the parameter values to the values of the fields in the current record pointed to by the data source.

You can create a simple application to understand how to use the *DataSource* property to link a query in a master-detail form. Suppose the data module for this application is called *LinkModule*, and that it contains a query component called *SalesQuery* that has the following *SQL* property:

```
SELECT Cust_No, Po_Number, Order_Date
FROM Sales
WHERE Cust_No = :Cust_No
```

The *LinkModule* data module also contains:

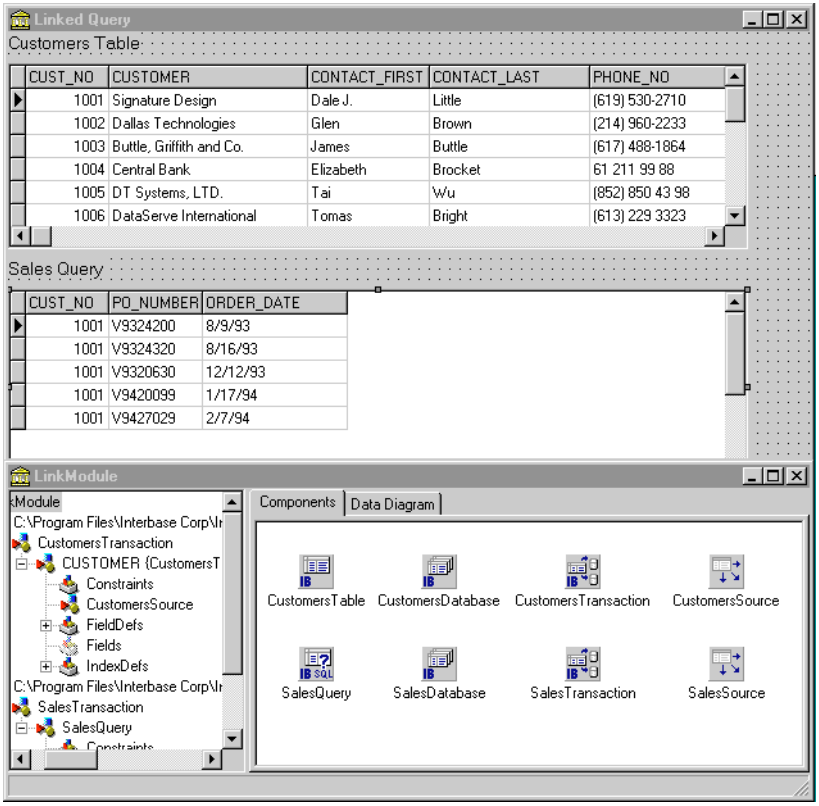
- A *TIBDatabase* component named *SalesDatabase* linked to the EMPLOYEE.GDB database, *SalesQuery* and *SalesTransaction*.
- A *TIBTransaction* component named *SalesTransaction* linked to *SalesQuery* and *SalesDatabase*.
- A *TIBTable* dataset component named *CustomersTable* linked to the CUSTOMER table, *CustomersDatabase* and *CustomersTransaction*.
- A *TIBDatabase* component named *CustomersDatabase* linked to the EMPLOYEE.GDB database, *CustomersTable* and *CustomersTransaction*.
- A *TIBTransaction* component named *CustomersTransaction* linked to *CustomersTable* and *CustomersDatabase*.

- A *TDataSource* component named *SalesSource*. The *DataSet* property of *SalesSource* points to *SalesQuery*.
- A *TDataSource* named *CustomersSource* linked to *CustomersTable*. The *DataSource* property of the *OrdersQuery* component is also set to *CustomersSource*. This is the setting that makes *OrdersQuery* a linked query.

Suppose, too, that this application has a form, named *LinkedQuery* that contains two data grids, a *Customers Table* grid linked to *CustomersSource*, and an *SalesQuery* grid linked to *SalesSource*.

FIGURE 14.1 illustrates how this application appears at design time.

FIGURE 14.1 Sample master/detail query form and data module at design time



Note If you build this application, create the table component and its data source before creating the query component.

If you compile this application, at runtime the *:Cust_No* parameter in the SQL statement for *SalesQuery* is not assigned a value, so *SalesQuery* tries to match the parameter by name against a column in the table pointed to by *CustomersSource*. *CustomersSource* gets its data from *CustomersTable*, which, in turn, derives its data from the CUSTOMER table. Because CUSTOMER contains a column called "Cust_No," the value from the *Cust_No* field in the current record of the *CustomersTable* dataset is assigned to the *:Cust_No* parameter for the *SalesQuery* SQL statement. The grids are linked in a master-detail relationship. At runtime, each time you select a different record in the Customers Table grid, the *SalesQuery* SELECT statement executes to retrieve all orders based on the current customer number.

Executing a query

After you specify an SQL statement in the SQL property and set any parameters for the query, you can execute the query. When a query is executed, the server receives and processes SQL statements from your application. If the query is against local tables, the SQL engine processes the SQL statement and, for a SELECT query, returns data to the application.

Note Before you execute a query for the first time, you may want to call the *Prepare* method to improve query performance. *Prepare* initializes the database server, each of which allocates system resources for the query. For more information about preparing a query, see **"Preparing a query" on page 207**.

The following sections describe executing both static and dynamic SQL statements at design time and at runtime.

Executing a query at design time

To execute a query at design time, set its *Active* property to *True* in the Object Inspector.

The results of the query, if any, are displayed in any data-aware controls associated with the query component.

Note The *Active* property can be used only with queries that returns a result set, such as by the SELECT statement.

Executing a query at runtime

To execute a query at runtime, use one of the following methods:

- *Open* executes a query that returns a result set, such as with the SELECT statement.
- *ExecSQL* executes a query that does not return a result set, such as with the INSERT, UPDATE, or DELETE statements.

Note If you do not know at design time whether a query will return a result set at runtime, code both types of query execution statements in a **try...except** block. Put a call to the *Open* method in the **try** clause. This allows you to suppress the error message that would occur due to using an activate method not applicable to the type of SQL statement used. Check the type of exception that occurs. If it is other than an *ENoResult* exception, the exception occurred for another reason and must be processed. This works because an action query will be executed when the query is activated with the *Open* method, but an exception occurs in addition to that.

```
try
  IBQuery2.Open;
except
  on E: Exception do
    if not (E is ENoResultSet) then
      raise;
end;
```

Executing a query that returns a result set

To execute a query that returns a result set (a query that uses a SELECT statement), follow these steps:

1. Call *Close* to ensure that the query is not already open. If a query is already open you cannot open it again without first closing it. Closing a query and reopening it fetches a new version of data from the server.
2. Call *Open* to execute the query.

For example:

```
IBQuery.Close;
IBQuery.Open;{ query returns a result set }
```

For information on navigating within a result set, see **"Disabling bi-directional cursors" on page 208**. For information on editing and updating a result set, see **"Working with result sets" on page 208**.

Executing a query without a result set

To execute a query that does not return a result set (a query that has an SQL statement such as INSERT, UPDATE, or DELETE), call *ExecSQL* to execute the query.

For example:

```
IBQuery.ExecSQL;  { query does not return a result set }
```

Preparing a query

Preparing a query is an optional step that precedes query execution. Preparing a query submits the SQL statement and its parameters, if any, for parsing, resource allocation, and optimization. The server, too, may allocate resources for the query. These operations improve query performance, making your application faster, especially when working with updatable queries.

An application can prepare a query by calling the *Prepare* method. If you do not prepare a query before executing it, then Delphi automatically prepares it for you each time you call *Open* or *ExecSQL*. Even though Delphi prepares queries for you, it is better programming practice to prepare a query explicitly. That way your code is self-documenting, and your intentions are clear. For example:

```
CustomerQuery.Close;
if not (CustomerQuery.Prepared) then
  CustomerQuery.Prepare;
CustomerQuery.Open;
```

This example checks the query component's *Prepared* property to determine if a query is already prepared. *Prepared* is a Boolean value that is *True* if a query is already prepared. If the query is not already prepared, the example calls the *Prepare* method before calling *Open*.

Unpreparing a query to release resources

The *UnPrepare* method sets the *Prepared* property to *False*. *UnPrepare*

- Ensures that the *SQL* property is prepared prior to executing it again.
- Notifies the server to release any resources it has allocated for the statement.

To unprepare a query, call

```
CustomerQuery.UnPrepare;
```

When you change the text of the *SQL* property for a query, the query component automatically closes and unprepares the query.

Improving query performance

Following are steps you can take to improve query execution speed:

- Set the *TIBQuery* component's *UniDirectional* property to *True* if you do not need to navigate backward through a result set (SQL-92 does not, itself, permit backward navigation through a result set).
- Prepare the query before execution. This is especially helpful when you plan to execute a single query several times. You need only prepare the query once, before its first use. For more information about query preparation, see **"Preparing a query" on page 207**.

Disabling bi-directional cursors

The *UniDirectional* property determines whether or not bi-directional cursors are enabled for a *TIBQuery* component. When a query returns a result set, it also receives a cursor, or pointer to the first record in that result set. The record pointed to by the cursor is the currently active record. The current record is the one whose field values are displayed in data-aware components associated with the result set's data source.

UniDirectional is *False* by default, meaning that the cursor for a result set can navigate both forward and backward through its records. Bi-directional cursor support requires some additional processing overhead, and can slow some queries. To improve query performance, you may be able to set *UniDirectional* to *True*, restricting a cursor to forward movement through a result set.

If you do not need to be able to navigate backward through a result set, you can set *UniDirectional* to *True* for a query. Set *UniDirectional* before preparing and executing a query. The following code illustrates setting *UniDirectional* prior to preparing and executing a query:

```
if not (CustomerQuery.Prepared) then begin
  CustomerQuery.UniDirectional := True;
  CustomerQuery.Prepare;
end;
CustomerQuery.Open;  { returns a result set with a one-way cursor }
```

Working with result sets

By default, the result set returned by a query is read-only. Your application can display field values from the result set in data-aware controls, but users cannot edit those values. To enable editing of a result set, your application must use a *TIBUpdateSQL* component.

Updating a read-only result set

Applications can update data returned in a read-only result set if they are using cached updates. To update a read-only result set associated with a query component:

1. Add a *TIBUpdateSQL* component to the data module in your application to essentially give you the ability to post updates to a read-only dataset.
2. Enter the SQL update statement for the result set to the update component's *ModifySQL*, *InsertSQL*, or *DeleteSQL* properties. To do this more easily, right click on the *TIBUpdateSQL* component to access the UpdateSQL Editor.
3. Set the query component's *CachedUpdate* property to *True*.

For more information about using cached updates, see **Chapter 16, "Working with Cached Updates."**

CHAPTER

15

Working with Stored Procedures

This chapter describes how to use stored procedures in your database applications. A stored procedure is a self-contained program written in the procedure and trigger language specific to the database system used. There are two fundamental types of stored procedures. The first type retrieves data (like with a SELECT query). The retrieved data can be in the form of a dataset consisting of one or more rows of data, divided into one or more columns. Or the retrieved data can be in the form of individual pieces of information. The second type does not return data, but performs an action on data stored in the database (like with a DELETE statement).

InterBase servers return all data (datasets and individual pieces of information) exclusively with output parameters.

In InterBase Express applications, access to stored procedures is provided by the *TIBStoredProc* and *TIBQuery* components. The choice of which to use for the access is predicated on how the stored procedure is coded, how data is returned (if any), and the database system used. The *TIBStoredProc* and *TIBQuery* components are both descendants of *TIBCustomDataSet*, and inherit behaviors from *TIBCustomDataSet*. For more information about *TIBCusomDataSet*, see **Chapter 12, "Understanding Datasets."**

A stored procedure component is used to execute stored procedures that do not return any data, to retrieve individual pieces of information in the form of output parameters, and to relay a returned dataset to an associated data source component. The stored procedure component allows values to be passed to and return from the stored procedure through parameters, each parameter defined in the *Params* property. The stored procedure component is the preferred means for using stored procedures that either do not return any data or only return data through output parameters.

A query component is primarily used to run InterBase stored procedures that only return datasets via output parameters. The query component can also be used to execute a stored procedure that does not return a dataset or output parameter values.

Use parameters to pass distinct values to or return values from a stored procedure. Input parameter values are used in such places as the WHERE clause of a SELECT statement in a stored procedure. An output parameter allows a stored procedure to pass a single value to the calling application. Some stored procedures return a result parameter. See **"Input parameters"** and **"Output parameters"** in the *InterBase 6 Language Reference* and **"Working with Stored Procedures"** in the *InterBase 6 Data Definition Guide* for more information.

When should you use stored procedures?

If your server defines stored procedures, you should use them if they apply to the needs of your application. A database server developer creates stored procedures to handle frequently-repeated database-related tasks. Often, operations that act upon large numbers of rows in database tables—or that use aggregate or mathematical functions—are candidates for stored procedures. If stored procedures exist on the remote database server your application uses, you should take advantage of them in your application. Chances are you need some of the functionality they provide, and you stand to improve the performance of your database application by:

- Taking advantage of the server's usually greater processing power and speed.
- Reducing the amount of network traffic since the processing takes place on the server where the data resides.

For example, consider an application that needs to compute a single value: the standard deviation of values over a large number of records. To perform this function in your application, all the values used in the computation must be fetched from the server, resulting in increased network traffic. Then your application must perform the computation. Because all you want in your application is the end result—a single value representing the standard deviation—it would be far more efficient for a stored procedure on the server to read the data stored there, perform the calculation, and pass your application the single value it requires.

See **"Working with Stored Procedures"** in the *InterBase 6 Data Definition Guide* for more information.

Using a stored procedure

How a stored procedure is used in a Delphi application depends on how the stored procedure was coded, whether and how it returns data, the specific database server used, or a combination of these factors.

In general terms, to access a stored procedure on a server, an application must:

1. Instantiate a *TIBStoredProc* component and optionally associate it with a stored procedure on the server. Or instantiate a *TIBQuery* component and compose the contents of its *SQL* property to perform either a SELECT query against the stored procedure or an EXECUTE command, depending on whether the stored procedure returns a result set. For more information about creating a *TIBStoredProc*, see **"Creating a stored procedure component" on page 214**. For more information about creating a *TIBQuery* component, see **Chapter 14, "Working with Queries"**.
2. Provide input parameter values to the stored procedure component, if necessary. When a stored procedure component is not associated with stored procedure on a server, you must provide additional input parameter information, such as parameter names and data types. For more information about providing input parameter information, see **"Setting parameter information at design time" on page 225**.
3. Execute the stored procedure.
4. Process any result and output parameters. As with any other dataset component, you can also examine the result dataset returned from the server. For more information about output and result parameters, see **"Using output parameters" on page 223** and **"Using the result parameter" on page 224.** For information about viewing records in a dataset, see **"Using stored procedures that return result sets" on page 216**.

Creating a stored procedure component

To create a stored procedure component for a stored procedure on a database server:

1. Place stored procedure, database, and transaction components from the InterBase page of the Component palette in a data module.
2. Set the *Database* and *Transaction* properties of the stored procedure component to the names of the database and transaction components.
3. Set the *DatabaseName* property in the *Database* component.

   Normally you should specify the *DatabaseName* property, but if the server database against which your application runs is currently unavailable, you can still create and set up a stored procedure component by omitting the *DatabaseName* and supplying a stored procedure name and input, output, and result parameters at design time. For more information about input parameters, see **"Using input parameters" on page 222**. For more information about output parameters, see **"Using output parameters" on page 223**. For more information about result parameters, see **"Using the result parameter" on page 224**
4. Optionally set the *StoredProcName* property to the name of the stored procedure to use. If you provided a value for the *Database* property, and the *Database* component is connected to the database, then you can select a stored procedure name from the drop-down list for the property. A single *TIBStoredProc* component can be used to execute any number of stored procedures by setting the *StoredProcName* property to a valid name in the application. It may not always be desirable to set the *StoredProcName* at design time.
5. Double-click the *Params* property value box to invoke the StoredProc Parameters editor to examine input and output parameters for the stored procedure. If you did not specify a name for the stored procedure in Step 4, or you specified a name for the stored procedure that does not exist on the server specified in the *DatabaseName* property in Step 3, then when you invoke the parameters editor, it is empty.

See **"Working with Stored Procedures"** in the *InterBase 6 Data Definition Guide* for more information.

Note If you do not specify the *Database* property in Step 2, then you must use the StoredProc Parameters editor to set up parameters at design time. For information about setting parameters at design time, see **"Setting parameter information at design time" on page 225**.

Creating a stored procedure

Ordinarily, stored procedures are created when the application and its database is created, using tools supplied by InterBase. However, it is possible to create stored procedures at runtime. For more information, see **"Creating procedures"** in the *InterBase 6 Data Definition Guide*.

A stored procedure can be created by an application at runtime using an SQL statement issued from a *TIBQuery* component, typically with a CREATE PROCEDURE statement. If parameters are used in the stored procedure, set the *ParamCheck* property of the *TIBQuery* to *False*. This prevents the *TIBQuery* from mistaking the parameter in the new stored procedure from a parameter for the *TIBQuery* itself.

Note You can also use the SQL Explorer to examine, edit, and create stored procedures on the server.

After the SQL property has been populated with the statement to create the stored procedure, execute it by invoking the *ExecSQL* method.

```
with IBQuery1 do begin
  ParamCheck := False;
  with SQL do begin
    Clear;
    Add('CREATE PROCEDURE GET_MAX_EMP_NAME');
    Add('RETURNS (Max_Name CHAR(15))');
    Add('AS');
    Add('BEGIN');
    Add('  SELECT MAX(LAST_NAME)');
    Add('  FROM EMPLOYEE');
    Add('  INTO :Max_Name;');
    Add('  SUSPEND;');
    Add('END');
  end;
  ExecSQL;
end;
```

Preparing and executing a stored procedure

To use a stored procedure, you can optionally prepare it, and then execute it.

You can prepare a stored procedure at:

- Design time, by choosing OK in the Parameters editor.

- Runtime, by calling the *Prepare* method of the stored procedure component.

For example, the following code prepares a stored procedure for execution:

```
IBStoredProc1.Prepare;
```

Note If your application changes parameter information at runtime, you should prepare the procedure again.

To execute a prepared stored procedure, call the *ExecProc* method for the stored procedure component. The following code illustrates code that prepares and executes a stored procedure:

```
IBStoredProc1.Params[0].AsString := Edit1.Text;
IBStoredProc1.Prepare;
IBStoredProc1.ExecProc;
```

Note If you attempt to execute a stored procedure before preparing it, the stored procedure component automatically prepares it for you, and then unprepares it after it executes. If you plan to execute a stored procedure a number of times, it is more efficient to call *Prepare* yourself, and then only call *UnPrepare* once, when you no longer need to execute the procedure.

When you execute a stored procedure, it can return all or some of these items:

- A dataset consisting of one or more records that can be viewed in data-aware controls associated with the stored procedure through a data source component.
- Output parameters.
- A result parameter that contains status information about the stored procedure's execution.

Using stored procedures that return result sets

Stored procedures that return data in datasets, rows and columns of data, should most often be used with a query component. However, a stored procedure component can also serve this purpose.

Retrieving a result set with a TIBQuery

To retrieve a dataset from a stored procedure using a *TIBQuery* component:

1. Instantiate a query component.
2. In the *TIBQuery.SQL* property, write a SELECT query that uses the name of the stored procedure instead of a table name.

3. If the stored procedure requires input parameters, express the parameter values as a comma-separated list, enclosed in parentheses, following the procedure name.
4. Set the *Active* property to *True* or invoke the *Open* method.

For example, the InterBase stored procedure GET_EMP_PROJ, below, accepts a value using the input parameter EMP_NO and returns a dataset through the output parameter PROJ_ID.

```
CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID CHAR(5))
AS
BEGIN
  FOR SELECT PROJ_ID
  FROM EMPLOYEE_PROJECT
  WHERE EMP_NO = :EMP_NO
  INTO :PROJ_ID
  DO
    SUSPEND;
END
```

The SQL statement issued from a *TIBQuery* to use this stored procedure would be:

```
SELECT *
FROM GET_EMP_PROJ(52)
```

Using stored procedures that return data using parameters

Stored procedures can be composed to retrieve individual pieces of information, as opposed to whole rows of data, through parameters. For instance, a stored procedure might retrieve the maximum value for a column, add one to that value, and then return that value to the application. Such stored procedures can be used and the values inspected using either a *TIBQuery* or a *TIBStoredProc* component. The preferred method for retrieving parameter values is with a *TIBStoredProc*.

Retrieving individual values with a TIBQuery

Parameter values retrieved via a *TIBQuery* component take the form of a single-row dataset, even if only one parameter is returned by the stored procedure. To retrieve individual values from stored procedure parameters using a *TIBQuery* component:

1. Instantiate a query component.
2. In the *TIBQuery.SQL* property, write a SELECT query that uses the name of the stored procedure instead of a table name. The SELECT clause of this query can specify the parameter by its name, as if it were a column in a table, or it can simply use the * operator to retrieve all parameter values.
3. If the stored procedure requires input parameters, express the parameter values as a comma-separated list, enclosed in parentheses, following the procedure name.
4. Set the *Active* property to *True* or invoke the *Open* method.

For example, the InterBase stored procedure GET_HIGH_EMP_NAME, below, retrieves the alphabetically last value in the LAST_NAME column of a table named EMPLOYEE. The stored procedure returns this value in the output parameter *High_Last_Name*.

```
CREATE PROCEDURE GET_HIGH_EMP_NAME
RETURNS (High_Last_Name CHAR(15))
AS
BEGIN
  SELECT MAX(LAST_NAME)
  FROM EMPLOYEE
  INTO :High_Last_Name;
  SUSPEND;
END
```

The SQL statement issued from a *TIBQuery* to use this stored procedure would be:

```
SELECT High_Last_Name
FROM GET_HIGH_EMP_NAME
```

Retrieving individual values with a TIBStoredProc

To retrieve individual values from stored procedure output parameters using a *TIBStoredProc* component:

1. Instantiate a stored procedure component.
2. In the *StoredProcName* property, specify the name of the stored procedure.
3. If the stored procedure requires input parameters, supply values for the parameters using the *Params* property or *ParamByName* method.
4. Invoke the *ExecProc* method.
5. Inspect the values of individual output parameters using the *Params* property or *ParamByName* method.

For example, the InterBase stored procedure GET_HIGH_EMP_NAME, below, retrieves the alphabetically last value in the LAST_NAME column of a table named EMPLOYEE. The stored procedure returns this value in the output parameter *High_Last_Name*.

```
CREATE PROCEDURE GET_HIGH_EMP_NAME
RETURNS (High_Last_Name CHAR(15))
AS
BEGIN
  SELECT MAX(LAST_NAME)
  FROM EMPLOYEE
  INTO :High_Last_Name;
  SUSPEND;
END
```

The Delphi code to get the value in the High_Last_Name output parameter and store it to the *Text* property of a *TEdit* component is:

```
with StoredProc1 do begin
  StoredProcName := 'GET_HIGH_EMP_NAME';
  ExecProc;
  Edit1.Text := ParamByName('High_Last_Name').AsString;
end;
```

Using stored procedures that perform actions on data

Stored procedures can be coded such that they do not return any data at all, and only perform some action in the database. SQL operations involving the INSERT and DELETE statements are good examples of this type of stored procedure. For instance, instead of allowing a user to delete a row directly, a stored procedure might be used to do so. This would allow the stored procedure to control what is deleted and also to handle any referential integrity aspects, such as a cascading delete of rows in dependent tables.

▸ *Executing an action stored procedure with a TIBQuery*

To execute an action stored procedure using a *TIBQuery* component:

1. Instantiate a query component.
2. In the *TIBQuery.SQL* property, include the command necessary to execute the stored procedure and the stored procedure name. (The command to execute a stored procedure can vary from one database system to another. In InterBase, the command is EXECUTE PROCEDURE.)

3. If the stored procedure requires input parameters, express the parameter values as a comma-separated list, enclosed in parentheses, following the procedure name.
4. Invoke the *TIBQuery.ExecSQL* method.

For example, the InterBase stored procedure ADD_EMP_PROJ, below, adds a new row to the table EMPLOYEE_PROJECT. No dataset is returned and no individual values are returned in output parameters.

```
CREATE PROCEDURE ADD_EMP_PROJ (EMP_NO SMALLINT, PROJ_ID CHAR(5))
AS
BEGIN
  BEGIN
    INSERT INTO EMPLOYEE_PROJECT (EMP_NO, PROJ_ID)
    VALUES (:EMP_NO, :PROJ_ID);
    WHEN SQLCODE -530 DO
      EXCEPTION UNKNOWN_EMP_ID;
  END
  SUSPEND;
END
```

The SQL statement issued from a *TIBQuery* to execute this stored procedure would be:

```
EXECUTE PROCEDURE ADD_EMP_PROJ(20, "GUIDE")
```

Executing an action stored procedure with a TIBStoredProc

To retrieve individual values from stored procedure output parameters using a *TIBStoredProc* component:

1. Instantiate a stored procedure component.
2. In the *StoredProcName* property, specify the name of the stored procedure.
3. If the stored procedure requires input parameters, supply values for the parameters using the *Params* property or *ParamByName* method.
4. Invoke the *ExecProc* method.

For example, the InterBase stored procedure ADD_EMP_PROJ, below, adds a new row to the table EMPLOYEE_PROJECT. No dataset is returned and no individual values are returned in output parameters.

```
CREATE PROCEDURE ADD_EMP_PROJ (EMP_NO SMALLINT, PROJ_ID CHAR(5))
AS
BEGIN
  BEGIN
    INSERT INTO EMPLOYEE_PROJECT (EMP_NO, PROJ_ID)
```

```
    VALUES (:EMP_NO, :PROJ_ID);
    WHEN SQLCODE -530 DO
      EXCEPTION UNKNOWN_EMP_ID;
  END
  SUSPEND;
END
```

The Delphi code to execute the ADD_EMP_PROJ stored procedure is:

```
with StoredProc1 do begin
  StoredProcName := 'ADD_EMP_PROJ';
  ExecProc;
end;
```

Understanding stored procedure parameters

There are four types of parameters that can be associated with stored procedures:

- *Input parameters*, used to pass values to a stored procedure for processing.
- *Output parameters*, used by a stored procedure to pass return values to an application.
- *Input/output parameters*, used to pass values to a stored procedure for processing, and used by the stored procedure to pass return values to the application.
- A *result parameter*, used to return an error or status value to an application. A stored procedure can only return one result parameter.

Whether a stored procedure uses a particular type of parameter depends both on the general language implementation of stored procedures on your database server and on a specific instance of a stored procedure. For example, individual stored procedures on any server may either be implemented using input parameters, or may not be. On the other hand, some uses of parameters are server-specific. For example, the InterBase implementation of a stored procedure never returns a result parameter.

Access to stored procedure parameters is provided by *TParam* objects in the *TIBStoredProc.Params* property. If the name of the stored procedure is specified at design time in the *StoredProcName* property, a *TParam* object is automatically created for each parameter and added to the *Params* property. If the stored procedure name is not specified until runtime, the *TParam* objects need to be programmatically created at that time. Not specifying the stored procedure and manually creating the *TParam* objects allows a single *TIBStoredProc* component to be used with any number of available stored procedures.

Note Some stored procedures return a dataset in addition to output and result parameters. Applications can display dataset records in data-aware controls, but must separately process output and result parameters. For more information about displaying records in data-aware controls, see **"Using stored procedures that return result sets" on page 216**.

Using input parameters

Application use input parameters to pass singleton data values to a stored procedure. Such values are then used in SQL statements within the stored procedure, such as a comparison value for a WHERE clause. If a stored procedure requires an input parameter, assign a value to the parameter prior to executing the stored procedure.

If a stored procedure returns a dataset and is used through a SELECT query in a *TIBQuery* component, supply input parameter values as a comma-separated list, enclosed in parentheses, following the stored procedure name. For example, the SQL statement below retrieves data from a stored procedure named GET_EMP_PROJ and supplies an input parameter value of 52.

```
SELECT PROJ_ID
FROM GET_EMP_PROJ(52)
```

If a stored procedure is executed with a *TIBStoredProc* component, use the *Params* property or the *ParamByName* method access to set each input parameter. Use the *TParam* property appropriate for the data type of the parameter, such as the *TParam.AsString* property for a CHAR type parameter. Set input parameter values prior to executing or activating the *TIBStoredProc* component. In the example below, the EMP_NO parameter (type SMALLINT) for the stored procedure GET_EMP_PROJ is assigned the value 52.

```
with IBStoredProc1 do begin
  ParamByName('EMP_NO').AsSmallInt := 52;
  ExecProc;
end;
```

Using output parameters

Stored procedures use output parameters to pass singleton data values to an application that calls the stored procedure. Output parameters are not assigned values except by the stored procedure and then only after the stored procedure has been executed. Inspect output parameters from an application to retrieve its value after invoking the *TIBStoredProc.ExecProc* method.

Use the *TIBStoredProc.Params* property or *TIBStoredProc.ParamByName* method to reference the *TParam* object that represents a parameter and inspect its value. For example, to retrieve the value of a parameter and store it into the *Text* property of a *TEdit* component:

```
with IBStoredProc1 do begin
  ExecProc;
  Edit1.Text := Params[0].AsString;
end;
```

Most stored procedures return one or more output parameters. Output parameters may represent the sole return values for a stored procedure that does not also return a dataset, they may represent one set of values returned by a procedure that also returns a dataset, or they may represent values that have no direct correspondence to an individual record in the dataset returned by the stored procedure. Each server's implementation of stored procedures differs in this regard.

Using input/output parameters

Input/output parameters serve both function that input and output parameters serve individually. Applications use an input/output parameter to pass a singleton data value to a stored procedure, which in turn reuses the input/output parameter to pass a singleton data value to the calling application. As with input parameters, the input value for an input/output parameter must be set before the using stored procedure or query component is activated. Likewise, the output value in an input/output parameter will not be available until after the stored procedure has been executed.

In the example Oracle stored procedure below, the parameter IN_OUTVAR is an input/output parameter.

```
CREATE OR REPLACE PROCEDURE UPDATE_THE_TABLE (IN_OUTVAR IN OUT
INTEGER)
AS
BEGIN
  UPDATE ALLTYPETABLE
```

```
  SET NUMBER82FLD = IN_OUTVAR
  WHERE KEYFIELD = 0;
  IN_OUTVAR:=1;
END UPDATE_THE_TABLE;
```

In the Delphi program code below, IN_OUTVAR is assigned an input value, the stored procedure executed, and then the output value in IN_OUTVAR is inspected and stored to a memory variable.

```
with StoredProc1 do begin
  ParamByName('IN_OUTVAR').AsInteger := 103;
  ExecProc;
  IntegerVar := ParamByName('IN_OUTVAR').AsInteger;
end;
```

Using the result parameter

In addition to returning output parameters and a dataset, some stored procedures also return a single result parameter. The result parameter is usually used to indicate an error status or the number of records processed base on stored procedure execution. See your database server's documentation to determine if and how your server supports the result parameter. Result parameters are not assigned values except by the stored procedure and then only after the stored procedure has been executed. Inspect a result parameter from an application to retrieve its value after invoking the *TIBStoredProc.ExecProc* method.

Use the *TIBStoredProc.Params* property or *TIBStoredProc.ParamByName* method to reference the TParam object that represents the result parameter and inspect its value.

```
DateVar := StoredProc1.ParamByName('MyOutputParam').AsDate;
```

Accessing parameters at design time

If you connect to a remote database server by setting the *Database* and *StoredProcName* properties at design time, then you can use the StoredProc Parameters editor to view the names and data types of each input parameter, and you can set the values for the input parameters to pass to the server when you execute the stored procedure.

IMPORTANT Do not change the names or data types for input parameters reported by the server, or when you execute the stored procedure an exception is raised.

Some servers do not report parameter names or data types. In these cases, use the SQL Explorer or IBConsole to look at the source code of the stored procedure on the server to determine input parameters and data types. See the SQL Explorer online help for more information.

At design time, if you do not receive a parameter list from a stored procedure on a remote server (for example because you are not connected to a server), then you must invoke the *StoredProc* Parameters editor, list each required input parameter, and assign each a data type and a value. For more information about using the *StoredProc* Parameters editor to create parameters, see **"Setting parameter information at design time" on page 225**.

Setting parameter information at design time

You can invoke the *StoredProc* parameter collection editor at design time to set up parameters and their values.

The parameter collection editor allows you to set up stored procedure parameters. If you set the *Database* and *StoredProcName* properties of the *TIBStoredProc* component at design time, all existing parameters are listed in the collection editor. If you do not set both of these properties, no parameters are listed and you must add them manually. Additionally, some database types do not return all parameter information, like types. For these database systems, use the SQL Explorer utility to inspect the stored procedures, determine types, and then configure parameters through the collection editor and the Object Inspector. The steps to set up stored procedure parameters at design time are:

1. Optionally set the *Database* and *StoredProcName* properties.
2. In the Object Inspector, invoke the parameter collection editor by clicking on the ellipsis button in the *Params* field.
3. If the *Database* and *StoredProcName* properties are not set, no parameters appear in the collection editor. Manually add parameter definitions by right-clicking within the collection editor and selecting Add from the context menu.
4. Select parameters individually in the collection editor to display their properties in the Object Inspector.
5. If a type is not automatically specified for the *ParamType* property, select a parameter type (*Input*, *Output*, *Input/Output*, or *Result*) from the property's drop-down list.
6. If a data type is not automatically specified for the *DataType* property, select a data type from the property's drop-down list.

7. Use the *Value* property to optionally specify a starting value for an input or input/output parameter.

Right-clicking in the parameter collection editor invokes a context menu for operating on parameter definitions. Depending on whether any parameters are listed or selected, enabled options include: adding new parameters, deleting existing parameters, moving parameters up and down in the list, and selecting all listed parameters.

You can edit the definition for any *TParam* you add, but the attributes of the *TParam* objects you add must match the attributes of the parameters for the stored procedure on the server. To edit the *TParam* for a parameter, select it in the parameter collection editor and edit its property values in the Object Inspector.

Note You can never set values for output and result parameters. These types of parameters have values set by the execution of the stored procedure.

Creating parameters at runtime

If the name of the stored procedure is not specified in StoredProcName until runtime, no *TParam* objects will be automatically created for parameters and they must be created programmatically. This can be done using the *TParam.Create* method or the *TParams.AddParam* method.

For example, the InterBase stored procedure GET_EMP_PROJ, below, requires one input parameter (EMP_NO) and one output parameter (PROJ_ID).

```
CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID CHAR(5))
AS
BEGIN
  FOR SELECT PROJ_ID
  FROM EMPLOYEE_PROJECT
  WHERE EMP_NO = :EMP_NO
  INTO :PROJ_ID
  DO
    SUSPEND;
END
```

The Delphi code to associate this stored procedure with a *TIBStoredProc* named *StoredProc1* and create *TParam* objects for the two parameters using the *TParam.Create* method is:

```
var
  P1, P2: TParam;
```

```
begin
  {...}
  with StoredProc1 do begin
    StoredProcName := 'GET_EMP_PROJ';
    Params.Clear;
    P1 := TParam.Create(Params, ptInput);
    P2 := TParam.Create(Params, ptOutput);
    try
      Params[0].Name := ‘EMP_NO’;
      Params[1].Name := ‘PROJ_ID’;
      ParamByname(‘EMP_NO’).AsSmallInt := 52;
      ExecProc;
      Edit1.Text := ParamByname(‘PROJ_ID’).AsString;
    finally
      P1.Free;
      P2.Free;
    end;
  end;
  {...}
end;
```

Viewing parameter information at design time

If you have access to a database server at design time, there are two ways to view information about the parameters used by a stored procedure:

- Invoke the SQL Explorer to view the source code for a stored procedure on a remote server. The source code includes parameter declarations that identify the data types and names for each parameter.
- Use the Object Inspector to view the property settings for individual *TParam* objects.

You can use the SQL Explorer to examine stored procedures on your database servers. If you are using ODBC drivers you cannot examine stored procedures with the SQL Explorer. While using the SQL Explorer is not always an option, it can sometimes provide more information than the Object Inspector viewing *TParam* objects. The amount of information returned about a stored procedure in the Object Inspector depends on your database server.

To view individual parameter definitions in the Object Inspector:

1. Select the stored procedure component.
2. Set the *Database* property of a stored procedure component to the *Database* property of a *TIBDatabase*.
3. Set the *StoredProcName* property to the name of the stored procedure.
4. Click the ellipsis button in for the *TIBStoredProc.Params* property in the Object Inspector.
5. Select individual parameters in the collection editor to view their property settings in the Object Inspector.

For some servers some or all parameter information may not be accessible.

In the Object Inspector, when viewing individual *TParam* objects, the *ParamType* property indicates whether the selected parameter is an input, output, input/output, or result parameter. The *DataType* property indicates the data type of the value the parameter contains, such as string, integer, or date. The *Value* edit box enables you to enter a value for a selected input parameter.

For more about setting parameter values, see **"Setting parameter information at design time" on page 225**.

Note You can never set values for output and result parameters. These types of parameters have values set by the execution of the stored procedure.

CHAPTER

16

Working with Cached Updates

Cached updates enable you to retrieve data from a database, cache and edit it locally, and then apply the cached updates to the database as a unit. When cached updates are enabled, updates to a dataset (such as posting changes or deleting records) are stored in an internal cache instead of being written directly to the dataset's underlying table. When changes are complete, your application calls a method that writes the cached changes to the database and clears the cache.

This chapter describes when and how to use cached updates. It also describes the *TIBUpdateSQL* component that can be used in conjunction with cached updates to update virtually any dataset, particularly datasets that are not normally updatable.

Deciding when to use cached updates

Cached updates are primarily intended to reduce data access contention on remote database servers by:

- Minimizing transaction times.
- Minimizing network traffic.

While cached updates can minimize transaction times and drastically reduce network traffic, they may not be appropriate for all database client applications that work with remote servers. There are three areas of consideration when deciding to use cached updates:

- Cached data is local to your application, and is not under transaction control. In a busy client/server environment this has two implications for your application:
  - Other applications can access and change the actual data on the server while your users edit their local copies of the data.
  - Other applications cannot see any data changes made by your application until it applies all its changes.
- In master/detail relationships managing the order of applying cached updates can be tricky. This is particularly true when there are nested master/detail relationships where one detail table is the master table for yet another detail table and so on.
- Applying cached updates to read-only, query-based datasets requires use of update objects.

The InterBase Express components provide cached update methods and transaction control methods you can use in your application code to handle these situations, but you must take care that you cover all possible scenarios your application is likely to encounter in your working environment.

Using cached updates

This section provides a basic overview of how cached updates work in an application. If you have not used cached updates before, this process description serves as a guideline for implementing cached updates in your applications.

To use cached updates, the following order of processes must occur in an application:

1. Enable cached updates. Enabling cached updates causes a read-only transaction that fetches as much data from the server as is necessary for display purposes and then terminates. Local copies of the data are stored in memory for display and editing. For more information about enabling and disabling cached updates, see **"Enabling and disabling cached updates" on page 231**.
2. Display and edit the local copies of records, permit insertion of new records, and support deletions of existing records. Both the original copy of each record and any edits to it are stored in memory. For more information about displaying and editing when cached updates are enabled, see **"Applying cached updates" on page 233**.

3. Fetch additional records as necessary. As a user scrolls through records, additional records are fetched as needed. Each fetch occurs within the context of another short duration, read-only transaction. (An application can optionally fetch all records at once instead of fetching many small batches of records.) For more information about fetching all records, see **"Fetching records" on page 232**.
4. Continue to display and edit local copies of records until all desired changes are complete.
5. Apply the locally cached records to the database or cancel the updates. For each record written to the database, an *OnUpdateRecord* event is triggered. If an error occurs when writing an individual record to the database, an *OnUpdateError* event is triggered which enables the application to correct the error, if possible, and continue updating. When updates are complete, all successfully applied updates are cleared from the local cache. For more information about applying updates to the database, see **"Applying cached updates" on page 233**.

If instead of applying updates, an application cancels updates, the locally cached copy of the records and all changes to them are freed without writing the changes to the database. For more information about canceling updates, see **"Canceling pending cached updates" on page 236**.

Enabling and disabling cached updates

Cached updates are enabled and disabled through the *CachedUpdates* properties of *TIBDataSet*, *TIBTable*, *TIBQuery*, and *TStoredProc*. *CachedUpdates* is *False* by default, meaning that cached updates are not enabled for a dataset.

Note Client datasets always cache updates. They have no *CachedUpdates* property because you cannot disable cached updates on a client dataset.

To use cached updates, set *CachedUpdates* to *True*, either at design time (through the Object Inspector), or at runtime. When you set *CachedUpdates* to *True*, the dataset's *OnUpdateRecord* event is triggered if you provide it. For more information about the *OnUpdateRecord* event, see **"Creating an OnUpdateRecord event handler" on page 255**.

For example, the following code enables cached updates for a dataset at runtime:

```
CustomersTable.CachedUpdates := True;
```

When you enable cached updates, a copy of all records necessary for display and editing purposes is cached in local memory. Users view and edit this local copy of data. Changes, insertions, and deletions are also cached in memory. They accumulate in memory until the current cache of local changes is applied to the database. If changed records are successfully applied to the database, the record of those changes are freed in the cache.

Note Applying cached updates does not disable further cached updates; it only writes the current set of changes to the database and clears them from memory.

To disable cached updates for a dataset, set *CachedUpdates* to *False*. If you disable cached updates when there are pending changes that you have not yet applied, those changes are discarded without notification. Your application can test the *UpdatesPending* property for this condition before disabling cached updates. For example, the following code prompts for confirmation before disabling cached updates for a dataset:

```
if (CustomersTable.UpdatesPending)
  if (Application.MessageBox("Discard pending updates?",
                             "Unposted changes",
                             MB_YES + MB_NO) = IDYES) then
    CustomersTable.CachedUpdates = False;
```

Fetching records

By default, when you enable cached updates, datasets automatically handle fetching of data from the database when necessary. Datasets fetch enough records for display. During the course of processing, many such record fetches may occur. If your application has specific needs, it can fetch all records at one time. You can fetch all records by calling the dataset's *FetchAll* method. *FetchAll* creates an in-memory, local copy of all records from the dataset. If a dataset contains many records or records with large Blob fields, you may not want to use *FetchAll*.

Client datasets use the *PacketRecords* property to indicate the number of records that should be fetched at any time. If you set the *FetchOnDemand* property to *True*, the client dataset automatically handles fetching of data when necessary. Otherwise, you can use the *GetNextPacket* method to fetch records from the data server. For more information about fetching records using a client dataset, see "Requesting data from an application server" in the "Creating and using a client dataset" chapter of the *Delphi 5 Developer's Guide*.

Applying cached updates

When a dataset is in cached update mode, changes to data are not actually written to the database until your application explicitly calls methods that apply those changes. Normally an application applies updates in response to user input, such as through a button or menu item.

IMPORTANT To apply updates to a set of records retrieved by an SQL query that does not return a live result set, you must use a *TIBUpdateSQL* object to specify how to perform the updates. For updates to joins (queries involving two or more tables), you must provide one *TIBUpdateSQL* object for each table involved, and you must use the *OnUpdateRecord* event handler to invoke these objects to perform the updates. For more information, see **"Updating a read-only result set" on page 254**. For more information about creating and using an *OnUpdateRecord* event handler, see **"Creating an OnUpdateRecord event handler" on page 255**.

Applying updates is a two-phase process that should occur in the context of a transaction component to enable your application to recover gracefully from errors.

When applying updates under transaction control, the following events take place:

1. A database transaction starts.
2. Cached updates are written to the database (phase 1). If you provide it, an *OnUpdateRecord* event is triggered once for each record written to the database. If an error occurs when a record is applied to the database, the *OnUpdateError* event is triggered if you provide one.

If the database write is unsuccessful:

- Database changes are rolled back, ending the database transaction.
- Cached updates are not committed, leaving them intact in the internal cache buffer.

If the database write is successful:

- Database changes are committed, ending the database transaction.
- Cached updates are committed, clearing the internal cache buffer (phase 2).

The two-phased approach to applying cached updates allows for effective error recovery, especially when updating multiple datasets (for example, the datasets associated with a master/detail form). For more information about handling update errors that occur when applying cached updates, see **"Handling cached update errors" on page 256**.

There are actually two ways to apply updates. To apply updates for a specified set of datasets associated with a database component, call the database component's *ApplyUpdates* method. To apply updates for a single dataset, call the dataset's *ApplyUpdates* and *Commit* methods. These choices, and their strengths, are described in the following sections.

Applying cached updates with a database component method

Ordinarily, applications cache updates at the dataset level. However, there are times when it is important to apply the updates to multiple interrelated datasets in the context of a single transaction. For example, when working with master/detail forms, you will likely want to commit changes to master and detail tables together.

To apply cached updates to one or more datasets in the context of a database connection, call the database component's *ApplyUpdates* method. The following code applies updates to the *CustomersQuery* dataset in response to a button click event:

```
procedure TForm1.ApplyButtonClick(Sender: TObject);
begin
    IBDatabase1.ApplyUpdates([CustomersQuery]);
end;
```

The above sequence starts a transaction, and writes cached updates to the database. If successful, it also commits the transaction, and then commits the cached updates. If unsuccessful, this method rolls back the transaction, and does not change the status of the cached updates. In this latter case, your application should handle cached update errors through a dataset's *OnUpdateError* event. For more information about handling update errors, see **"Handling cached update errors" on page 256**.

The main advantage to calling a database component's *ApplyUpdates* method is that you can update any number of dataset components that are associated with the database. The parameter for the *ApplyUpdates* method for a database is an array of *TIBCustomDataSet*. For example, the following code applies updates for two queries used in a master/detail form:

```
IBDatabase1.ApplyUpdates([CustomerQuery, OrdersQuery]);
```

For more information about updating master/detail tables, see **"Applying updates for master/detail tables" on page 235**.

Applying cached updates with dataset component methods

You can apply updates for individual datasets directly using the dataset's *ApplyUpdates* and *CommitUpdates* methods. Each of these methods encapsulate one phase of the update process:

1. *ApplyUpdates* writes cached changes to a database (phase 1).
2. *CommitUpdates* clears the internal cache when the database write is successful (phase 2).

Applying updates at the dataset level gives you control over the order in which updates are applied to individual datasets. Order of update application is especially critical for handling master/detail relationships. To ensure the correct ordering of updates for master/detail tables, you should always apply updates at the dataset level. For more information see **"Applying updates for master/detail tables" on page 235**.

The following code illustrates how you apply updates within a transaction for the *CustomerQuery* dataset previously used to illustrate updates through a database method:

```
procedure TForm1.ApplyButtonClick(Sender: TObject)
begin
  IBTransaction1.StartTransaction;
  CustomerQuery.ApplyUpdates;  {try to write the updates to the
database }
  IBTransaction1.Commit;           { on success, commit the changes }
  except
    IBTransaction1.Rollback;         { on failure, undo any changes }
    raise;              { raise the exception again to prevent a call
to CommitUpdates }
  end;
  CustomerQuery.CommitUpdates; {on success, clear the internal cache }
end;
```

If an exception is raised during the *ApplyUpdates* call, the database transaction is rolled back. Rolling back the transaction ensures that the underlying database table is not changed. The *raise* statement inside the **`try...except`** block re-*raises* the exception, thereby preventing the call to *CommitUpdates*. Because *CommitUpdates* is not called, the internal cache of updates is not cleared so that you can handle error conditions and possibly retry the update.

Applying updates for master/detail tables

When you apply updates for master/detail tables, the order in which you list datasets to update is significant. Generally you should always update master tables before detail tables, except when handling deleted records. In complex master/detail relationships where the detail table for one relationship is the master table for another detail table, the same rule applies.

You can update master/detail tables at the database or dataset component levels. For purposes of control (and of creating explicitly self-documented code), you should apply updates at the dataset level. The following example illustrates how you should code cached updates to two tables, *Master* and *Detail*, involved in a master/detail relationship:

```
IBTransaction1.StartTransaction;
try
  Master.ApplyUpdates;
  Detail.ApplyUpdates;
  Database1.Commit;
except
  IBTransaction1.Rollback;
  raise;
end;
Master.CommitUpdates;
Detail.CommitUpdates;
```

If an error occurs during the application of updates, this code also leaves both the cache and the underlying data in the database tables in the same state they were in before the calls to *ApplyUpdates*.

If an exception is raised during the call to *Master.ApplyUpdates*, it is handled like the single dataset case previously described. Suppose, however, that the call to *Master.ApplyUpdates* succeeds, and the subsequent call to *Detail.ApplyUpdates* fails. In this case, the changes are already applied to the master table. Because all data is updated inside a database transaction, however, even the changes to the master table are rolled back when *IBTransaction1.Rollback* is called in the except block. Furthermore, *UpdatesMaster.CommitUpdates* is not called because the exception which is re-*raised* causes that code to be skipped, so the cache is also left in the state it was before the attempt to update.

To appreciate the value of the two-phase update process, assume for a moment that *ApplyUpdates* is a single-phase process which updates the data *and the cache*. If this were the case, and if there were an error while applying the updates to the *Detail* table, then there would be no way to restore both the data and the cache to their original states. Even though the call to *IBTransaction1.Rollback* would restore the database, there would be no way to restore the cache.

Canceling pending cached updates

Pending cached updates are updated records that are posted to the cache but not yet applied to the database. There are three ways to cancel pending cached updates:

- To cancel all pending updates and disable further cached updates, set the *CachedUpdates* property to *False*.
- To discard all pending updates without disabling further cached updates, call the *CancelUpdates* method.
- To cancel updates made to the current record call *RevertRecord*.

The following sections discuss these options in more detail.

Cancelling pending updates and disabling further cached updates

To cancel further caching of updates and delete all pending cached updates without applying them, set the *CachedUpdates* property to *False*. When *CachedUpdates* is set to *False*, the *CancelUpdates* method is automatically invoked.

From the update cache, deleted records are undeleted, modified records revert to original values, and newly inserted record simply disappear.

Note This option is not available for client datasets.

Canceling pending cached updates

CancelUpdates clears the cache of all pending updates, and restores the dataset to the state it was in when the table was opened, cached updates were last enabled, or updates were last successfully applied. For example, the following statement cancels updates for the *CustomersTable*:

```
CustomersTable.CancelUpdates;
```

From the update cache, deleted records are undeleted, modified records revert to original values, and newly inserted records simply disappear.

Note Calling *CancelUpdates* does not disable cached updating. It only cancels currently pending updates. To disable further cached updates, set the *CachedUpdates* property to *False*.

Canceling updates to the current record

RevertRecord restores the current record in the dataset to the state it was in when the table was opened, cached updates were last enabled, or updates were last successfully applied. It is most frequently used in an *OnUpdateError* event handler to correct error situations. For example,

```
CustomersTable.RevertRecord;
```

Undoing cached changes to one record does not affect any other records. If only one record is in the cache of updates and the change is undone using *RevertRecord*, the *UpdatesPending* property for the dataset component is automatically changed from *True* to *False*.

If the record is not modified, this call has no effect. For more information about creating an *OnUpdateError* handler, see **"Creating an OnUpdateRecord event handler" on page 255**.

Undeleting cached records

To undelete a cached record requires some coding because once the deleted record is posted to the cache, it is no longer the current record and no longer even appears in the dataset. In some instances, however, you may want to undelete such records. The process involves using the *UpdateRecordTypes* property to make the deleted records "visible," and then calling *RevertRecord*. Here is a code example that undeletes all deleted records in a table:

```
procedure TForm1.UndeleteAll(DataSet: TDataSet)
begin
  DataSet.UpdateRecordTypes := [cusDeleted];
       { show only deleted records }
  try
    DataSet.First;
       { go to the first previously deleted record }
    while not (DataSet.Eof)
      DataSet.RevertRecord;
       { undelete until we reach the last record ]
  except
             { restore updates types to recognize only modified,
inserted, and unchanged }
    DataSet.UpdateRecordTypes := [cusModified, cusInserted,
cusUnmodified];
    raise;
  end;
  DataSet.UpdateRecordTypes := [cusModified, cusInserted,
cusUnmodified];
end;
```

Specifying visible records in the cache

The *UpdateRecordTypes* property controls what type of records are visible in the cache when cached updates are enabled. *UpdateRecordTypes* works on cached records in much the same way as filters work on tables. *UpdateRecordTypes* is a set, so it can contain any combination of the following values:

Value	Meaning
cusModified	Modified records
cusInserted	Inserted records
cusDeleted	Deleted records
cusUninserted	Uninserted records
cusUnmodified	Unmodified records

TABLE 16.1 TIBUpdateRecordType values

The default value for *UpdateRecordTypes* includes only *cusModified, cusInserted, cusUnmodified*, and *cusUninserted* with deleted records (*cusDeleted*) not displayed.

The *UpdateRecordTypes* property is primarily useful in an *OnUpdateError* event handler for accessing deleted records so they can be undeleted through a call to *RevertRecord*. This property is also useful if you wanted to provide a way in your application for users to view only a subset of cached records, for example, all newly inserted (*cusInserted*) records.

For example, you could have a set of four radio buttons (*RadioButton1* through *RadioButton4*) with the captions All, Modified, Inserted, and Deleted. With all four radio buttons assigned to the same *OnClick* event handler, you could conditionally display all records (except deleted, the default), only modified records, only newly inserted records, or only deleted records by appropriately setting the *UpdateRecordTypes* property.

```
procedure TForm1.UpdateFilterRadioButtonsClick(Sender: TObject);
begin
  if RadioButton1.Checked then
    CustomerQuery.UpdateRecordTypes := [cusUnmodified, cusModified,
cusInserted]
  else if RadioButton2.Checked then
    CustomerQuery.UpdateRecordTypes := [cusModified]
  else if RadioButton3.Checked then
    CustomerQuery.UpdateRecordTypes := [cusInserted]
```

```
  else
    CustomerQuery.UpdateRecordTypes := [cusDeleted];
end;
```

For more information about creating an *OnUpdateError* handler, see **"Creating an OnUpdateRecord event handler" on page 255**.

Checking update status

When cached updates are enabled for your application, you can keep track of each pending update record in the cache by examining the *UpdateStatus* property for the record. Checking update status is most frequently used in *OnUpdateRecord* and *OnUpdateError* event handlers. For more information about creating and using an *OnUpdateRecord* event, see **"Creating an OnUpdateRecord event handler" on page 255**. For more information about creating and using an *OnUpdateError* event, see **"Handling cached update errors" on page 256**.

As you iterate through a set of pending changes, *UpdateStatus* changes to reflect the update status of the current record. *UpdateStatus* returns one of the following values for the current record:

Value	Meaning
usUnmodified	Record is unchanged
usModified	Record is changed
usInserted	Record is a new record
usDeleted	Record is deleted

TABLE 16.2 Return values for UpdateStatus

When a dataset is first opened all records will have an update status of *usUnmodified*. As records are inserted, deleted, and so on, the status values change. Here is an example of *UpdateStatus* property used in a handler for a dataset's *OnScroll* event. The event handler displays the update status of each record in a status bar.

```
procedure TForm1.CustomerQueryAfterScroll(DataSet: TDataSet);
begin
  with CustomerQuery do begin
    case UpdateStatus of
      usUnmodified: StatusBar1.Panels[0].Text := 'Unmodified';
      usModified: StatusBar1.Panels[0].Text := 'Modified';
```

```
      usInserted: StatusBar1.Panels[0].Text := 'Inserted';
      usDeleted: StatusBar1.Panels[0].Text := 'Deleted';
      else StatusBar1.Panels[0].Text := 'Undetermined status';
    end;
  end;
end;
```

Note If a record's *UpdateStatus* is *usModified*, you can examine the *OldValue* property for each field in the dataset to determine its previous value. *OldValue* is meaningless for records with *UpdateStatus* values other than *usModified*. For more information about examining and using *OldValue*, see "Accessing a field's OldValue, NewValue, and CurValue properties" in the *Delphi 5 Developer's Guide*.

Using update objects to update a dataset

TIBUpdateSQL is an update component that uses SQL statements to update a dataset. You must provide one *TIBUpdateSQL* component for each underlying table accessed by the original query that you want to update.

Note If you use more than one update component to perform an update operation, you must create an *OnUpdateRecord* event to execute each update component.

An update component actually encapsulates four *TIBQuery* components. Each of these query components perform a single update task. One query component provides an SQL UPDATE statement for modifying existing records; a second query component provides an INSERT statement to add new records to a table; a third component provides a DELETE statement to remove records from a table, and a forth component provides a SELECT statement to refresh the records in a table.

When you place an update component in a data module, you do not see the query components it encapsulates. They are created by the update component at runtime based on four update properties for which you supply SQL statements:

- *ModifySQL* specifies the UPDATE statement.
- *InsertSQL* specifies the INSERT statement.
- *DeleteSQL* specifies the DELETE statement.
- *RefreshSQL* specifies the SELECT statement

At runtime, when the update component is used to apply updates, it:

1. Selects an SQL statement to execute based on the *UpdateKind* parameter automatically generated on a record update event. *UpdateKind* specifies whether the current record is modified, inserted, or deleted.
2. Provides parameter values to the SQL statement.
3. Prepares and executes the SQL statement to perform the specified update.

Specifying the UpdateObject property for a dataset

One or more update objects can be associated with a dataset to be updated. Associate update objects with the update dataset either by setting the dataset component's *UpdateObject* property to the update object or by setting the update object's *DataSet* property to the update dataset. Which method is used depends on whether only one base table in the update dataset is to be updated or multiple tables.

You must use one of these two means of associating update datasets with update objects. Without proper association, the dynamic filling of parameters in the update object's SQL statements cannot occur. Use one association method or the other, but never both.

How an update object is associated with a dataset also determines how the update object is executed. An update object might be executed automatically, without explicit intervention by the application, or it might need to be explicitly executed. If the association is made using the dataset component's *UpdateObject* property, the update object will automatically be executed. If the association is made with the update object's *DataSet* property, you must programmatically execute the update object.

The sections that follow explain the process of associating update objects with update dataset components in greater detail, along with suggestions about when each method should be used and effects on update execution.

Using a single update object

When only one of the base tables referenced in the update dataset needs to be updated, associate an update object with the dataset by setting the dataset component's *UpdateObject* property to the name of the update object.

```
IBQuery1.UpdateObject := UpdateSQL1;
```

The update SQL statements in the update object are automatically executed when the update dataset's *ApplyUpdates* method is called. The update object is invoked for each record that requires updating. Do not call the update object's *ExecSQL* method in a handler for the *OnUpdateRecord* event as this will result in a second attempt to apply each record's update.

If you supply a handler for the dataset's *OnUpdateRecord* event, the minimum action that you need to take in that handler is setting the event handler's *UpdateAction* parameter to *uaApplied*. You may optionally perform data validation, data modification, or other operations like setting parameter values.

Using multiple update objects

When more than one base table referenced in the update dataset needs to be updated, you need to use multiple update objects: one for each base table updated. Because the dataset component's *UpdateObject* only allows one update object to be associated with the dataset, you must associate each update object with the dataset by setting its *DataSet* property to the name of the dataset. The *DataSet* property for update objects is not available at design time in the Object Inspector. You can only set this property at runtime.

```
IBUpdateSQL1.DataSet := IBQuery1;
```

The update SQL statements in the update object are not automatically executed when the update dataset's *ApplyUpdates* method is called. To update records, you must supply a handler for the dataset component's *OnUpdateRecord* event and call the update object's *ExecSQL* or *Apply* method. This invokes the update object for each record that requires updating.

In the handler for the dataset's *OnUpdateRecord* event, the minimum actions that you need to take in that handler are:

- Calling the update object's *SetParams* method (if you later call *ExecSQL*).
- Executing the update object for the current record with *ExecSQL* or *Apply*.
- Setting the event handler's *UpdateAction* parameter to *uaApplied*.

You may optionally perform data validation, data modification, or other operations that depend on each record's update.

Note It is also possible to have one update object associated with the dataset using the dataset component's *UpdateObject* property, and the second and subsequent update objects associated using their *DataSet* properties. The first update object is executed automatically on calling the dataset component's *ApplyUpdates* method. The rest need to be manually executed.

Creating SQL statements for update components

To update a record in an associated dataset, an update object uses one of three SQL statements. The four SQL statements delete, insert, refresh, and modify records cached for update. The statements are contained in the update object's string list properties *DeleteSQL*, *InsertSQL*, *RefreshSQL*, and *ModifySQL*. As each update object is used to update a single table, the object's update statements each reference the same base table.

As the update for each record is applied, one of the four SQL statements is executed against the base table updated. Which SQL statement is executed depends on the *UpdateKind* parameter automatically generated for each record's update.

Creating the SQL statements for update objects can be done at design time or at runtime. The sections that follow describe the process of creating update SQL statements in greater detail.

Creating SQL statements at design time

To create the SQL statements for an update component,

1. Select the *TIBUpdateSQL* component.
2. Select the name of the update component from the drop-down list for the dataset component's *UpdateObject* property in the Object Inspector. This step ensures that the Update SQL editor you invoke in the next step can determine suitable default values to use for SQL generation options.
3. Right-click the update component and select UpdateSQL Editor from the context menu to invoke the Update SQL editor. The editor creates SQL statements for the update component's *ModifySQL*, *RefreshSQL*, *InsertSQL*, and *DeleteSQL* properties based on the underlying data set and on the values you supply to it.

The Update SQL editor has two pages. The Options page is visible when you first invoke the editor. Use the Table Name combo box to select the table to update. When you specify a table name, the Key Fields and Update Fields list boxes are populated with available columns.

The Update Fields list box indicates which columns should be updated. When you first specify a table, all columns in the Update Fields list box are selected for inclusion. You can multi-select fields as desired.

The Key Fields list box is used to specify the columns to use as keys during the update. Instead of setting Key Fields you can click the Primary Keys button to choose key fields for the update based on the table's primary index. Click Dataset Defaults to return the selection lists to the original state: all fields selected as keys and all selected for update.

Check the Quote Field Names check box if your server requires quotation marks around field names.

After you specify a table, select key columns, and select update columns, click Generate SQL to generate the preliminary SQL statements to associate with the update component's *ModifySQL*, *InsertSQL*, *RefreshSQL*, and *DeleteSQL* properties. In most cases you may want or need to fine tune the automatically generated SQL statements.

To view and modify the generated SQL statements, select the SQL page. If you have generated SQL statements, then when you select this page, the statement for the *ModifySQL* property is already displayed in the SQL Text memo box. You can edit the statement in the box as desired.

IMPORTANT Keep in mind that generated SQL statements are starting points for creating update statements. You may need to modify these statements to make them execute correctly. For example, when working with data that contains NULL values, you need to modify the WHERE clause to read

```
WHERE field IS NULL
```

rather then using the generated field variable. Test each of the statements directly yourself before accepting them.

Use the Statement Type radio buttons to switch among generated SQL statements and edit them as desired.

To accept the statements and associate them with the update component's SQL properties, click OK.

Understanding parameter substitution in update SQL statements

Update SQL statements use a special form of parameter substitution that enables you to substitute old or new field values in record updates. When the Update SQL editor generates its statements, it determines which field values to use. When you write the update SQL, you specify the field values to use.

When the parameter name matches a column name in the table, the new value in the field in the cached update for the record is automatically used as the value for the parameter. When the parameter name matches a column name prefixed by the string "OLD_", then the old value for the field will be used. For example, in the update SQL statement below, the parameter :LastName is automatically filled with the new field value in the cached update for the inserted record.

```
INSERT INTO Names
(LastName, FirstName, Address, City, State, Zip)
VALUES (:LastName, :FirstName, :Address, :City, :State, :Zip)
```

New field values are typically used in the *InsertSQL* and *ModifySQL* statements. In an update for a modified record, the new field value from the update cache is used by the UPDATE statement to replace the old field value in the base table updated.

In the case of a deleted record, there are no new values, so the *DeleteSQL* property uses the ":OLD_FieldName" syntax. Old field values are also normally used in the WHERE clause of the SQL statement for a modified or deletion update to determine which record to update or delete.

In the WHERE clause of an UPDATE or DELETE update SQL statement, supply at least the minimal number of parameters to uniquely identify the record in the base table that is updated with the cached data. For instance, in a list of customers, using just a customer's last name may not be sufficient to uniquely identify the correct record in the base table; there may be a number of records with "Smith" as the last name. But by using parameters for last name, first name, and phone number could be a distinctive enough combination. Even better would be a unique field value like a customer number.

For more information about old and new value parameter substitution, see "Accessing a field's OldValue, NewValue, and CurValue properties" in the *Delphi 5 Developer's Guide*.

Composing update SQL statements

The *TIBUpdateSQL* component has four properties for updating SQL statements: *DeleteSQL*, *InsertSQL*, *RefreshSQL*, and *ModifySQL*. As the names of the properties imply, these SQL statements delete, insert, refresh, and modify records in the base table.

The *DeleteSQL* property should contain only an SQL statement with the DELETE command. The base table to be updated must be named in the FROM clause. So that the SQL statement only deletes the record in the base table that corresponds to the record deleted in the update cache, use a WHERE clause. In the WHERE clause, use a parameter for one or more fields to uniquely identify the record in the base table that corresponds to the cached update record. If the parameters are named the same as the field and prefixed with "OLD_", the parameters are automatically given the values from the corresponding field from the cached update record. If the parameter are named in any other manner, you must supply the parameter values.

```
DELETE FROM Inventory I
WHERE (I.ItemNo = :OLD_ItemNo)
```

Some tables types might not be able to find the record in the base table when fields used to identify the record contain NULL values. In these cases, the delete update fails for those records. To accommodate this, add a condition for those fields that might contain NULLs using the IS NULL predicate (in addition to a condition for a non-NULL value). For example, when a FirstName field may contain a NULL value:

```
DELETE FROM Names
```

```
WHERE (LastName = :OLD_LastName) AND
  ((FirstName = :OLD_FirstName) OR (FirstName IS NULL))
```

The *InsertSQL* statement should contain only an SQL statement with the INSERT command. The base table to be updated must be named in the INTO clause. In the VALUES clause, supply a comma-separated list of parameters. If the parameters are named the same as the field, the parameters are automatically given the value from the cached update record. If the parameter are named in any other manner, you must supply the parameter values. The list of parameters supplies the values for fields in the newly inserted record. There must be as many value parameters as there are fields listed in the statement.

```
INSERT INTO Inventory
(ItemNo, Amount)
VALUES (:ItemNo, 0)
```

The *RefreshSQL* statement should contain only an SQL statement with the SELECT command. The base table to be updated must be named in the FROM clause. If the parameters are named the same as the field, the parameters are automatically given the value from the cached update record. If the parameter are named in any other manner, you must supply the parameter values.

```
SELECT COUNTRY, CURRENCY
FROM Country
WHERE
  COUNTRY = :COUNTRY and CURRENCY = :CURRENCY
```

The *ModifySQL* statement should contain only an SQL statement with the UPDATE command. The base table to be updated must be named in the FROM clause. Include one or more value assignments in the SET clause. If values in the SET clause assignments are parameters named the same as fields, the parameters are automatically given values from the fields of the same name in the updated record in the cache. You can assign additional field values using other parameters, as long as the parameters are not named the same as any fields and you manually supply the values. As with the *DeleteSQL* statement, supply a WHERE clause to uniquely identify the record in the base table to be updated using parameters named the same as the fields and prefixed with "OLD_". In the update statement below, the parameter :ItemNo is automatically given a value and :Price is not.

```
UPDATE Inventory I
SET I.ItemNo = :ItemNo, Amount = :Price
WHERE (I.ItemNo = :OLD_ItemNo)
```

Considering the above update SQL, take an example case where the application end-user modifies an existing record. The original value for the ItemNo field is 999. In a grid connected to the cached dataset, the end-user changes the ItemNo field value to 123 and Amount to 20. When the ApplyUpdates method is invoked, this SQL statement affects all records in the base table where the ItemNo field is 999, using the old field value in the parameter :OLD_ItemNo. In those records, it changes the ItemNo field value to 123 (using the parameter :ItemNo, the value coming from the grid) and Amount to 20.

Using an update component's Query property

Use the *Query* property of an update component to access one of the update SQL properties *DeleteSQL*, *InsertSQL*, *RefreshSQL*, or *ModifySQL*, such as to set or alter the SQL statement. Use *UpdateKind* constant values as an index into the array of query components. The *Query* property is only accessible at runtime.

The statement below uses the *UpdateKind* constant *ukDelete* with the *Query* property to access the *DeleteSQL* property.

```
with IBUpdateSQL1.Query[ukDelete] do begin
  Clear;
  Add('DELETE FROM Inventory I');
  Add('WHERE (I.ItemNo = :OLD_ItemNo)');
end;
```

Normally, the properties indexed by the *Query* property are set at design time using the Update SQL editor. You might, however, need to access these values at runtime if you are generating a unique update SQL statement for each record and not using parameter binding. The following example generates a unique *Query* property value for each row updated:

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  with IBUpdateSQL1 do begin
    case UpdateKind of
      ukModified:
        begin
          Query[UpdateKind].Text := Format('update emptab set Salary
= %d where EmpNo = %d',
            [EmpAuditSalary.NewValue, EmpAuditEmpNo.OldValue]);
          ExecSQL(UpdateKind);
        end;
      ukInserted:
        {...}
      ukDeleted:
```

```
        {...}
    end;
  end;
  UpdateAction := uaApplied;
end;
```

Note *Query* returns a value of type *TIBDataSetUpdateObject*. To treat this return value as a *TIBUpdateSQL* component, to use properties and methods specific to *TIBUpdateSQL*, typecast the *UpdateObject* property. For example:

```
with (DataSet.UpdateObject as IBUpdateSQL).Query[UpdateKind] do begin
  { perform operations on the statement in DeleteSQL }
end;
```

For an example of using this property, see **"Calling the SetParams method" on page 251**.

▸ *Using the DeleteSQL, InsertSQL, ModifySQL, and RefreshSQL properties*

Use the *DeleteSQL*, *InsertSQL*, *ModifySQL*, and *RefreshSQL* properties to set the update SQL statements for each. These properties are all string list containers. Use the methods of string lists to enter SQL statement lines as items in these properties. Use an integer value as an index to reference a specific line within the property. The *DeleteSQL*, *InsertSQL*, *ModifySQL*, and *RefreshSQL* properties are accessible both at design time and at runtime.

```
with UpdateSQL1.DeleteSQL do begin
  Clear;
  Add('DELETE FROM Inventory I');
  Add('WHERE (I.ItemNo = :OLD_ItemNo)');
end;
```

Below, the third line of an SQL statement is altered using an index of 2 with the *ModifySQL* property.

```
UpdateSQL1.ModifySQL[2] := 'WHERE ItemNo = :ItemNo';
```

Executing update statements

There are a number of methods involved in executing the update SQL for an individual record update. These method calls are typically used within a handler for the *OnUpdateRecord* event of the update object to execute the update SQL to apply the current cached update record. The various methods are applicable under different circumstances. The sections that follow discuss each of the methods in detail.

Calling the Apply method

The *Apply* method for an update component manually applies updates for the current record. There are two steps involved in this process:

1. Values for the record are bound to the parameters in the appropriate update SQL statement.
2. The SQL statement is executed.

Call the *Apply* method to apply the update for the current record in the update cache. Only use *Apply* when the update object is not associated with the dataset using the dataset component's *UpdateObject* property, in which case the update object is not automatically executed. *Apply* automatically calls the *SetParams* method to bind old and new field values to specially named parameters in the update SQL statement. Do not call *SetParams* yourself when using *Apply*. The *Apply* method is most often called from within a handler for the dataset's *OnUpdateRecord* event.

If you use the dataset component's UpdateObject property to associate dataset and update object, this method is called automatically. Do not call *Apply* in a handler for the dataset component's *OnUpdateRecord* event as this will result in a second attempt to apply the current record's update.

In a handler for the *OnUpdateRecord* event, the *UpdateKind* parameter is used to determine which update SQL statement to use. If invoked by the associated dataset, the *UpdateKind* is set automatically. If you invoke the method in an *OnUpdateRecord* event, pass an *UpdateKind* constant as the parameter of *Apply*.

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  IBUpdateSQL1.Apply(UpdateKind);
  UpdateAction := uaApplied;
end;
```

If an exception is raised during the execution of the update program, execution continues in the *OnUpdateError* event, if it is defined.

Note The operations performed by *Apply* are analogous to the *SetParams* and *ExecSQL* methods described in the following sections.

Calling the SetParams method

The *SetParams* method for an update component uses special parameter substitution rules to substitute old and new field values into the update SQL statement. Ordinarily, *SetParams* is called automatically by the update component's *Apply* method. If you call *Apply* directly in an *OnUpdateRecord* event, do not call *SetParams* yourself. If you execute an update object using its *ExecSQL* method, call *SetParams* to bind values to the update statement's parameters.

SetParams sets the parameters of the SQL statement indicated by the *UpdateKind* parameter. Only those parameters that use a special naming convention automatically have a value assigned. If the parameter has the same name as a field or the same name as a field prefixed with "OLD_" the parameter is automatically a value. Parameters named in other ways must be manually assigned values. For more information see the section **"Understanding parameter substitution in update SQL statements" on page 245**.

The following example illustrates one such use of *SetParams*:

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  with DataSet.UpdateObject as TIBUpdateSQL do begin
    SetParams(UpdateKind);
    if UpdateKind = ukModified then
      IBQuery[UpdateKind].ParamByName('DateChanged').Value := Now;
    ExecSQL(UpdateKind);
  end;
  UpdateAction := uaApplied;
end;
```

This example assumes that the *ModifySQL* property for the update component is as follows:

```
UPDATE EmpAudit
SET EmpNo = :EmpNo, Salary = :Salary, Changed = :DateChanged
WHERE EmpNo = :OLD_EmpNo
```

In this example, the call to *SetParams* supplies values to the *EmpNo* and *Salary* parameters. The *DateChanged* parameter is not set because the name does not match the name of a field in the dataset, so the next line of code sets this value explicitly.

Calling the ExecSQL method

The *ExecSQL* method for an update component manually applies updates for the current record. There are two steps involved in this process:

1. Values for the record are bound to the parameters in the appropriate update SQL statement.
2. The SQL statement is executed.

Call the *ExecSQL* method to apply the update for the current record in the update cache. Only use *ExecSQL* when the update object is not associated with the dataset using the dataset component's *UpdateObject* property, in which case the update object is not automatically executed. *ExecSQL* does not automatically call the *SetParams* method to bind update SQL statement parameter values; call *SetParams* yourself before invoking *ExecSQL*. The *ExecSQL* method is most often called from within a handler for the dataset's *OnUpdateRecord* event.

If you use the dataset component's *UpdateObject* property to associate dataset and update object, this method is called automatically. Do not call *ExecSQL* in a handler for the dataset component's *OnUpdateRecord* event as this will result in a second attempt to apply the current record's update.

In a handler for the *OnUpdateRecord* event, the *UpdateKind* parameter is used to determine which update SQL statement to use. If invoked by the associated dataset, the *UpdateKind* is set automatically. If you invoke the method in an *OnUpdateRecord* event, pass an *UpdateKind* constant as the parameter of *ExecSQL*.

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  with (DataSet.UpdateObject as TIBUpdateSQL) do begin
    SetParams(UpdateKind);
    ExecSQL(UpdateKind);
  end;
  UpdateAction := uaApplied;
end;
```

If an exception is raised during the execution of the update program, execution continues in the *OnUpdateError* event, if it is defined.

Note The operations performed by *ExecSQL* and *SetParams* are analogous to the *Apply* method described previously.

Using dataset components to update a dataset

Applying cached updates usually involves use of one or more update objects. The update SQL statements for these objects apply the data changes to the base table. Using update components is the easiest way to update a dataset, but it is not a requirement. You can alternately use dataset components like *TIBTable* and *TIBQuery* to apply the cached updates.

In a handler for the dataset component's OnUpdateRecord event, use the properties and methods of another dataset component to apply the cached updates for each record.

For example, the following code uses a table component to perform updates:

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  if UpdateKind = ukInsert then
    UpdateTable.AppendRecord([DataSet.Fields[0].NewValue,
DataSet.Fields[1].NewValue])
  else
    if UpdateTable.Locate('KeyField',
VarToStr(DataSet.Fields[1].OldValue), []) then
      case UpdateKind of
        ukModify:
          begin
            Edit;
            UpdateTable.Fields[1].AsString :=
VarToStr(DataSet.Fields[1].NewValue);
            Post;
          end;
        ukInsert:
          begin
            Insert;
            UpdateTable.Fields[1].AsString :=
VarToStr(DataSet.Fields[1].NewValue);
            Post;
          end;
        ukModify: DeleteRecord;
      end;
        UpdateAction := uaApplied;
end;
```

Updating a read-only result set

To manually update a read-only dataset:

1. Add a *TIBUpdateSQL* component to the data module in your application.
2. Set the dataset component's *UpdateObject* property to the name of the *TIBUpdateSQL* component in the data module.
3. Enter the SQL update statement for the result set to the update component's *ModifySQL*, *InsertSQL*, *DeleteSQL*, or *RefreshSQL* properties, or use the Update SQL editor.
4. Close the dataset.
5. Set the dataset component's *CachedUpdates* property to *True*.
6. Reopen the dataset.

In many circumstances, you may also want to write an *OnUpdateRecord* event handler for the dataset.

Controlling the update process

When a dataset component's *ApplyUpdates* method is called, an attempt is made to apply the updates for all records in the update cache to the corresponding records in the base table. As the update for each changed, deleted, or newly inserted record is about to be applied, the dataset component's *OnUpdateRecord* event fires.

Providing a handler for the *OnUpdateRecord* event allows you to perform actions just before the current record's update is actually applied. Such actions can include special data validation, updating other tables, or executing multiple update objects. A handler for the *OnUpdateRecord* event affords you greater control over the update process.

The sections that follow describe when you might need to provide a handler for the *OnUpdateRecord* event and how to create a handler for this event.

Determining if you need to control the updating process

Some of the time when you use cached updates, all you need to do is call *ApplyUpdates* to apply cached changes to the base tables in the database. In most other cases, however, you either might want to or must provide additional processing to ensure that updates can be properly applied. Use a handler for the updated dataset component's *OnUpdateRecord* event to provide this additional processing.

For example, you might want to use the *OnUpdateRecord* event to provide validation routines that adjust data before it is applied to the table, or you might want to use the *OnUpdateRecord* event to provide additional processing for records in master and detail tables before writing them to the base tables.

In many cases you must provide additional processing. For example, if you access multiple tables using a joined query, then you must provide one *TIBUpdateSQL* object for each table in the query, and you must use the *OnUpdateRecord* event to make sure each update object is executed to write changes to the tables.

The following sections describe how to create and use an *TIBUpdateSQL* object and how to create and use an *OnUpdateRecord* event.

Creating an OnUpdateRecord event handler

The *OnUpdateRecord* event handles cases where a single update component cannot be used to perform the required updates, or when your application needs more control over special parameter substitution. The *OnUpdateRecord* event fires once for the attempt to apply the changes for each modified record in the update cache.

To create an *OnUpdateRecord* event handler for a dataset:

1. Select the dataset component.
2. Choose the Events page in the Object Inspector.
3. Double-click the *OnUpdateRecord* property value to invoke the code editor.

Here is the skeleton code for an *OnUpdateRecord* event handler:

```
procedure TForm1.DataSetUpdateRecord(DataSet: TDataSet;
 UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  { perform updates here... }
end;
```

The *DataSet* parameter specifies the cached dataset with updates.

The *UpdateKind* parameter indicates the type of update to perform. Values for *UpdateKind* are *ukModify*, *ukInsert*, and *ukDelete*. When using an update component, you need to pass this parameter to its execution and parameter binding methods. For example using *ukModify* with the *Apply* method executes the update object's *ModifySQL* statement. You may also need to inspect this parameter if your handler performs any special processing based on the kind of update to perform.

The *UpdateAction* parameter indicates if you applied an update or not. Values for *UpdateAction* are *uaFail* (the default), *uaAbort*, *uaSkip*, *uaRetry*, *uaApplied*. Unless you encounter a problem during updating, your event handler should set this parameter to *uaApplied* before exiting. If you decide not to update a particular record, set the value to *uaSkip* to preserve unapplied changes in the cache.

If you do not change the value for *UpdateAction*, the entire update operation for the dataset is aborted. For more information about *UpdateAction*, see **"Specifying the action to take" on page 259**.

In addition to these parameters, you will typically want to make use of the *OldValue* and *NewValue* properties for the field component associated with the current record. For more information about *OldValue* and *NewValue* see "Accessing a field's OldValue, NewValue, and CurValue properties" in the *Delphi 5 Developer's Guide*.

IMPORTANT The *OnUpdateRecord* event, like the *OnUpdateError* and *OnCalcFields* event handlers, should never call any methods that change which record in a dataset is the current record.

Here is an *OnUpdateRecord* event handler that executes two update components using their *Apply* methods. The *UpdateKind* parameter is passed to the *Apply* method to determine which update SQL statement in each update object to execute.

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  EmployeeUpdateSQL.Apply(UpdateKind);
  JobUpdateSQL.Apply(UpdateKind);
  UpdateAction := uaApplied;
end;
```

In this example the *DataSet* parameter is not used. This is because the update components are not associated with the dataset component using its *UpdateObject* property.

Handling cached update errors

Because there is a delay between the time a record is first cached and the time cached updates are applied, there is a possibility that another application may change the record in a database before your application applies its updates. Even if there is no conflict between user updates, errors can occur when a record's update is applied.

A dataset component's *OnUpdateError* event enables you to catch and respond to errors. You should create a handler for this event if you use cached updates. If you do not, and an error occurs, the entire update operation fails.

IMPORTANT Do not call any dataset methods that change the current record (such as *Next* and *Prior*) in an *OnUpdateError* event handler. Doing so causes the event handler to enter an endless loop.

Here is the skeleton code for an *OnUpdateError* event handler:

```
procedure TForm1.DataSetUpdateError(DataSet: TDataSet; E:
EDatabaseError;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  { ... perform update error handling here ... }
end;
```

The following sections describe specific aspects of error handling using an *OnUpdateError* handler, and how the event's parameters are used.

Referencing the dataset to which to apply updates

DataSet references the dataset to which updates are applied. To process new and old record values during error handling you must supply this reference.

Indicating the type of update that generated an error

The *OnUpdateRecord* event receives the parameter *UpdateKind*, which is of type *TUpdateKind*. It describes the type of update that generated the error. Unless your error handler takes special actions based on the type of update being carried out, your code probably will not make use of this parameter.

The following table lists possible values for *UpdateKind:*

Value	Meaning
ukModify	Editing an existing record caused an error
ukInsert	Inserting a new record caused an error
ukDelete	Deleting an existing record caused an error

TABLE 16.3 UpdateKind values

The example below shows the decision construct to perform different operations based on the value of the *UpdateKind* parameter.

```
procedure TForm1.DataSetUpdateError(DataSet: TDataSet; E:
EDatabaseError;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  case UpdateKind of
    ukModify:
      begin
        { handle error due to applying record modification update }
      end;
    ukInsert:
      begin
        { handle error due to applying record insertion update }
      end;
    ukDelete:
      begin
        { handle error due to applying record deletion update }
      end;
  end;
end;
```

Specifying the action to take

UpdateAction is a parameter of type *TUpdateAction*. When your update error handler is first called, the value for this parameter is always set to *uaFail*. Based on the error condition for the record that caused the error and what you do to correct it, you typically set *UpdateAction* to a different value before exiting the handler. *UpdateAction* can be set to one of the following values:

Value	Meaning
uaAbort	Aborts the update operation without displaying an error message
uaFail	Aborts the update operation, and displays an error message; this is the default value for *UpdateAction* when you enter an update error handler
uaSkip	Skips updating the row, but leaves the update for the record in the cache
uaRetry	Repeats the update operation; correct the error condition before setting *UpdateAction* to this value
uaApplied	Not used in error handling routines

TABLE 16.4 UpdateAction values

If your error handler can correct the error condition that caused the handler to be invoked, set *UpdateAction* to the appropriate action to take on exit. For error conditions you correct, set *UpdateAction* to *uaRetry* to apply the update for the record again.

When set to *uaSkip*, the update for the row that caused the error is skipped, and the update for the record remains in the cache after all other updates are completed.

Both *uaFail* and *uaAbort* cause the entire update operation to end. *uaFail* raises an exception, and displays an error message. *uaAbort* raises a silent exception (does not display an error message).

Note If an error occurs during the application of cached updates, an exception is *raised* and an error message displayed. Unless the *ApplyUpdates* is called from within a `try...except` construct, an error message to the user displayed from inside your *OnUpdateError* event handler may cause your application to display the same error message twice. To prevent error message duplication, set *UpdateAction* to *uaAbort* to turn off the system-generated error message display.

The *uaApplied* value should only be used inside an *OnUpdateRecord* event. Do not set this value in an update error handler. For more information about update record events, see **Creating an OnUpdateRecord event handler on page 255**.

CHAPTER

17

Debugging with SQL Monitor

Use the *TIBSQLMonitor* component to monitor the dynamic SQL that passes through the InterBase server. You can write an application that can view only its own SQL statements, or you can write a generic SQL monitor application that monitors the dynamic SQL of all applications built with InterBase Express (IBX).

Use the *TIBSQLMonitor* component to watch dynamic SQL taking place in all InterBase data access applications both before and after they have been compiled.

SQL monitoring involves a bit of overhead, so you should be aware of the following:

- If no SQL monitors are loaded, there is little to no overhead
- SQL monitoring can be switched off globally by an application to ensure that it does not get bogged down during debugging
- Disabling monitoring in an application that does not require it further reduces the overhead

Building a simple monitoring application

To build a simple SQL monitoring application, follow these steps:

1. Open a new form in Delphi.
2. Add a *Memo* component to the form and clear the *Lines* property.
3. Add a *TIBSQLMonitor* component to the form
4. Double-click the *OnSQL* event and add the following line of code:

```
Memo1.Lines.Add(EventText);
```

5. Compile the application.

You can now start another IBX application and monitor the code.

Disabling monitoring on a single application

[to come]

Globally disabling monitoring

[to come]

CHAPTER

18

Importing and Exporting Data

InterBase Express (IBX) provides a convenient means to migrate data to and from the database. The *TIBSQL* component, along with the *TIBBatchInput* and *TIBBatchOutput* objects make it possible to import and export data to and from databases in virtually any format.

Descendents of this class can specify a file name (for input or output), and a *TIBXSQLDA* object representing a record or parameters. The *ReadyFile* method is called right before performing the batch input or output.

Exporting and importing raw data

Use the *TIBSQL* component, along with the *TIBOutputRawFile* and *TIBInputRawFile* objects to perform batch imports and exports of raw data. A raw file is the equivalent of InterBase external file output. Raw files are not limited to straight character format, so whatever structure is defined by your query is what goes in the file.

Use an SQL SELECT statement to export the data to the raw file, and an INSERT statement to import the raw data into another database.

Raw files are probably the fastest way, aside from external tables, to get data in and out of an InterBase database, although dealing with fixed-width files can be somewhat difficult.

Exporting raw data

To export raw data, you will need *TIBSQL*, *TIBDatabase*, and *TIBTransaction* components. Associate the components with each other, select a source database, and set the connections to active.

Tip Use the Database Editor to set up the database component. To start the Database Editor, right click the database component with the mouse and select Database Editor from the drop down menu.

The following code snippet outputs selected data with an SQL SELECT statement from the SOURCE table to the file **source_raw**.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  RawOutput : TIBOutputRawFile;
begin
  IBSQL1.SQL.Text := 'Select name, number, hired from Source';
  RawOutput := TIBOutputRawFile.Create;
  try
    RawOutput.Filename := 'source_raw';
    IBSQL1.BatchOutput(RawOutput);
  finally
   RawOutput.Free;
  end;
end;
```

Importing raw data

To import raw data, you will need *TIBSQL*, *TIBDatabase*, and *TIBTransaction* components. Associate the components with each other, select a destination database, and set the connections to active.

Tip Use the Database Editor to set up the database component. To start the Database Editor, right click the database component with the mouse and select Database Editor from the drop down menu.

It is important to note that you must import data into a table with the same column definitions and datatypes, and in the same order; otherwise, all sorts of unpredictable and undesirable results may occur.

The following code snippet inputs selected data with an SQL INSERT statement from the **source_raw** file created in the last example into the DESTINATION table.

```
procedure TForm1.Button2Click(Sender: TObject);
var
  RawInput : TIBInputRawFile;
begin
  IBSQL2.SQL.Text := 'Insert into Destination values(:name, :number,
:hired)';
  RawInput := TIBInputRawFile.Create;
  try
    RawInput.Filename := 'source_raw';
    IBSQL2.BatchInput(RawInput);
  finally
    RawInput.Free;
  end;
end;
```

Exporting and importing delimited data

Use the *TIBSQL* component, along with *TIBOutputDelimitedFile* and *TIBInputDelimitedFile* objects to perform batch exports and imports of data to and from a database into pipe-tilde (|~) and Tab-Ctrl-F delimited files.

Use an SQL SELECT statement to export the data to the delimited file, and an INSERT statement to import the delimited data into another database.

By default, the column delimiter is a tab, and the row delimiter is a tab-line feed (Tab-Ctrl-F). Use the *ColDelimiter* and *RowDelimiter* properties to change the column delimiter and row delimiter, respectively.

For example, to set the column delimiter to a comma, you could use the following line of code:

```
DelimOutput.ColDelimiter := ',';
```

Note Columns may contain spaces before the delimiter. For example, if you have a column called NAME which is defined as a CHAR(10), and the name "Joe" is in that column, then "Joe" will be followed by 7 spaces before the column is delimited.

Exporting delimited data

To export delimited data, you will need *TIBSQL*, *TIBDatabase*, and *TIBTransaction* components. Set up the database component, and associate the components with each other. In the following example, the database and transaction components are set to active in the code.

Tip Use the Database Editor to set up the database component. To start the Database Editor, right click the database component with the mouse and select Database Editor from the drop down menu.

The following code snippet outputs selected data with an SQL SELECT statement from the SOURCE table to the file **source_delim**.

```
procedure TForm1.Button3Click(Sender: TObject);
var
  DelimOutput : TIBOutputDelimitedFile;
begin
  IBSQL3.Database.Open;
  IBSQL3.Transaction.StartTransaction;
  IBSQL3.SQL.Text := 'Select name, number, hired from Source';
  DelimOutput := TIBOutputDelimitedFile.Create;
  try
    DelimOutput.Filename := 'source_delim';
    IBSQL3.BatchOutput(DelimOutput);
  finally
    DelimOutput.Free;
    IBSQL3.Transaction.Commit;
  end;
end;
```

Importing delimited data

To import delimited data, you will need *TIBSQL*, *TIBDatabase*, and *TIBTransaction* components.et up the database component, and associate the components with each other. In the following example, the database and transaction components are set to active in the code.

Tip Use the Database Editor to set up the database component. To start the Database Editor, right click the database component with the mouse and select Database Editor from the drop down menu

It is important to note that you must import data into a table with the same column definitions and datatypes, and in the same order; otherwise, all sorts of unpredictable and undesirable results may occur.

The following code snippet inputs selected data with an SQL INSERT statement from the **source_delim** file created in the last example into the DESTINATION table.

```
procedure TForm1.Button4Click(Sender: TObject);
var
  DelimInput : TIBInputDelimitedFile;
begin
  IBSQL4.Database.Open;
  IBSQL4.Transaction.StartTransaction;
  IBSQL4.SQL.Text := 'Insert into Destination values(:name, :number,
:hired)';
  DelimInput := TIBInputDelimitedFile.Create;
  try
    DelimInput.Filename := 'source_delim';
    IBSQL4.BatchInput(DelimInput);
  finally
    DelimInput.Free;
    IBSQL4.Transaction.Commit;
  end;
end;
```

CHAPTER

19

Working with InterBase Services

InterBase Express (IBX) comes with a set of service components (located on the InterBase Admin page of the Component palette), which allow you to build InterBase database and server administration tools directly into your application.

This chapter shows you how to build all of the InterBase database services into your applications, including:

- Configuration
- Backup and Restore
- Licensing
- Security
- Validation
- Statistics
- Log
- Server properties

Overview of the InterBase service components

This section describes the general concepts of the InterBase service components and methods for attaching and detaching from a services manager.

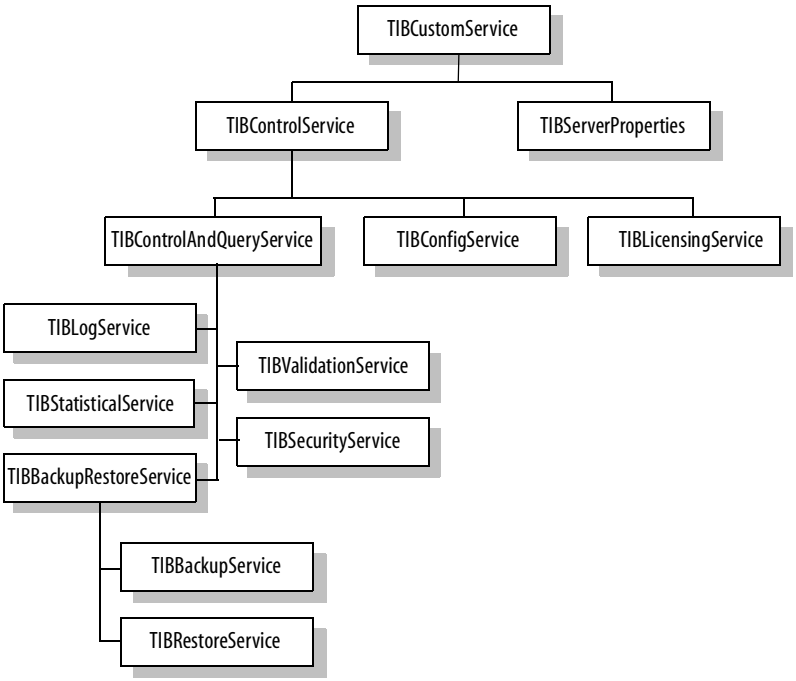
About the services manager

All InterBase servers include a facility called the *services manager*. The InterBase service components enable client applications to submit requests to the services manager of an InterBase server, and the service manager performs the tasks. the server can be local (on the same host as your application), or remote (on another host on the network). The services components offer the same features when connected to either local or remote InterBase servers.

Service component hierarchy

The root object of the InterBase service components is *TIBCustomService*, from which descend *TIBControlService* and *TIBServerProperties*. *TIBServerProperties* contains properties and methods specific to server configuration, while *TIBControlService* is the ancestor object from which all the database configuration and administration components descend.

FIGURE 19.1 InterBase service component hierarchy



The following three components descend directly from *TIBControlService*:

- *TIBControlAndQueryService* contains all the database administration elements, such as monitoring, maintenance, and backup and restore, as well as all of the user validation and security elements.
- *TIBConfigService* contains all the methods and properties for database configuration.
- *TIBLicensingService* contains all the properties and methods to add and remove database licenses.

Attaching to a service manager

To initiate a connection from your application to an InterBase service manager:

1. Place a service component on a form.
2. Set the *ServerName* property for that component to the name of the server on which the services are to be run.

3. Use the *Protocol* property to set the network protocol with which to connect to the server.
4. Set the *Active* property to *True*. A login dialog is displayed. If you do not wish to display the login dialog, set the user name and password in the *Params* string editor, and set *LoginPrompt* to *False*.

To start the service, use the *ServiceStart* method.

Note *TIBLicensingService* and *TIBSecurityService* do not require that you start the service using the *ServiceStart* method. For example, to add a license you could use:

```
Action := LicenseAdd;
ServiceStart;
```

or you could use:

```
AddLicense;
```

Detaching from a service manager

After you finish your tasks with the services components, you should end the connection with the service manager by setting the *Active* property to *False*. This calls the *Detach* method which detaches the service component from the service manager.

Setting database properties

The configuration service component, *TIBConfigService* allows the SYSDBA user to attach to an InterBase database server and configure its behavior, including:

- **Bringing a database online**
- **Shutting down a database**
- **Setting the sweep interval**
- **Setting the async mode**
- **Setting the page buffers**
- **Setting the access mode**
- **Setting the database reserve space**
- **Activating the database shadow**

Bringing a database online

Use the *BringDatabaseOnline* method of the *TIBConfigService* component to bring a database back online.

For example, you could associate the *BringDatabaseOnline* method to a menu item:

```
procedure TForm1.BringDatabaseOnline1Click(Sender: TObject);
begin
 with IBConfigService1 do
  begin
    BringDatabaseOnline;
  end;
end;
```

For more information, refer to "Restarting a database" in the *InterBase 6 Operations Guide*.

Shutting down a database

Use the *ShutdownDatabase* method of the *TIBConfigService* component to shut down the database (or perform an action of type *TShutdownMode* and shut down the database) after a specified number of seconds.

The database shutdown options are:

Shutdown Mode	Meaning
Forced	Shut down the database after the specified number of seconds; to shut down the database immediately, set the shutdown interval to 0
DenyTransaction	Deny new transactions and shut down the database after the specified number of seconds; if transactions are active after the shutdown interval has expired, the shutdown will fail; to shut down the database immediately, set the shutdown interval to 0
DenyAttachment	Deny new attachments and shut down the database after the specified number of seconds; if attachments are active after the shutdown interval has expired, the shutdown will fail; to shut down the database immediately, set the shutdown interval to 0

TABLE 19.1 Database shutdown modes

For example, you could use radio buttons to select the shut down mode and an *Edit* component to specify the number of seconds before shutting down a database:

```
if RadioButton1.Checked then
  ShutdownDatabase(Forced, (StrToInt(Edit4.Text)));
if RadioButton2.Checked then
  ShutdownDatabase(DenyTransaction,(StrToInt(Edit4.Text)));
if RadioButton3.Checked then
  ShutdownDatabase(DenyAttachment,(StrToInt(Edit4.Text)));
```

For more information, refer to "Database shutdown and restart" in the *InterBase 6 Operations Guide*.

Setting the sweep interval

Use the *SetSweepInterval* method of the *TIBConfigService* component to set the database sweep interval. The sweep interval refers to the number of transactions between database sweeps. To turn off database sweeps, set the sweep interval to 0.

For example, you could set up an application that allows a user to set the sweep interval in an *Edit* component:

```
procedure TDBConfigForm.Button1Click(Sender: TObject);
begin
  with IBConfigService1 do
  begin
    SetSweepInterval(StrtoInt(Edit1.Text));
  end;
end;
```

For more information, refer to "Sweep interval and automated housekeeping" in the *InterBase 6 Operations Guide*.

Setting the async mode

InterBase 6 allows you to write to databases in both synchronous and asynchronous modes. In synchronous mode, the database writes are forced. In asynchronous mode, the database writes are buffered.

Set the *SetAsyncMode* method of the *IBConfigService* component to *True* to set the database write mode to asynchronous.

```
procedure TDBConfigForm.CheckBox2Click(Sender: TObject);
begin
  with IBConfigService1 do
  begin
    SetAsyncMode(True);
  end;
end;
```

For more information, refer to "Forced writes vs. buffered writes" in the *InterBase 6 Operations Guide*.

Setting the page buffers

The *SetPageBuffers* method of the *IBConfigService* component lets you set the number of database page buffers. For example, you could set up an application that allows a user to set the number of page buffers in an *Edit* component:

```
procedure TDBConfigForm.Button1Click(Sender: TObject);
begin
  with IBConfigService1 do
  begin
    SetPageBuffers(StrtoInt(Edit2.Text));
  end;
end;
```

For more information on page buffers, refer to "Default cache size per database" in the *InterBase 6 Operations Guide*.

Setting the access mode

Set the *SetReadOnly* method of the *IBConfigService* component to *True* to set the database access mode to read-only.

```
procedure TDBConfigForm.CheckBox1Click(Sender: TObject);
begin
  with IBConfigService1 do
  begin
    SetReadOnly(True);
  end;
```

```
end;
```

Note Once you set the database to read-only, you will be unable to change any of the other database options until you set *SetReadOnly* method to *False* again.

For more information on access mode, refer to "Read-only databases" in the *InterBase 6 Operations Guide*.

Setting the database reserve space

Use the *SetReserveSpace* method of the *IBConfigService* component to reserve space on the data page for versioning.

```
procedure TDBConfigForm.CheckBox3Click(Sender: TObject);
begin
  with IBConfigService1 do
  begin
    SetReserveSpace(True);
  end;
end;
```

Activating the database shadow

The *ActivateShadow* method of the *IBConfigService* component lets you activate a shadow file for database use.

For example, you could associate the *ActivateShadow* method to a button:

```
procedure TDBConfigForm.Button2Click(Sender: TObject);
begin
  with IBConfigService1 do
  begin
    ActivateShadow;
  end;
end;
```

For more information, see "Shadowing" in the *InterBase 6 Operations Guide*.

Backing up and restoring databases

IBX comes with both Backup and Restore services: *TIBBackupService* and *TIBRestoreService*, respectively. These are discussed in **"Backing up databases"** and **"Restoring databases."**

For more information on backup and restore, refer to "Database backup and restore" in the *InterBase 6 Operations Guide*.

Setting common backup and restore properties

TIBBackupService and *TIBRestoreService* descend from a common ancestor, which contains the following properties :

Property	Meaning
BackupFile	The path of the backup file name
BackupFileLength	The length in pages of the restored database file; must exceed 2 gigabytes; you must supply a length for each database file except the last
DatabaseName	Path of the primary file of the database from the server's point of view; you can specify multiple database files
Verbose	If set to *True*, displays backup or restore information in verbose mode
BufferSize	The number of default cache buffers to configure for attachments to the restored database

TABLE 19.2 Common backup and restore properties

Backing up databases

TIBBackupService contains many properties and methods to allow you to build a backup component into your application. Only the SYSDBA user or the database owner will be able to perform backup operations on a database.

When backing up a database under normal circumstances, the backup file will always be on the local server since the backup service cannot open a file over a network connection. However, *TIBBackupService* can create a remote file in one of the following is true:

- The server is running on Windows NT, the path to the backup file is specified as an UNC name, and the destination for the file is another Windows NT machine (or a machine which can be connected to via UNC naming conventions).
- The destination drive is mounted via NFS (or some equivalent) on the machine running the InterBase server.

Setting the backup options

The *Options* property allows you to build backup options of type *TBackupOption* into your application. The backup options are:

Option	Meaning
IgnoreChecksums	Ignore checksums during backup
IgnoreLimbo	Ignored limbo transactions during backup
MetadataOnly	Output backup file for metadata only with empty tables
NoGarbageCollect	Suppress normal garbage collection during backup; improves performance on some databases
OldMetadataDesc	Output metadata in pre-4.0 format
NonTransportable	Output backup file with non-XDR data format; improves space and performance by a negligible amount
ConvertExtTables	Convert external table data to internal tables

TABLE 19.3 *TIBBackupService* options

Displaying backup output

Set the *Verbose* property to *True* to display the backup output to a data control, such as a *Memo* component.

Setting up a backup component

To set up a simple backup component:

1. Drop a *TIBBackupService* component on a Delphi form.
2. Drop *Button* and *Memo* components on the form.

3. Enter the name and path of the database to be backed up in the *DatabaseName* field and the name and path of the database backup file in the *BackupFile* string field of the Object Inspector, or double click on the button and set the properties in code:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
with IBBackupService1 do
  begin
   DatabaseName := 'd:\temp\examples\database\employee.gdb';
   BackupFile.Add('d:\temp\employee1.gbk');
  end;
end;
```

4. Attach to the service manager as described in **"Attaching to a service manager" on page 271**, or set the properties in code:

```
with IBBackupService1 do
begin
  ServerName := 'Poulet';
  LoginPrompt := False;
  Params.Add('user_name=sysdba');
  Params.Add('password=masterkey');
  Active := True;
end;
```

5. Set any other options in the Object inspector (or set them in code), and then start the service with the *ServiceStart* method.

The final code for a backup application that displays verbose backup output in a *Memo* component might look like this:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with IBBackupService1 do
  begin
    ServerName := 'Poulet';
    LoginPrompt := False;
    Params.Add('user_name=sysdba');
    Params.Add('password=masterkey');
    Active := True;
    try
      verbose := True;
      Options := [NonTransportable, IgnoreLimbo];
      DatabaseName := 'd:\interbase\examples\database\employee.gdb';
      BackupFile.Add('d:\temp\employee1.gbk');
      ServiceStart;
      While not Eof do
        Memo1.Lines.Add(GetNextLine);
    finally
      Active := False;
    end;
  end;
end;
```

Backing up a database to multiple files

InterBase allows you to back up a database to multiple files. To do this, you must specify the size of each backup file except for the last, which will hold the remaining information.

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  with IBBackupService1 do
  begin
    ServerName := 'Poulet';
    LoginPrompt := False;
    Params.Add('user_name=sysdba');
    Params.Add('password=masterkey');
    Active := True;
    try
      Verbose := True;
      Options := [MetadataOnly, NoGarbageCollection];
```

```
      DatabaseName := 'c:\interbase\examples\database\employee.gdb';
      BackupFile.Add('c:\temp\e1.gbk = 2048');
      BackupFile.Add('c:\temp\e2.gbk' = 4096);
      BackupFile.Add('c:\temp\e3.gbk'); ServiceStart;
      While not Eof do
        Memo1.Lines.Add(GetNextLine);
    finally
      Active := False;
    end;
  end;
end;
```

Restoring databases

TIBRestoreService contains many properties and methods to allow you to build a restore component into your application. Only the SYSDBA user or the database owner may use the *TIBRestoreService* to overwrite an existing database.

The username and password used to connect to the *TIBRestoreService* will be used to connect to the database for restore.

Setting the database cache size

Use the *PageBuffers* property to set the cache size for the restored database. The default is 2048 buffer pages in the database cache. To change the database cache size, set it in the Object Inspector or in code:

```
PageBuffers := 3000
```

Setting the page size

InterBase supports database page sizes of 1024, 2048, 4096, and 8192 bytes. By default, the database will be restored with the page size with which it was created. To change the page size, you can set it in the Object Inspector or in code:

```
PageSize := 4096;
```

Changing the page size can improve database performance, depending on the datatype size, row length, and so forth. For a discussion of how page size affects performance, see "Page size" in the *InterBase 6 Operations Guide.*

Setting the restore options

The *Options* property allows you to build restore options of type *TRestoreOption* into your application. The restore options are:

Option	Meaning
DeactivateIndex	Do not build indexes during restore
NoShadow	Do not recreate shadow files during restore
NoValidity	Do not enforce validity conditions (for example, NOT NULL) during restore
OneRelationATime	Commit after completing a restore of each table
Replace	Replace database if one exists
Create	Restore but do not overwrite an existing database
UseAllSpace	Do not reserve 20% of each datapage for future record versions; useful for read-only databases

TABLE 19.4 *TIBRestoreService* options

Displaying restore output

Set the *Verbose* property to *True* to display the restore output to a data control, such as a *Memo* component.

Setting up a restore component

To set up a simple restore component:

1. Drop a *TIBRestoreService* component on a Delphi form.
2. Drop *Button* and *Memo* components on the form.
3. Enter the name and path of the database to be restored in the *DatabaseName* field and the name and path of the database backup file in the *BackupFile* string field of the Object Inspector, or double click on the button and set the properties in code:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
with IBRestoreService1 do
 begin
 DatabaseName.Add('c:\interbase\examples\database\employee.gdb');
 BackupFile.Add('c:\temp\employee1.gbk');
 end;
```

4. Attach to the service manager as described in **"Attaching to a service manager" on page 271**, or set the properties in code:

```
begin
with IBRestoreService1 do
 begin
  ServerName := 'Poulet';
  LoginPrompt := False;
  Params.Add('user_name=sysdba');
  Params.Add('password=masterkey');
  Active := True;
 end;
```

5. Set any other options in the Object inspector (or set them in code), and then start the restore service with the *ServiceStart* method.

The final code for a restore application that displays verbose restore output in a *Memo* component might look like this:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   with IBRestoreService1 do
  begin
    ServerName := 'Poulet';
    LoginPrompt := False;
    Params.Add('user_name=sysdba');
    Params.Add('password=masterkey');
    Active := True;
    try
      Verbose := True;
      Options := [Replace, UseAllSpace];
      PageBuffers := 3000;
      PageSize := 4096;
      DatabaseName.Add('c:\interbase6\tutorial\tutorial.gdb');
      BackupFile.Add('c:\interbase6\tutorial\backups\tutor5.gbk');
      ServiceStart;
      While not Eof do
        Memo1.Lines.Add(GetNextLine);
    finally
      Active := False;
    end;
  end;
end;
```

Restoring a database from multiple backup files

InterBase allows you to restore a database from multiple files. The following code example shows how to do this.

```
procedure TForm1.Button3Click(Sender: TObject);
begin
 with IBRestoreService1 do
  begin
    ServerName := 'Poulet';
    LoginPrompt := False;
    Params.Add('user_name=sysdba');
    Params.Add('password=masterkey');
    Active := True;
    try
      Verbose := True;
      Options := [Replace, UseAllSpace];
      PageBuffers := 3000;
      PageSize := 4096;
      BackupFile.Add('c:\temp\employee1.gbk');
      BackupFile.Add('c:\temp\employee2.gbk');
      BackupFile.Add('c:\temp\employee3.gbk');
     DatabaseName.Add('c:\interbase\examples\database\employee.gdb');
      ServiceStart;
      While not Eof do
        Memo1.Lines.Add(GetNextLine);
    finally
      Active := False;
    end;
  end;
end;
```

Restoring a database to multiple files

You might want to restore a database to multiple files to distribute it among different disks, which provides more flexibility in allocating system resources. The following code example shows how to do this.

```
procedure TForm1.Button2Click(Sender: TObject);
begin
 with IBRestoreService1 do
  begin
    ServerName := 'Poulet';
    LoginPrompt := False;
```

```
    Params.Add('user_name=sysdba');
    Params.Add('password=masterkey');
    Active := True;
    try
      Verbose := True;
      Options := [Replace, UseAllSpace];
      PageBuffers := 3000;
      PageSize := 4096;
      BackupFile.Add('c:\temp\employee1.gbk');
      DatabaseName.Add('c:\temp\employee2.gdb = 2048');
      DatabaseName.Add('c:\temp\employee3.gdb = 2048');
      DatabaseName.Add('c:\temp\employee4.gdb');
      ServiceStart;
      While not Eof do
        Memo1.Lines.Add(GetNextLine);
    finally
      Active := False;
    end;
  end;
end;
```

Performing database maintenance

TIBValidationService contains many properties and methods to allow you to perform database validation and resolve limbo transactions. These are discussed in the following sections. For more information, refer to "Database Configuration and Maintenance" in the *InterBase 6 Operations Guide*.

Validating a database

Use the *Options* property of *TIBValidationService* component to invoke a database validation. Set any of the following options of type *TValidateOption* to *True* to perform the appropriate validation:

Option	Meaning
LimboTransactions	Returns limbo transaction information, including: • Transaction ID • Host site • Remote site • Remote database path • Transaction state • Suggested transaction action • Transaction action • Multiple database information
CheckDB	Request a read-only validation of the database without correcting any problems
IgnoreChecksum	Ignore all checksum errors when validating or sweeping
KillShadows	Remove references to unavailable shadow files
MendDB	Mark corrupted records as unavailable so that subsequent operations skip them
SweepDB	Request database sweep to mark outdated records as free space
ValidateDB	Locate and release pages that are allocated but unassigned to any data structures
ValidateFull	Check record and page structures, releasing unassigned record fragments; use with *ValidateDB*

TABLE 19.5 *TIBValidationService* options

To set these options in code, use the *Options* property:

```
Options := [CheckDB, IgnoreChecksum, KillShadows];
```

Note Not all combinations of validation options work together. For example, you could not simultaneously mend and validate the database at the same time. Conversely, some options are intended to be used with other options, such as *IgnoreChecksum* with *SweepDB* or *ValidateDB*, or *ValidateFull* with *ValidateDB*.

To use the *LimboTransactions* option, see the following section.

Displaying limbo transaction information

Use the *FetchLimboTransaction* method along with the*LimboTransactions* option to retrieve a record of all current limbo transactions. The following code snippet will display the contents of the *TLimboTransactionInfo* record, provided that there are any limbo transactions to display.

```
try
      Options := [LimboTransactions];
      FetchLimboTransactionInfo;
      for I := 0 to LimboTransactionInfoCount - 1 do
      begin
        with LimboTransactionInfo[i] do
        begin
          Memo1.Lines.Add('Transaction ID: ' + IntToStr(ID));
          Memo1.Lines.Add('Host Site: ' + HostSite);
          Memo1.Lines.Add('Remote Site: ' + RemoteSite);
          Memo1.Lines.Add('Remote Database Path: ' +
RemoteDatabasePath);
         //Memo1.Lines.Add('Transaction State: ' + TransactionState);
          Memo1.Lines.Add('---------------------------------');
        end;
      end;
    finally
```

Resolving limbo transactions

You can correct transactions in a limbo state using the *GlobalAction* property of the *TIBValidationService* to perform one of the following actions of type *TTransactionGlobalAction* on the database specified by the *DatabaseName* property:

Action	Meaning
CommitGlobal	Commits the limbo transaction specified by ID or commits all limbo transactions
RollbackGlobal	Rolls back the limbo transaction specified by ID or rolls back all limbo transactions
RecoverTwoPhaseGlobal	Performs automated two-phase recovery, either for a limbo transaction specified by ID or for all limbo transactions
NoGlobalAction	Takes no action.

TABLE 19.6 *TIBValidationService* actions

For example, to set the global action using radio buttons:

```
with IBValidationService1 do
try
  if RadioButton1.Checked then GlobalAction := (CommitGlobal);
  if RadioButton2.Checked then GlobalAction := (RollbackGlobal);
  if RadioButton3.Checked then GlobalAction :=
(RecoverTwoPhaseGlobal);
  if RadioButton4.Checked then GlobalAction := (NoGlobalAction);
```

Requesting database and server status reports

TIBStatisticalService contains many properties and methods to allow you to build a statistical component into your application. Only the SYSDBA user or owner of the database will be able to run this service.

Requesting database statistics

Use the *Options* property of *TIBStatisticalService* to request database statistics. These options are incremental; that is, setting *DbLog* to *True* also returns *HeaderPages* statistics, setting *IndexPages* to *True* returns also returns *DbLog* and *HeaderPages* statistics, and so forth. Set any of the following options of type *TStatOption* to *True* to retrieve the appropriate information:

Option	Meaning
HeaderPages	Stop reporting statistics after reporting the information on the header page
DbLog	Stop reporting statistics after reporting information on the log pages
IndexPages	Request statistics for the user indexes in the database
DataPages	Request statistics for data tables in the database
SystemRelations	Request statistics for system tables and indexes in addition to user tables and indexes

TABLE 19.7 TIBStatisticalService options

To use the statistical service:

1. Drop an *IBStatisticalServices* component on a Delphi form.
2. Attach to the service manager as described in **"Attaching to a service manager" on page 271**.
3. Set the *DatabaseName* property to the path of the database for which you would like statistics.
4. Set the options for which statistics you would like to receive, either by setting them to *True* in the Object Inspector, or in code using the *Options* property.
5. Start the statistical service using the *ServiceStart* method.

The following example displays the statistics for a database. With a button click, *HeaderPages* and *DBLog* statistics are returned until the end of the file is reached.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
with IBStatisticalService1 do
  begin
    ServerName := 'Poulet';
    DatabaseName := 'C:\interbase6\tutorial\tutorial.gdb';
    LoginPrompt := False;
    Params.Add('user_name=sysdba');
    Params.Add('password=masterkey');
    Active := True;
    ServiceStart;
    try
      Options := [DataPages, DBLog];
      While not Eof do
        Memo1.Lines.Add(GetNextLine);
    finally
    Active := False;
    end;
  end;
 end;
```

Using the log service

Use the *TIBLogService* to retrieve the **interbase.log** file, if it exists, from the server. If the log file does not exist, an error is returned.

To use the log service:

1. Drop a *TIBLogService* component on a Delphi application.
2. Drop *Button* and *Memo* components on the same application.
3. Attach to the service manager as described in **"Attaching to a service manager" on page 271**.
4. Start the log service using the *ServiceStart* method.

The following example displays the contents of the **interbase.log** file. With a click of the button, the log file is displayed until the end of the file is reached.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
with IBLogService1 do
  begin
    ServerName := 'Poulet';
    LoginPrompt := False;
    Params.Add('user_name=sysdba');
    Params.Add('password=masterkey');
    Active := True;
    ServiceStart;
    try
      While not Eof do
        Memo1.Lines.Add(GetNextLine);
    finally
    Active := False;
    end;
  end;
end;
```

Configuring users

Security for InterBase relies on a central database for each server host. This database contains legitimate users who have permission to connect to databases and InterBase services on that host. The database also contains an encrypted password for the user. This user and password applies to any database on that server host.

You can use the *TIBSecurityService* component to list, add, delete, and modify users. These are discussed in the following sections.

For more information on InterBase database security, refer to "DataBase Security" in the *InterBase 6 Operations Guide*.

Adding a user to the security database

Use the *AddUser* method along with the following properties to add a user to the **isc4.gdb** security database.

Property	Purpose
UserName	User name to create; maximum 31 characters
Password	Password for the user; maximum 31 characters, only first 8 characters are significant
FirstName	Optional first name of person using this user name
MiddleName	Optional middle name of person using this user name
LastName	Optional last name of person using this user name
UserID	Optional user ID number, defined in **/etc/passwd**, to assign to the user
GroupID	Optional groupID number, defined in **/etc/group**, to assign to the user
SQLRole	)ptional role to use when attaching to the isc4.gdb security database; for more information on roles in InterBase, refer to "ANSI SQL 3 roles" in the *InterBase 6 Operations Guide*

TABLE 19.8 *TIBSecurityService* properties

The following code snippet allows you to set user information in *Edit* components, and then adds the user with the *AddUser* method.

```
try
  UserName := Edit1.Text;
  FirstName := Edit2.Text;
  MiddleName := Edit3.Text;
  LastName := Edit4.Text;
  UserID := StrToInt(Edit5.Text);
  GroupID := StrToInt(Edit6.Text);
  Password := Edit7.Text;
  AddUser;
finally
```

Listing users in the security database

Use the *DisplayUser* and *DisplayUsers* methods to display information for a single user or all users in the **isc4.gdb** security database, respectively.

Displaying information for a single user

To view the information for a single user, use the *DisplayUser* method. The following code snippet displays all the informatoin contained in the *TUserInfoArray*, keyed on the *UserName* field.

```
try
    UserName := Edit1.Text;
    DisplayUser(UserName);
    Edit2.Text := UserInfo[0].FirstName;
    Edit3.Text := UserInfo[0].MiddleName;
    Edit4.Text := UserInfo[0].LastName;
    Edit5.Text := IntToStr(UserInfo[0].UserID);
    Edit6.Text := IntToStr(UserInfo[0].GroupID);
finally
```

Displaying information for all users

To view all users, use the *DisplayUsers* method. *DisplayUsers* displays the user information contained in the *TUserInfo* array. The following code snippet displays all users in a memo window.

```
try
  DisplayUsers;
  for I := 0 to UserInfoCount - 1 do
    begin
      with UserInfo[i] do
        begin
           Memo1.Lines.Add('User Name : ' + UserName);
           Memo1.Lines.Add('Name: ' + FirstName + ' ' + MiddleName +
            ' ' + LastName);
           Memo1.Lines.Add('UID: ' + IntToStr(UserId));
           Memo1.Lines.Add('GID: ' + IntToStr(GroupId));
           Memo1.Lines.Add('---------------------------------');
        end;
      end;
finally
```

Removing a user from the security database

Use the *DeleteUser* method to remove a user from the **isc4.gdb** security database.

The following code snippet calls the *DeleteUser* method to delete the user indicated by the *UserName* property:

```
try
    UserName := Edit1.Text;
    DeleteUser;
finally
    Edit1.Clear;
    Active := False;
end;
```

If you remove a user entry from **isc4.gdb**, no one can log into any database on that server using that name. You must create a new entry for that name using the *AddUser* method.

Modifying a user in the security database

Use the *ModifyUser* method along with the properties listed in **TABLE 19.8** to modify user information in the **isc4.gdb** security database. You cannot change the *UserName* property, only the properties associated with that user name.

To modify user information you could display the user information using the example in **"Displaying information for a single user" on page 293**. The *TUserInfo* record is displayed in the Edit boxes. Use the *ModifyUser* code in the same way as the *AddUser* code.

Administering software activation certificates

You can use the *TIBLicensingService* component to install or remove software activation certificates.

Listing software activation certificates

You cannot use the *TIBLicensingService* component to view license information. You must use the License option in *TIBStatisticalService*. For more information, see **"Displaying license information" on page 297**.

Adding a software activation certificate

Use the *AddLicense* method along with the *Key* and *ID* properties to add a software activation certificate.

For example, the following code attaches to a server, and adds a license with the click of a button after the *Key* and *ID* are entered into *Edit* components.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 with IBLicensingService1 do
  begin
    ServerName := 'Poulet';
    LoginPrompt := False;
    Protocol := Local;
    Params.Add('user_name=SYSDBA');
    Params.Add('password=masterkey');
    Active := True;
    try
      Key := Edit1.Text;
      ID := Edit2.Text;
      AddLicense;
      ServiceStart;
    finally
    if Active then
      Active := False;
    end;
  end;
end;
```

Using the Action property

Instead of using the *AddLicense* or *RemoveLicense* methods, you could use the *Action* property:

```
Action := LicenseAdd;
```

or

```
Action := LicenseRemove;
```

Removing a software activation certificate

Use the *RemoveLicense* method along with the *Key* property to remove a software activation certificate:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
 with IBLicensingService1 do
  begin
    ServerName := 'Poulet';
    LoginPrompt := False;
    Params.Add('user_name=SYSDBA');
    Params.Add('password=masterkey');
    Active := True;
    try
      Key := Edit3.Text;
    finally
      RemoveLicense;
      Active := False;
    end;
  end;
end;
```

Displaying server properties

Use the *Options* property of *TIBServerProperties* to return server configuration information, including the version of the database and server, license and license mask information, and InterBase configuration parameters. These options are discussed in the following sections.

Displaying the database information

Use the *Database* option to display the *TDatabaseInfo* record, which consists of the number of databases attached to the server, the number of databases on the server, and the names and paths of the database files.

You can set the *Database* option to *True* in the Object Inspector, or set it in code.

The following code displays the elements of the *TDatabaseInfo* record. *NoOfAttachements* and *NoOfDatabases* are strings displayed in *Label* components, while *DbName* is an array of type string, and displayed in a *Memo* component.

```
Options := [Database];
      FetchDatabaseInfo;
      Label1.Caption := 'Number of Attachments = ' +
IntToStr(DatabaseInfo.NoOfAttachments);
      Label2.Caption := 'Number of Databases = ' +
IntToStr(DatabaseInfo.NoOfDatabases);
      for I:= 0 to High(DatabaseInfo.DbName) do
        Memo1.Lines.Add(DatabaseInfo.DbName[i])
```

Displaying license information

Use the *License* option to display all software activation certificate IDs and keys currently enabled on the server. The license information is stored as a record, which consists of the Key, ID, and description, all of which are arrays of type string. The number of licensed users on the system is displayed as an integer. You can set *License* to *True* in the Object Inspector, or you can set it in code.

The following code displays the number of licensed users in a *Label* component, and displays the *Key*, *ID*, and *Desc* arrays in a *Memo* component.

```
Options := [License];
      FetchLicenseInfo;
      Label1.Caption := 'Licensed Users = ' +
        IntToStr(LicenseInfo.LicensedUsers);
      for I:= 0 to High(LicenseInfo.Key) do
        Memo1.Lines.Add(LicenseInfo.Key[i] + ':' + LicenseInfo.ID[i]
        + ':' + LicenseInfo.Desc[i]);
```

Displaying license mask information

Use the *LicenseMask* option to display the license and capability masks on the server. A license mask is a bitmask representing the software activation certificate options currently enabled on the server. A capability mask is a bitmask representing the capabilities currently enabled on the server. You can set *LicenseMask* to *True* in the Object Inspector, or you can set it in code.

The following code displays the *LicenseMask* and *CapabilityMask* elements of the *TLicenseMaskInfo* record as *Label* components on a form.

```
Options := [LicenseMask];
      FetchLicenseMaskInfo;
      Label1.Caption := 'License Mask = ' +
IntToStr(LicenseMaskInfo.LicenseMask);
      Label2.Caption := 'Capability Mask = ' +
IntToStr(LicenseMaskInfo.CapabilityMask);
```

Displaying InterBase configuration parameters

Use the *ConfigParams* option along with the *FetchConfigParams* or *Fetch* method to display the parameters and values in the **ibconfig** file on the server (**isc_config** on UNIX and Linux). ConfigParams displays the location of the InterBase executable, the lock file, the message file, and the security database. It also displays the configuration file parameters. You can set *ConfigParams* to *True* in the Object Inspector, or you can set it in code.

The following code snippet shows how you could display configuration parameters as label captions.

```
Options := [ConfigParameters];
FetchConfigParams;
Label1.Caption := 'Base File = ' + ConfigParams.BaseLocation;
Label2.Caption := 'Lock File = ' + ConfigParams.LockFileLocation;
Label3.Caption := 'Message File = ' +
ConfigParams.MessageFileLocation;
Label4.Caption := 'Security Database = ' +
ConfigParams.SecurityDatabaseLocation;
```

You could also set the *ConfigFileData* array to display server key values in a *Memo* component.

```
var
  I: Integer;
  st1: string;
.
.
.
for I:= 0 to High(ConfigParams.ConfigFileData.ConfigFileValue) do
      begin
        case ConfigParams.ConfigFileData.ConfigFileKey[i] of
        ISCCFG_IPCMAP_KEY:  st1 := 'IPCMAP_KEY';
        ISCCFG_LOCKMEM_KEY:  st1 := 'LOCKMEM_KEY';
        .
        .
```

```
        .
        ISCCFG_DUMMY_INTRVL_KEY: st1 := 'DUMMY_INTRVL_KEY';
        end;
        Memo1.Lines.Add(st1 + ' = ' +
IntTostr(ConfigParams.ConfigFileData.ConfigFileValue[i]));
```

Displaying the server version

Use the *Version* option to display the server version information. The *TVersionInfo* record contains the server version, the implementation version, and the service version.

You can set the *Version* option to *True* in the Object Inspector, or set the *Options* property in code.

The following code displays server properties in *Label* components when a button is clicked:

```
Options := [Version];
     FetchVersionInfo;
    Label1.Caption := 'Server Version = ' + VersionInfo.ServerVersion;
     Label2.Caption := 'Server Implementation = ' +
        VersionInfo.ServerImplementation;
      Label3.Caption := 'Service Version = ' +
        IntToStr(VersionInfo.ServiceVersion);
end;
```

CHAPTER

20

Programming with Database Events

Use the *TIBEvents* component in your IBX-based application to register interest in and asynchronously handle InterBase server events. The InterBase event mechanism enables applications to respond to action and database changes made by other, concurrently running applications without the need for those applications to communicate directly with each other, and without incurring the expense of CPU time required for period polling to determine if an event has occurred.

Use the *TIBEvents* component in your application to register an event (or a list of events) with the *event manager*. The event manager maintains a list of events posted to it by triggers and stored procedures. It also maintains a list of applications that have registered an interest in events. Each time a new event is posted to it, the event manager notifies interested applications that the event has occurred.

To use *TIBEvents* in your application:

1. Create a trigger or stored procedure on the InterBase server which will post an event.
2. Add a *TIBDatabase* and a *TIBEvents* component to your form.
3. Add the events to the *Events* list and register them with the event manager.
4. Write an *OnEventAlert* event handler for each event.

Events are passed by triggers or stored procedures only when the transaction under which they occur is posted. In addition, InterBase consolidates events before posting them. For example, if an InterBase trigger posts 20 x STOCK_LOW events within a transaction, when the transaction is committed these will be consolidated into a single STOCK_LOW event, and the client will only receive one event notification.

For more information on events, refer to "Working with Events" in the *InterBase 6 Programmer's Guide*.

Setting up event alerts

Double click on the ellipsis button (...) of the *Events* property add an event to the *Events* list. Each *TIBEvents* component can handle up to 15 events. If you need to respond to more that 15 events use more that one *TIBEvents* component. If you attempt to add too many events at runtime, an exception will be raised.

To add an event to the *Events* list use the following code

```
TIBEvents.Events.Add( 'STOCK_LOW')
```

Writing an event handler

OnEventAlert is called every time an InterBase event is received by an *IBEvents* component. The *EventName* variable contains the name of the event that has just been received. The *EventCount* variable contains the number of *EventName* events that have been received since *OnEventAlert* was last called.

To cancel interest in any further events, set *CancelAlerts* to *True*. If you later decide that you want to receive events again, call the *QueueEvents* method. You cannot call *RegisterEvents*, *UnregisterEvents*, *QueueEvents* or *CancelEvents* from within an *OnEventAlert* event handler.

OnEventAlert runs as a separate thread to allow for true asynchronous event processing, however, the *IBEvents* component provides synchronization code to ensure that only one *OnEventAlert* event handler executes at any one time.

CHAPTER

21 Writing Installation Wizards

This chapter talks about the install/uninstall components.

Installing

TIBInstall (and its ancestor, *TIBSetup*) provide properties to allow you to build an InterBase install component into your application. *TIBInstall* allows you to set the installation source and destination, display your own installation messages, and set the individual InterBase components to be installed. These are discussed in the following sections.

The following sections describe how to set up an installation application, including selecting the installation options, setting the source and destination installation directories, and tracking the installation progress. Once the installation component is set up, execute it using the *InstallExecute* method.

Defining the installation component

Use the following properties with *TIBInstall* to define your installation component:

Property	Purpose
DestinationDirectory	Sets or returns the installation target path; if not set, defaults to what is in the Windows Registry
InstallOptions	Sets which InterBase components are to be installed; see below
MsgFilePath	Sets or returns the directory path where the **ibinstall.msg** file can be found
Progress	Returns an integer from 0 to 100 indicating the percentage of installation completed; if unset, no progress is displayed
RebootToComplete	If set to *True*, returns a message instructing the user to reboot after installation is complete
SourceDirectory	Sets or returns the path of the installation source files; in most cases, this will be a path on the InterBase CD
UnInstallFile	Returns the name and path of the uninstall file, which contains information on the installed options

TABLE 21.1 *TIBInstall* properties

Setting the installation options

The *InstallOptions* property allows you to set which InterBase components are to be installed. Set any of the following options to *True* to install it. For more information on each option, refer to the online help for *TInstallOptions*.

Option	Installs:
CmdLineTools	the InterBase command line tools, including isql,gbak, and gsec
ConnectivityClients	the InterBase connectivity clients, including ODBC, OLE DB, and JDBC
Examples	the InterBase database and API examples
MainComponents	the main InterBase components, including the client, server, documentation, GUI tools, and development tools.

TABLE 21.2 *TIBInstall* options

TIBInstall keeps track of the installed options in the uninstall file.

The following code snippet shows how you could set up a series of check boxes to allow a user to select the InterBase main components:

```
procedure TSampleform.ExecuteClick(Sender: TObject);
var
  MComps : TMainOptions;
begin
  Execute.Visible := False;
  Cancel.Visible  := True;
  MComps := [];

if ServerCheck.Checked then
    Include(MComps, moServer);

if ClientCheck.Checked then
    Include(MComps, moClient);

if ConServerCheck.Checked then
    Include(MComps, moConServer);

if GuiToolsCheck.Checked then
    Include(MComps, moGuiTools);

if DevCheck.Checked then
    Include(MComps, moDevelopment);

if DocCheck.Checked then
    Include(MComps, moDocumentation);

IBInstall1.InstallOptions.MainComponents := MComps;
```

Setting up the source and destination directories

Use the *SourceDirectory*, *DestinationDirectory* and *SuggestedDestination* properties along with the *InstallCheck* method to set up the source and destination directories for your installation component. The following code snippet uses two TDirectoryListBox components, *SrcDir* and *DestDir*, to allow the user to change the source and destination directories. The *InstallCheck* method checks to see if everything is prepared for the installaion.

```
try
    IBInstall1.SourceDirectory := SrcDir.Directory;
    IBInstall1.DestinationDirectory := DestDir.Directory;
    IBInstall1.InstallCheck;
  except
```

```
    on E:EIBInstallError do
    begin
      Label1.Caption := '';
      Cancel.Visible := False;
      Execute.Visible := True;
      ProgressBar1.Visible := False;
      Exit;
    end;
  end;
```

- *Setting up the installation progress components*

Use the *Progress* property, along with a *ProgressBar* component track the installation status.

```
function TSampleform.IBInstall1StatusChange(
  Sender: TObject; StatusComment : String): TStatusResult;
begin
    Result := srContinue;
    ProgressBar1.Position := IBInstall1.Progress;
    Label1.Caption := StatusComment;

if Cancelling then
    begin
      if Application.MessageBox(PChar('UserAbort'),
         PChar('Do you want to exit'), MB_YESNO ) = IDYES then
         Result := srAbort;
    end
    else
       // Update billboards and other stuff as necessary
       Application.ProcessMessages;
end;
```

Defining the uninstall component

Use the *TIBUnInstall* component to define which components are removed and what messages are displayed when the user uninstalls InterBase. The following code snippet shows a simple uninstall component.

```
procedure TUninstall.bUninstallClick(Sender: TObject);
begin
```

```
  IBUninstall1.UnInstallFile :=
'C:\Program Files\InterBase Corp\InterBase\ibuninst.000';
  bUninstall.Visible := False;
  ProgressBar1.Visible := True;

try
     IBUninstall1.UnInstallCheck;
  except
    on E:EIBInstallError do
    begin
      Application.MessageBox(PChar(E.Message), PChar('Precheck
Error'), MB_OK);
      Label1.Caption := '';
      bUninstall.Visible := True;
      ProgressBar1.Visible := False;
      Exit;
    end;
  end;

try
   IBUninstall1.UnInstallExecute;
  except
   on  E:EIBInstallError do
   begin
    Application.MessageBox(PChar(E.Message), PChar( 'Install Error'),
MB_OK );
      Label1.Caption := '';
      bUninstall.Visible := True;
      ProgressBar1.Visible := False;
      Exit;
   end;
  end;
  Label1.Caption := 'Uninstall Completed';
  ProgressBar1.Visible := False;
  bCancel.Visible := False;
  bExit.Visible := True;
end;
```

CHAPTER

22

Migrating to InterBase Express

This chapter covers migration from the BDE to IBX. [To come]

APPENDIX

A

Differences Between Delphi Components and InterBase Components

This chapter documents the differences between traditional Delphi components.

[To come]

Index

Q

R

S

V

W