CHAPTER

28

Writing Desktop Database Applications

IN THIS CHAPTER

In this chapter, you'll learn the art and science of accessing external database files from your Delphi applications. If you're new to database programming, we do assume a bit of database knowledge, but this chapter will get you started on the road to creating high-quality database applications. If database applications are "old hat" to you, you'll benefit from the chapter's demonstration of Delphi's spin on database programming. In this chapter, you first learn about datasets and techniques for manipulating them, and later you learn how to work with tables and queries specifically. Along the way, this chapter outlines the important points you need to know to be a productive Delphi database developer.

Delphi 5 ships with version 5.0 of the Borland Database Engine (BDE), which offers you the capability to communicate with Paradox, dBASE, Access, FoxPro, ODBC, ASCII text, and SQL database servers all in much the same manner. Unlike previous versions, the Standard edition of Delphi 5 does not contain database connectivity. The Professional edition provides connections to file-based Paradox, dBASE, Access, FoxPro, and ASCII text formats, in addition to connectivity to Local InterBase and ODBC data sources. Delphi Enterprise builds upon Delphi Professional, adding high-performance BDE SQL Links connections for InterBase, Microsoft SQL Server, Oracle, Informix Dynamic Server, Sybase Adaptive Server, and DB2. Additionally, Delphi Enterprise also provides ADOExpress components for native access to Microsoft ActiveX Data Objects (ADO) data sources. The topics discussed pertain primarily to using Delphi with file-based data, such as Paradox and dBASE tables, although the chapter will also touch on data access via ODBC and ADO. This chapter also serves as a primer for the next chapter, "Developing Client/Server Applications."

Working with Datasets

A *dataset* is a collection of rows and columns of data. Each *column* is of some homogeneous data type, and each *row* is made up of a collection of data of each column data type. Additionally, a column is also known as a *field*, and a row is sometimes called a *record*. VCL encapsulates a dataset into an abstract component called `TDataSet`. `TDataSet` introduces many of the properties and methods necessary for manipulating and navigating a dataset.

To help keep the nomenclature clear and to cover some of the basics, the following list explains some of the common database terms that are used in this and other database-oriented chapters:

- A *dataset* is a collection of discrete data records. Each record is made up of multiple fields. Each field can contain a different type of data (integer number, string, decimal number, graphic, and so on). Datasets are represented by VCL's abstract `TDataset` class.
- A *table* is a special type of dataset. A table is generally a file containing records that are physically stored on a disk somewhere. VCL's `TTable` class encapsulates this functionality.

- A *query* is also a special type of dataset. Think of queries as "memory tables" that are generated by special commands that manipulate some physical table or set of tables. VCL has a `TQuery` class to handle queries.
- A *database* refers to a directory on a disk (when dealing with nonserver data such as Paradox and dBASE files) or a SQL database (when dealing with SQL servers). A database can contain multiple tables. As you may have guessed, VCL also has a `TDatabase` class.
- An *index* defines rules by which a table is ordered. To have an index on a particular field in a table means to sort its records based on the value that field holds for each record. The `TTable` component contains properties and methods that help you manipulate indexes.

NOTE

We mentioned earlier that this chapter assumes a bit of database knowledge. This chapter is not intended to be a primer on database programming, and we expect that you're already familiar with the items in this list. If terms such as *database*, *table*, and *index* sound foreign to you, you might want to obtain an introductory text on database concepts.

VCL Database Architecture

During the development of Delphi 3, VCL's database architecture was significantly overhauled in order to open the dataset architecture to allow non-BDE datasets to more easily be used within Delphi. At the root of this architecture is the base `TDataSet` class. `TDataSet` is a component that provides an abstract representation of dataset records and fields. A number of methods of TDataSet can be overridden in order to create a component that communicates with some particular physical data format. Following this formula, VCL's `TBDEDataSet` descends from `TDataSet` and serves as the base class for data sources that communicate via the BDE. If you'd like to learn how to create a `TDataSet` descendant to plug some type of custom data into this architecture, you'll find an example in Chapter 30, "Extending Database VCL."

BDE Data-Access Components

The Data Access page of the Component Palette contains the VCL components you'll use to access and manage BDE datasets. These are shown in Figure 28.1. VCL represents datasets with three components: `TTable`, `TQuery`, and `TStoredProc`. These components all descend directly from the `TDBDataSet` component, which descends from `TBDEDataSet` (which, in turn, descends from `TDataSet`). As mentioned earlier, `TDataSet` is an abstract component that encapsulates dataset management, navigation, and manipulation. `TBDEDataSet` is an abstract component that represents a BDE-specific dataset. `TDBDataSet` introduces concepts such as BDE databases and sessions (these are explained in detail in the next chapter). Throughout the rest of this chapter, we'll refer to this type of BDE-specific dataset simply as a *dataset.*

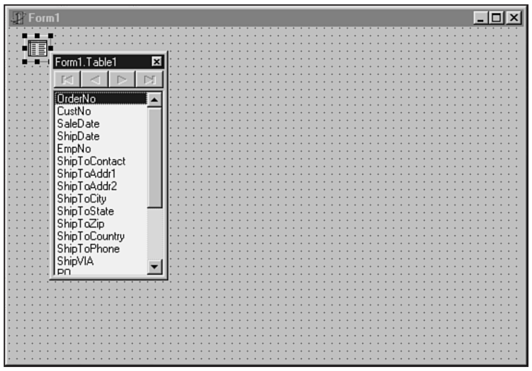


FIGURE 28.1
The Data Access page of the Component Palette.

As their names imply, `TTable` is a component that represents the structure and data contained within a database table, `TQuery` is a component representing the set of data returned from a SQL query operation, and `TStoredProc` encapsulates a stored procedure on a SQL server. In this chapter, for simplicity's sake, we use the `TTable` component when discussing datasets. Later, the `TQuery` component is covered in detail.

Opening a Dataset

Before you can do any nifty manipulation of your dataset, you must first open it. To open a dataset, simply call its `Open()` method, as shown in this example:

```
Table1.Open;
```

This is equivalent, by the way, to setting a dataset's `Active` property to `True`:

```
Table1.Active := True;
```

There's slightly less overhead in the latter method, because the `Open()` method ends up setting the `Active` property to `True`. However, the overhead is so minimal that it's not worth worrying about.

Once the dataset has been opened, you're free to manipulate it, as you'll see in just a moment. When you finish using the dataset, you should close it by calling its `Close()` method, like this:

```
Table1.Close;
```

Alternatively, you could close it by setting its `Active` property to `False`, like this:

```
Table1.Active := False;
```

TIP

When you're communicating with SQL servers, a connection to the database must be established when you first open a dataset in that database. When you close the last dataset in a database, your connection is terminated. Opening and closing these connections involves a certain amount of overhead. Therefore, if you find that you open and close the connection to the database often, use a `TDatabase` component instead to maintain a connection to a SQL server's database throughout many open and close operations. The `TDatabase` component is explained in more detail in the next chapter.

Navigating Datasets

`TDataSet` provides some simple methods for basic record navigation. The `First()` and `Last()` methods move you to the first and last records in the dataset, respectively, and the `Next()` and `Prior()` methods move you either one record forward or back in the dataset. Additionally, the `MoveBy()` method, which accepts an `Integer` parameter, moves you a specified number of records forward or back.

NOTE

One of the big, but less obvious, benefits of using the BDE is that it allows navigable SQL tables and queries. SQL data generally is not navigable—you can move forward through the rows of a query but not backward. Unlike ODBC, BDE makes SQL data navigable.

BOF, EOF, and Looping

`BOF` and `EOF` are Boolean properties of `TDataSet` that reveal whether the current record is the first or last record in the dataset. For example, you might need to iterate through each record in a dataset until reaching the last record. The easiest way to do so would be to employ a `while` loop to keep iterating over records until the `EOF` property returns `True`, as shown here:

```
Table1.First;                       // go to beginning of data set
while not Table1.EOF do             // iterate over table
begin
```

```
  // do some stuff with current record
  Table1.Next;                          // move to next record
end;
```

CAUTION

Be sure to call the `Next()` method inside your `while-not-EOF` loop; otherwise, your application will get caught in an endless loop.

Avoid using a `repeat..until` loop to perform actions on a dataset. The following code may look OK on the surface, but bad things may happen if you try to use it on an empty dataset, because the `DoSomeStuff()` procedure will always execute at least once, regardless of whether the dataset contains records:

```
repeat
  DoSomeStuff;
  Table1.Next;
until Table1.EOF;
```

Because the `while-not-EOF` loop performs the check up front, you won't encounter such a problem with this construct.

Bookmarks

Bookmarks enable you to save your place in a dataset so that you can come back to the same spot at a later time. Bookmarks are very easy to use in Delphi because you only have one property to remember.

Delphi represents a bookmark as type `TBookmarkStr`. `TTable` has a property of this type called `Bookmark`. When you read from this property, you obtain a bookmark, and when you write to this property, you go to a bookmark. When you find a particularly interesting place in a dataset that you'd like to be able to get back to easily, here's the syntax to use:

```
var
  BM: TBookmarkStr;
begin
  BM := Table1.Bookmark;
```

When you want to return to the place in the dataset you marked, just do the reverse—set the `Bookmark` property to the value you obtained earlier by reading the `Bookmark` property:

```
Table1.Bookmark := BM;
```

`TBookmarkStr` is defined as an `AnsiString`, so memory is automatically managed for bookmarks (you never have to free them). If you'd like to clear an existing bookmark, just set it to an empty string:

```
BM := '';
```

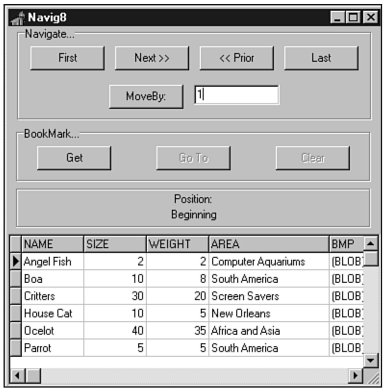
Note that `TBookmarkStr` is an `AnsiString` for storage convenience. You should consider it an opaque data type and not depend on the implementation, because the bookmark data is completely determined by BDE and the underlying data layers.

NOTE

Although 32-bit Delphi still supports `GetBookmark()`, `GotoBookmark()`, and `FreeBookmark()` from Delphi 1.0, because the 32-bit Delphi technique is a bit cleaner and less prone to error, you should use this newer technique unless you have to maintain compatibility with 16-bit projects.

Navigational Example

You'll now create a small project that incorporates the `TDataSet` navigational methods and properties you just learned. This project will be called Navig8, and the main form for this project is shown in Figure 28.2.



NAME	SIZE	WEIGHT	AREA	BMP
Angel Fish	2	2	Computer Aquariums	(BLOB)
Boa	10	8	South America	(BLOB)
Critters	30	20	Screen Savers	(BLOB)
House Cat	10	5	New Orleans	(BLOB)
Ocelot	40	35	Africa and Asia	(BLOB)
Parrot	5	5	South America	(BLOB)

FIGURE 28.2
The Navig8 project's main form.

To display the data contained in a `TTable` object, this project will employ the `TDBGrid` component. The process of "wiring" a data-aware control such as the `TDBGrid` component to a dataset requires several steps. The following list covers the steps for displaying `Table1`'s data in `DBGrid1`:

1. Set `Table1`'s `DatabaseName` property to an existing alias or directory. Use the `DBDEMOS` alias if you installed Delphi's sample programs.
2. Choose a table from the list presented in `Table1`'s `TableName` property.
3. Drop a `TDataSource` component on the form and wire it to `TTable` by setting `DataSource1`'s dataset property to `Table1`. `TDataSource` serves as a conduit between data sources and controls; it's explained in more detail earlier in the chapter.
4. Wire the `TDBGrid` component to the `TDataSource` component by setting `DBGrid1`'s `DataSource` property to `DataSource1`.
5. Open the table by setting `Table1`'s `Active` property to `True`.
6. Poof! You now have data in the grid control.

TIP

A shortcut for picking components from the drop-down list provided for the `DataSet` and `DataSource` properties is to double-click the area to the right of the property name in the Object Inspector. This sets the property value to the first item in the drop-down list.

The source code for main unit of Navig8, called `Nav.pas`, is shown in Listing 28.1.

LISTING 28.1 The Source Code for `Nav.pas`

```
unit Nav;

interface

uses
  SysUtils, Windows, Messages, Classes, Controls, Forms, StdCtrls,
  Grids, DBGrids, DB, DBTables, ExtCtrls;

type
  TForm1 = class(TForm)
    Table1: TTable;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    GroupBox1: TGroupBox;
    GetButton: TButton;
    GotoButton: TButton;
    ClearButton: TButton;
    GroupBox2: TGroupBox;
    FirstButton: TButton;
    LastButton: TButton;
    NextButton: TButton;
    PriorButton: TButton;
    MoveByButton: TButton;
    Edit1: TEdit;
    Panel1: TPanel;
    PosLbl: TLabel;
    Label1: TLabel;
    procedure FirstButtonClick(Sender: TObject);
    procedure LastButtonClick(Sender: TObject);
    procedure NextButtonClick(Sender: TObject);
    procedure PriorButtonClick(Sender: TObject);
    procedure MoveByButtonClick(Sender: TObject);
    procedure DataSource1DataChange(Sender: TObject; Field: TField);
    procedure GetButtonClick(Sender: TObject);
    procedure GotoButtonClick(Sender: TObject);
    procedure ClearButtonClick(Sender: TObject);
  private
    BM: TBookmarkStr;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FirstButtonClick(Sender: TObject);
begin
  Table1.First;          // Go to first record in table
end;

procedure TForm1.LastButtonClick(Sender: TObject);
begin
  Table1.Last;           // Go to last record in table
end;

procedure TForm1.NextButtonClick(Sender: TObject);
begin
  Table1.Next;           // Go to next record in table
```

continues

LISTING 28.1 Continued

```
end;

procedure TForm1.PriorButtonClick(Sender: TObject);
begin
  Table1.Prior;         // Go to prior record in table
end;

procedure TForm1.MoveByButtonClick(Sender: TObject);
begin
  // Move a specified number of record forward or back in the table
  Table1.MoveBy(StrToInt(Edit1.Text));
end;

procedure TForm1.DataSource1DataChange(Sender: TObject; Field: TField);
begin
  // Set caption appropriately, depending on state of Table1 BOF/EOF
  if Table1.BOF then PosLbl.Caption := 'Beginning'
  else if Table1.EOF then PosLbl.Caption := 'End'
  else PosLbl.Caption := 'Somewheres in between';
end;

procedure TForm1.GetButtonClick(Sender: TObject);
begin
  BM := Table1.Bookmark;        // Get a bookmark
  GotoButton.Enabled := True;   // Enable/disable proper buttons
  GetButton.Enabled := False;
  ClearButton.Enabled := True;
end;

procedure TForm1.GotoButtonClick(Sender: TObject);
begin
  Table1.Bookmark := BM;        // Go to the bookmark position
end;

procedure TForm1.ClearButtonClick(Sender: TObject);
begin
  BM := '';                     // clear the bookmark
  GotoButton.Enabled := False;  // Enable/disable appropriate buttons
  GetButton.Enabled := True;
  ClearButton.Enabled := False;
end;

end.
```

This example illustrates quite well the fact that you can use Delphi's database classes to do quite a lot of database manipulation in your programs with very little code.

Note that you should initially set the `Enabled` properties of `GotoButton` and `FreeButton` to `False`, because you can't use them until a bookmark is allocated. The `FreeButtonClick()` and `GetButtonClick()` methods ensure that the proper buttons are enabled, depending on whether a bookmark has been set.

Most of the other procedures in this example are one-liners, although one method that does require some explanation is `TForm1.DataSource1DataChange()`. This method is wired to `DataSource1`'s `OnDataChange` event, which fires every time a field value changes (for example, when you move from one record to another). This event checks to see whether you're at the beginning, in the middle, or at the end of a dataset; it then changes the label's caption appropriately. You'll learn more about the `TTable` and `TDataSource` events a bit later in this chapter.

BOF and EOF

You may notice that when you run the Navig8 project, `PosLbl`'s caption indicates that you're at the beginning of the dataset, which makes sense. However, if you move to the next record and back again, `PosLbl`'s caption isn't aware that you're at the first record. Notice, however, that `PosLbl.Caption` does indicate `BOF` if you click the Prior button once more. Note that the same holds true for `EOF` if you try this at the end of the dataset. Why?

The reason is that the BDE cannot be sure you're at the beginning or end of the dataset anymore, because another user of the table (if it's a networked table) or even another process within your program could have added a record to the beginning or end of the table in the time it took you to move from the first to the second record and then back again.

With that in mind, `BOF` can only be `True` under one of the following circumstances:

- You just opened the dataset.
- You just called the dataset's `First()` method.
- A call to `TDataSet.Prior()` failed, indicating that there are no prior records.

Likewise, `EOF` can only be `True` under the following circumstances:

- You opened an empty dataset.
- You just called the dataset's `Last()` method.
- A call to `TDataSet.Next()` failed, indicating that there are no more records.

A subtle but important piece of information that you can garner from this list is that you know a dataset is empty when both `BOF` and `EOF` are `True`.

TDataSource

A TDataSource component was used in that last example, so let's digress for a moment to discuss this very important object. TDataSource is the conduit that enables data-access components such as TTable components to connect to data controls such as TDBEdit and TDBLookupCombo components. In addition to being the interface between datasets and data-aware controls, TDataSource contains a couple of handy properties and events that make your life easier when manipulating data.

The State property of TDataSource reveals the current state of the underlying dataset. The value of State tells you whether the dataset is currently inactive or in Insert, Edit, SetKey, or CalcFields mode, for example. The State property of TDataSet is explained in more detail later in this chapter. The OnStateChange event fires whenever the value of this property changes.

The OnDataChange event of TDataset is executed whenever the dataset becomes active or a data-aware control informs the dataset that something has changed.

The OnUpdateData event occurs whenever a record is posted or updated. This is the event that causes data-aware controls to change their value based on the contents of the table. You can respond to the event yourself to keep track of such changes within your application.

Working with Fields

Delphi enables you to access the fields of any dataset through the TField object and its descendants. Not only can you get and set the value of a given field of the current record of a dataset, but you can also change the behavior of a field by modifying its properties. You can also modify the dataset, itself, by changing the visual order of fields, removing fields, or even creating new calculated or lookup fields.

Field Values

It's very easy to access field values from Delphi. TDataSet provides a default array property called FieldValues[] that returns the value of a particular field as a Variant. Because FieldValues[] is the default array property, you don't need to specify the property name to access the array. For example, the following piece of code assigns the value of Table1's CustName field to String S:

```
S := Table1['CustName'];
```

You could just as easily store the value of an integer field called CustNo in an integer variable called I:

```
I := Table1['CustNo'];
```

A powerful corollary to this is the capability to store the values of several fields into a `Variant` array. The only catches are that the `Variant` array index must be zero based and the `Variant` array contents should be `varVariant`. The following code demonstrates this capability:

```
const
  AStr = 'The %s is of the %s category and its length is %f in.';
var
  VarArr: Variant;
  F: Double;
begin
  VarArr := VarArrayCreate([0, 2], varVariant);
  { Assume Table1 is attached to Biolife table }
  VarArr := Table1['Common_Name;Category;Length_In'];
  F := VarArr[2];
  ShowMessage(Format(AStr, [VarArr[0], VarArr[1], F]));
end;
```

Delphi 1 programmers will note that the `FieldValues[]` technique is much easier than the previous technique for accessing field values. That technique (which still works in 32-bit Delphi for backward compatibility) involves using `TDataset`'s `Fields[]` array property or `FieldsByName()` function to access individual `TField` objects associated with the dataset. The `TField` component provides information about a specific field.

`Fields[]` is a zero-based array of `TField` objects, so `Fields[0]` returns a `TField` representing the first logical field in the record. `FieldsByName()` accepts a string parameter that corresponds to a given field name in the table; therefore, `FieldsByName('OrderNo')` would return a `TField` component representing the `OrderNo` field in the current record of the dataset.

Given a `TField` object, you can retrieve or assign the field's value using one of the `TField` properties shown in Table 28.1.

TABLE 28.1 Properties to Access `TField` Values

Property	*Return Type*
`AsBoolean`	`Boolean`
`AsFloat`	`Double`
`AsInteger`	`Longint`
`AsString`	`String`
`AsDateTime`	`TDateTime`
`Value`	`Variant`

If the first field in the current dataset is a string, you can store its value in the `String` variable `S`, like this:

```
S := Table1.Fields[0].AsString;
```

The following code sets the integral variable `I` to contain the value of the `'OrderNo'` field in the current record of the table:

```
I := Table1.FieldsByName('OrderNo').AsInteger;
```

Field Data Types

If you want to know the type of a field, look at `TField`'s `DataType` property, which indicates the data type with respect to the database table (irrespective of a corresponding Object Pascal type). The `DataType` property is of `TFieldType`, and `TFieldType` is defined as follows:

```
type
  TFieldType = (ftUnknown, ftString, ftSmallint, ftInteger, ftWord,
    ftBoolean, ftFloat, ftCurrency, ftBCD, ftDate, ftTime, ftDateTime,
    ftBytes, ftVarBytes, ftAutoInc, ftBlob, ftMemo, ftGraphic, ftFmtMemo,
    ftParadoxOle, ftDBaseOle, ftTypedBinary, ftCursor, ftFixedChar,
    ftWideString, ftLargeint, ftADT, ftArray, ftReference, ftDataSet,
    ftOraBlob, ftOraClob, ftVariant, ftInterface, ftIDispatch, ftGuid);
```

There are descendants of `TField` designed to work specifically with many of the preceding data types. These are covered a bit later in this chapter.

Field Names and Numbers

To find the name of a specified field, use `TField`'s `FieldName` property. For example, the following code places the name of the first field in the current table in the `String` variable `S`:

```
var
  S: String;
begin
  S := Table1.Fields[0].FieldName;
end;
```

Likewise, you can obtain the number of a field you know only by name by using the `FieldNo` property. The following code stores the number of the `OrderNo` field in the `Integer` variable `I`:

```
var
  I: integer;
begin
  I := Table1.FieldsByName('OrderNo').FieldNo;
end;
```

NOTE

To determine how many fields a dataset contains, use `TDataset`'s `FieldList` property. `FieldList` represents a flattened view of all the nested fields in a table containing fields that are abstract data types (ADTs).

For backward compatibility, the `FieldCount` property still works, but it will skip over any ADT fields.

Manipulating Field Data

Here's a three-step process for editing one or more fields in the current record:

1. Call the dataset's `Edit()` method to put the dataset into Edit mode.
2. Assign new values to the fields of your choice.
3. Post the changes to the dataset either by calling the `Post()` method or by moving to a new record, which will automatically post the edit.

For instance, a typical record edit looks like this:

```
Table1.Edit;
Table1['Age'] := 23;
Table1.Post;
```

TIP

Sometimes you work with datasets that contain read-only data. Examples of this would include a table located on a CD-ROM drive or a query with a non-live result set. Before attempting to edit data, you can determine whether the dataset contains read-only data before you try to modify it by checking the value of the `CanModify` property. If `CanModify` is `True`, you have the green light to edit the dataset.

Along the same lines as editing data, you can insert or append records to a dataset in much the same way:

1. Call the dataset's `Insert()` or `Append()` method to put the dataset into Insert or Append mode.
2. Assign values to the dataset's fields.
3. Post the new record to the dataset either by calling `Post()` or by moving to a new record, which forces a post to occur.

NOTE

When you're in Edit, Insert, or Append mode, keep in mind that your changes will always post when you move off the current record. Therefore, be careful when you use the `Next()`, `Prior()`, `First()`, `Last()`, and `MoveBy()` methods while editing records.

If at some point, before your additions or modifications to the dataset are posted, you want to abandon your changes, you can do so by calling the `Cancel()` method. For instance, the following code cancels the edit before changes are posted to the table:

```
Table1.Edit;
Table1['Age'] := 23;
Table1.Cancel;
```

`Cancel()` undoes changes to the dataset, takes the dataset out of Edit, Append, or Insert mode, and puts it back into Browse mode.

To round out the set of `TDataSet`'s record-manipulation methods, the `Delete()` method removes the current record from the dataset. For example, the following code deletes the last record in the table:

```
Table1.Last;
Table1.Delete;
```

The Fields Editor

Delphi gives you a great degree of control and flexibility when working with dataset fields through the Fields Editor. You can view the Fields Editor for a particular dataset in the Form Designer, either by double-clicking the `TTable`, `TQuery`, or `TStoredProc` or by selecting Fields Editor from the dataset's local menu. The Fields Editor window enables you to determine which of a dataset's fields you want to work with and create new calculated or lookup fields. You can use a local menu to accomplish these tasks. The Fields Editor window with its local menu deployed is shown in Figure 28.3.

To demonstrate the usage of the Fields Editor, open a new project and drop a `TTable` component onto the main form. Set `Table1`'s `DatabaseName` property to `DBDEMOS` (this is the alias that points to the Delphi sample tables) and set the `TableName` property to `ORDERS.DB`. To provide some visual feedback, also drop a `TDataSource` and `TDBGrid` component on the form. Hook `DataSource1` to `Table1` and then hook `DBGrid1` to `DataSource1`. Now set `Table1`'s `Active` property to `True`, and you'll see `Table1`'s data in the grid.

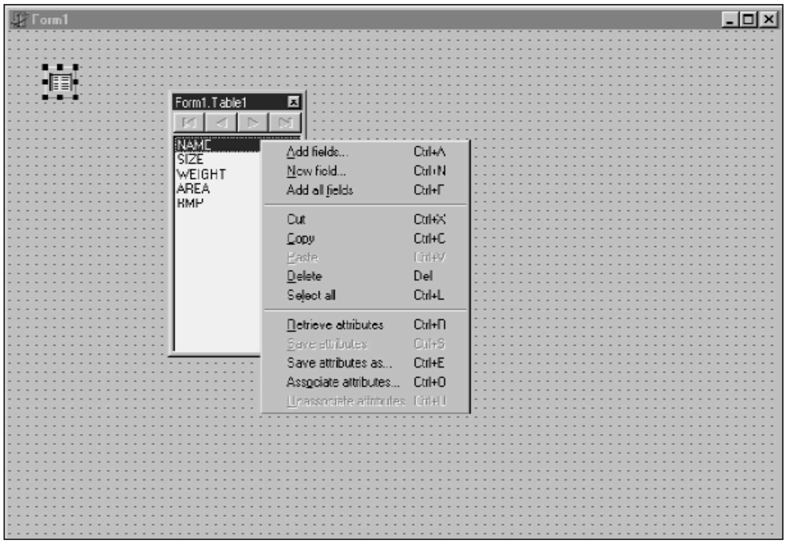


FIGURE 28.3
The Fields Editor's local menu.

Adding Fields

Invoke the Fields Editor by double-clicking `Table1`, and you'll see the Fields Editor window, as shown in Figure 28.3. Let's say you want to limit your view of the table to only a few fields. Select Add Fields from the Fields Editor local menu. This will invoke the Add Fields dialog. Highlight the `OrderNo`, `CustNo`, and `ItemsTotal` fields in this dialog and click OK. The three selected fields will now be visible in the Fields Editor and in the grid.

Delphi creates `TField` descendant objects, which map to the dataset fields you select in the Fields Editor. For example, for the three fields mentioned in the preceding paragraph, Delphi adds the following declarations of `TField` descendants to the source code for your form:

```
Table1OrderNo: TFloatField;
Table1CustNo: TFloatField;
Table1ItemsTotal: TCurrencyField;
```

Notice that the name of the field object is the concatenation of the `TTable` name and the field name. Because these fields are created in code, you can also access `TField` descendant properties and methods in your code rather than solely at design time.

TField Descendants

Let's digress for just a moment on the topic of `TFields`. There are one or more different `TField` descendant objects for each field type (field types are described in the "Field Data

Types" section, earlier in this chapter). Many of these field types also map to Object Pascal data types. Table 28.2 shows the various classes in the `TField` hierarchy, their ancestor classes, their field types, and the Object Pascal types to which they equate.

Table 28.2 `TField` Descendants and their Field Types

Field Class	*Ancestor*	*Field Type*	*Object Pascal Type*
`TStringField`	`TField`	`ftString`	`String`
`TWideStringField`	`TStringField`	`ftWideString`	`WideString`
`TGuidField`	`TStringField`	`ftGuid`	`TGUID`
`TNumericField`	`TField`	*	*
`TIntegerField`	`TNumericField`	`ftInteger`	`Integer`
`TSmallIntField`	`TIntegerField`	`ftSmallInt`	`SmallInt`
`TLargeintField`	`TNumericField`	`ftLargeint`	`Int64`
`TWordField`	`TIntegerField`	`ftWord`	`Word`
`TAutoIncField`	`TIntegerField`	`ftAutoInc`	`Integer`
`TFloatField`	`TNumericField`	`ftFloat`	`Double`
`TCurrencyField`	`TFloatField`	`ftCurrency`	`Currency`
`TBCDField`	`TNumericField`	`ftBCD`	`Double`
`TBooleanField`	`TField`	`ftBoolean`	`Boolean`
`TDateTimeField`	`TField`	`ftDateTime`	`TDateTime`
`TDateField`	`TDateTimeField`	`ftDate`	`TDateTime`
`TTimeField`	`TDateTimeField`	`ftTime`	`TDateTime`
`TBinaryField`	`TField`	*	*
`TBytesField`	`TBinaryField`	`ftBytes`	None
`TVarBytesField`	`TBytesField`	`ftVarBytes`	None
`TBlobField`	`TField`	`ftBlob`	None
`TMemoField`	`TBlobField`	`ftMemo`	None
`TGraphicField`	`TBlobField`	ftGraphic	None
`TObjectField`	`TField`	*	*
`TADTField`	`TObjectField`	`ftADT`	None
`TArrayField`	`TObjectField`	`ftArray`	None
`TDataSetField`	`TObjectField`	`ftDataSet`	`TDataSet`
`TReferenceField`	`TDataSetField`	`ftReference`	

Field Class	Ancestor	Field Type	Object Pascal Type
`TVariantField`	`TField`	`ftVariant`	`OleVariant`
`TInterfaceField`	`TField`	`ftInterface`	`IUnknown`
`TIDispatchField`	`TInterfaceField`	`ftIDispatch`	`IDispatch`
`TAggregateField`	`TField`	None	None

**Denotes an abstract base class in the* `TField` *hierarchy*

As Table 28.2 shows, BLOB and Object field types are special in that they don't map directly to native Object Pascal types. BLOB fields are discussed in more detail later in this chapter.

Fields and the Object Inspector

When you select a field in the Fields Editor, you can access the properties and events associated with that `TField` descendant object in the Object Inspector. This feature enables you to modify field properties such as minimum and maximum values, display formats, and whether the field is required as well as whether it's read-only. Some of these properties, such as `ReadOnly`, are obvious in their purpose, but some aren't quite as intuitive. Some of the less intuitive properties are covered later in this chapter. Figure 28.4 shows the `OrderNo` field focused in the Object Inspector.

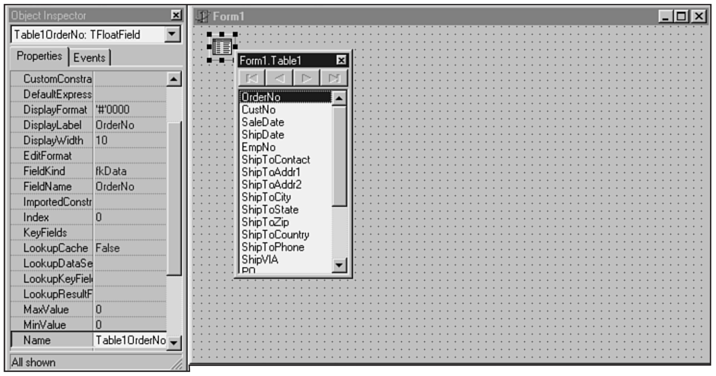


Table1OrderNo: TFloatField

CustomConstra	
DefaultExpress	
DisplayFormat	'#'0000
DisplayLabel	OrderNo
DisplayWidth	10
EditFormat	
FieldKind	fkData
FieldName	OrderNo
ImportedConstr	
Index	0
KeyFields	
LookupCache	False
LookupDataSe	
LookupKeyFiel	
LookupResultF	
MaxValue	0
MinValue	0
Name	Table1OrderNo

OrderNo
CustNo
SaleDate
ShipDate
EmpNo
ShipToContact
ShipToAddr1
ShipToAddr2
ShipToCity
ShipToState
ShipToZip
ShipToCountry
ShipToPhone
ShipVIA
PO

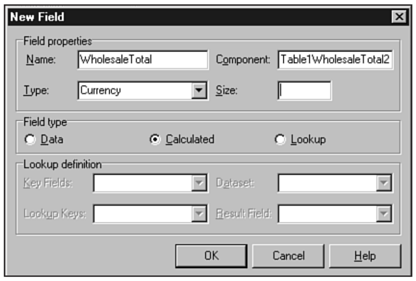
Figure 28.4
Editing a field's properties.

Switch to the Events page of the Object Inspector and you'll see that there are also events associated with field objects. The events `OnChange`, `OnGetText`, `OnSetText`, and `OnValidate` are all well-documented in the online help. Simply click to the left of the event in the Object

Inspector and press F1. Of these, `OnChange` is probably the most common to use. It enables you to perform some action whenever the contents of the field change (moving to another record or adding a record, for example).

Calculated Fields

You can also add calculated fields to a dataset using the Fields Editor. Let's say, for example, you wanted to add a field that figures the wholesale total for each entry in the `ORDERS` table, and the wholesale total was 32 percent of the normal total. Select New Field from the Fields Editor local menu, and you'll be presented with the New Field dialog, as shown in Figure 28.5. Enter the name, `WholesaleTotal`, for the new field in the Name edit control. The type of this field is Currency, so enter that in the Type edit control. Make sure the Calculated radio button is selected in the Field Type group; then press OK. Now the new field will show up in the grid, but it won't yet contain any data.



New Field

Field properties
Name: WholesaleTotal
Component: Table1WholesaleTotal2
Type: Currency
Size:

Field type
Data [✓ Calculated] Lookup

Lookup definition
Key Fields:
Dataset:
Lookup Keys:
Result Field:

OK Cancel Help

FIGURE 28.5
Adding a calculated field with the New Field dialog.

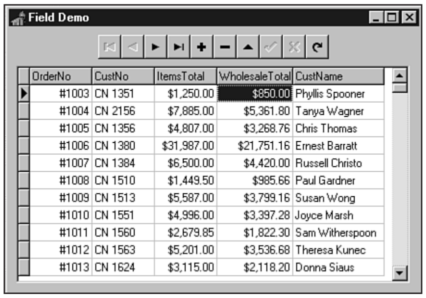
To cause the new field to become populated with data, you must assign a method to `Table1`'s `OnCalcFields` event. The code for this event simply assigns the value of the `WholesaleTotal` field to be 32 percent of the value of the existing `SalesTotal` field. This method, which handles `Table1.OnCalcFields`, is shown here:

```
procedure TForm1.Table1CalcFields(DataSet: TDataSet);
begin
  DataSet['WholesaleTotal'] := DataSet['ItemsTotal'] * 0.68;
end;
```

Figure 28.6 shows that the `WholesaleTotal` field in the grid now contains the correct data.

Lookup Fields

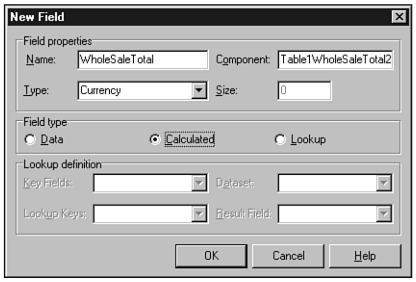
Lookup fields enable you to create fields in a dataset that actually look up their values from another dataset. To illustrate this, you'll add a lookup field to the current project. The `CustNo` field of the `ORDERS` table doesn't mean anything to someone who doesn't have all the customer numbers memorized. You can add a lookup field to `Table1` that looks into the `CUSTOMER` table and then, based on the customer number, retrieves the name of the current customer.



OrderNo	CustNo	ItemsTotal	WholesaleTotal	CustName
#1003	CN 1351	$1,250.00	$850.00	Phyllis Spooner
#1004	CN 2156	$7,885.00	$5,361.80	Tanya Wagner
#1005	CN 1356	$4,807.00	$3,268.76	Chris Thomas
#1006	CN 1380	$31,987.00	$21,751.16	Ernest Barratt
#1007	CN 1384	$6,500.00	$4,420.00	Russell Christo
#1008	CN 1510	$1,449.50	$985.66	Paul Gardner
#1009	CN 1513	$5,587.00	$3,799.16	Susan Wong
#1010	CN 1551	$4,996.00	$3,397.28	Joyce Marsh
#1011	CN 1560	$2,679.85	$1,822.30	Sam Witherspoon
#1012	CN 1563	$5,201.00	$3,536.68	Theresa Kunec
#1013	CN 1624	$3,115.00	$2,118.20	Donna Siaus

FIGURE 28.6
The calculated field has been added to the table.

First, you should drop in a second `TTable` object, setting its `DatabaseName` property to `DBDEMOS` and its `TableName` property to `CUSTOMER`. This is `Table2`. Then you once again select New Field from the Fields Editor local menu to invoke the New Field dialog. This time you'll call the field `CustName`, and the field type will be a `String`. The size of the string is 15 characters. Don't forget to select the Lookup button in the Field Type radio group. The Dataset control in this dialog should be set to `Table2`—the dataset you want to look into. The Key Fields and Lookup Keys controls should be set to `CustNo`—this is the common field upon which the lookup will be performed. Finally, the Result field should be set to `Contact`—this is the field you want displayed. Figure 28.7 shows the New Field dialog for the new lookup field. The new field will now display the correct data, as shown in the completed project in Figure 28.8.

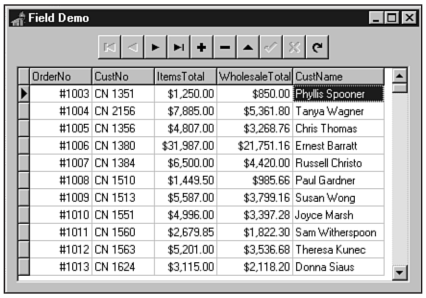


Field properties
Name: WholeSaleTotal Component: Table1WholeSaleTotal2
Type: Currency Size: 0

Field type
Data [✓ Calculated] Lookup

Lookup definition
Key Fields: Dataset:
Lookup Keys: Result Field:

FIGURE 28.7
Adding a lookup field with the New Field dialog.



OrderNo	CustNo	ItemsTotal	WholesaleTotal	CustName
#1003	CN 1351	$1,250.00	$850.00	Phyllis Spooner
#1004	CN 2156	$7,885.00	$5,361.80	Tanya Wagner
#1005	CN 1356	$4,807.00	$3,268.76	Chris Thomas
#1006	CN 1380	$31,987.00	$21,751.16	Ernest Barratt
#1007	CN 1384	$6,500.00	$4,420.00	Russell Christo
#1008	CN 1510	$1,449.50	$985.66	Paul Gardner
#1009	CN 1513	$5,587.00	$3,799.16	Susan Wong
#1010	CN 1551	$4,996.00	$3,397.28	Joyce Marsh
#1011	CN 1560	$2,679.85	$1,822.30	Sam Witherspoon
#1012	CN 1563	$5,201.00	$3,536.68	Theresa Kunec
#1013	CN 1624	$3,115.00	$2,118.20	Donna Siaus

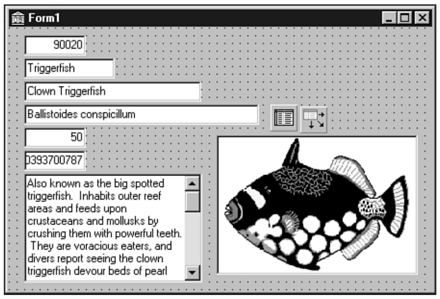
FIGURE 28.8

Viewing the table containing a lookup field.

Drag-and-Drop Fields

Another less obvious feature of the Fields Editor is that it enables you to drag fields from its Fields list box and drop them onto your forms. We can easily demonstrate this feature by starting a new project that contains only a `TTable` on the main form. Assign `Table1.DatabaseName` to `DBDEMOS` and assign `Table1.TableName` to `BIOLIFE.DB`. Invoke the Fields Editor for this table and add all the fields in the table to the Fields Editor list box. You can now drag one or more of the fields at a time from the Fields Editor window and drop them on your main form.

You'll notice a couple of cool things happening here: First, Delphi senses what kind of field you're dropping onto your form and creates the appropriate data-aware control to display the data (that is, a `TDBEdit` is created for a string field, whereas a `TDBImage` is created for a graphic field). Second, Delphi checks to see if you have a `TDataSource` object connected to the dataset; it hooks to an existing one if available or creates one if needed. Figure 28.9 shows the result of dragging and dropping the fields of the `BIOLIFE` table onto a form.



90020

Triggerfish

Clown Triggerfish

Ballistoides conspicillum

50

0393700787

Also known as the big spotted triggerfish. Inhabits outer reef areas and feeds upon crustaceans and mollusks by crushing them with powerful teeth. They are voracious eaters, and divers report seeing the clown triggerfish devour beds of pearl

FIGURE 28.9

Dragging and dropping fields on a form.

Working with BLOB Fields

A BLOB (Binary Large Object) field is a field that's designed to contain an indeterminate amount of data. A BLOB field in one record of a dataset may contain three bytes of data, whereas the same field in another record of that dataset may contain 3K bytes. Blobs are most useful for holding large amounts of text, graphic images, or raw data streams such as OLE objects.

TBlobField and Field Types

As discussed earlier, VCL includes a `TField` descendant called `TBlobField`, which encapsulates a BLOB field. `TBlobField` has a `BlobType` property of type `TBlobType`, which indicates what type of data is stored in the BLOB field. `TBlobType` is defined in the `DB` unit as follows:

```
TBlobType = ftBlob..ftOraClob;
```

All these field types and the type of data associated with these field types are listed in Table 28.3.

TABLE 28.3 `TBlobField` Field Types

Field Type	*Type of Data*
`ftBlob`	Untyped or user-defined data
`ftMemo`	Text
`ftGraphic`	Windows bitmap
`ftFmtMemo`	Paradox formatted memo
`ftParadoxOle`	Paradox OLE object
`ftDBaseOLE`	dBASE OLE object
`ftTypedBinary`	Raw data representation of an existing type
`ftCursor..ftDataSet`	Not valid BLOB types
`ftOraBlob`	BLOB fields in Oracle8 tables
`ftOraClob`	CLOB fields in Oracle8 tables

You'll find that most of the work you need to do in getting data in and out of `TBlobField` components can be accomplished by loading or saving the BLOB to a file or by using a `TBlobStream`. `TBlobStream` is a specialized descendant of `TStream` that uses the BLOB field inside the physical table as the stream location. To demonstrate these techniques for interacting with `TBlobField` components, you'll create a sample application.

NOTE

If you ran the Setup program on the CD-ROM accompanying this book, it should have set up a BDE alias that points to the `\Data` subdirectory of the directory in which you installed the software. In this directory, you can find the tables used in the applications throughout this book. Several of the examples on the CD-ROM expect the `DDGData` alias.

BLOB Field Example

This project creates an application that enables the user to store WAV files in a database table and play them directly from the table. Start the project by creating a main form with the components shown in Figure 28.10. The `TTable` component can map to the `Wavez` table in the `DDGUtils` alias or your own table of the same structure. The structure of the table is as follows:

Field Name	*Field Type*	*Size*
`WaveTitle`	Character	25
`FileName`	Character	25
`Wave`	BLOB	

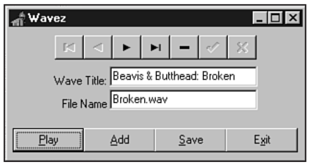


FIGURE 28.10
Main form for Wavez, the BLOB field example.

The Add button is used to load a WAV file from disk and add it to the table. The method assigned to the `OnClick` event of the Add button is shown here:

```
procedure TMainForm.sbAddClick(Sender: TObject);
begin
  if OpenDialog.Execute then
  begin
    tblSounds.Append;
    tblSounds['FileName'] := ExtractFileName(OpenDialog.FileName);
    tblSoundsWave.LoadFromFile(OpenDialog.FileName);
    edTitle.SetFocus;
  end;
end;
```

The code first attempts to execute `OpenDialog`. If it's successful, `tblSounds` is put into Append mode, the `FileName` field is assigned a value, and the `Wave` BLOB field is loaded from the file specified by `OpenDialog`. Notice that `TBlobField`'s `LoadFromFile` method is very handy here, and the code is very clean for loading a file into a BLOB field.

Similarly, the Save button saves the current WAV sound found in the `Wave` field to an external file. The code for this button is as follows:

```
procedure TMainForm.sbSaveClick(Sender: TObject);
begin
  with SaveDialog do
  begin
    FileName := tblSounds['FileName'];    // initialize file name
    if Execute then                       // execute dialog
      tblSoundsWave.SaveToFile(FileName); // save blob to file
  end;
end;
```

There's even less code here. `SaveDialog` is initialized with the value of the `FileName` field. If `SaveDialog`'s execution is successful, `tblSoundsWave`'s `SaveToFile` method is called to save the contents of the BLOB field to the file.

The handler for the Play button does the work of reading the WAV data from the BLOB field and passing it to the `PlaySound()` API function to be played. The code for this handler, shown next, is a bit more complex than the code shown thus far:

```
procedure TMainForm.sbPlayClick(Sender: TObject);
var
  B: TBlobStream;
  M: TMemoryStream;
begin
  B := TBlobStream.Create(tblSoundsWave, bmRead); // create blob stream
  Screen.Cursor := crHourGlass;                   // wait hourglass
  try
    M := TMemoryStream.Create;                    // create memory stream
    try
      M.CopyFrom(B, B.Size);             // copy from blob to memory stream
      // Attempt to play sound. Raise exception if something goes wrong
      Win32Check(PlaySound(M.Memory, 0, SND_SYNC or SND_MEMORY));
    finally
      M.Free;
    end;
  finally
    Screen.Cursor := crDefault;
    B.Free;                                       // clean up
  end;
end;
```

The first thing this method does is to create an instance of `TBlobStream`, `B`, using the `tblSoundsWave` BLOB field. The first parameter passed to `TBlobStream.Create()` is the BLOB field object, and the second parameter indicates how you want to open the stream. Typically, you'll use `bmRead` for read-only access to the BLOB stream or `bmReadWrite` for read/write access.

TIP

The dataset must be in Edit, Insert, or Append mode to open a `TBlobStream` with `bmReadWrite` privilege.

An instance of `TMemoryStream`, `M`, is then created. At this point, the cursor shape is changed to an hourglass to let the user know that the operation may take a couple of seconds. The stream `B` is then copied to the stream `M`. The function used to play a WAV sound, `PlaySound()`, requires a filename or a memory pointer as its first parameter. `TBlobStream` doesn't provide pointer access to the stream data, but `TMemoryStream` does through its `Memory` property. Given that, you can successfully call `PlaySound()` to play the data pointed at by `M.Memory`. Once the function is called, it cleans up by freeing the streams and restoring the cursor. The complete code for the main unit of this project is shown in Listing 28.2.

LISTING 28.2 The Main Unit for the Wavez Project

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, DBCtrls, DB, DBTables, StdCtrls, Mask, Buttons, ComCtrls;

type
  TMainForm = class(TForm)
    tblSounds: TTable;
    dsSounds: TDataSource;
    tblSoundsWaveTitle: TStringField;
    tblSoundsWave: TBlobField;
    edTitle: TDBEdit;
    edFileName: TDBEdit;
    Label1: TLabel;
    Label2: TLabel;
    OpenDialog: TOpenDialog;
    tblSoundsFileName: TStringField;
    SaveDialog: TSaveDialog;
    pnlToobar: TPanel;
    sbPlay: TSpeedButton;
    sbAdd: TSpeedButton;
    sbSave: TSpeedButton;
    sbExit: TSpeedButton;
    Bevel1: TBevel;
    dbnNavigator: TDBNavigator;
    stbStatus: TStatusBar;
    procedure sbPlayClick(Sender: TObject);
    procedure sbAddClick(Sender: TObject);
    procedure sbSaveClick(Sender: TObject);
    procedure sbExitClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    procedure OnAppHint(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses MMSystem;

procedure TMainForm.sbPlayClick(Sender: TObject);
var
  B: TBlobStream;
  M: TMemoryStream;
begin
  B := TBlobStream.Create(tblSoundsWave, bmRead); // create blob stream
  Screen.Cursor := crHourGlass;                   // wait hourglass
  try
    M := TMemoryStream.Create;                    // create memory stream
    try
      M.CopyFrom(B, B.Size);             // copy from blob to memory stream
      // Attempt to play sound.  Show error box if something goes wrong
      Win32Check(PlaySound(M.Memory, 0, SND_SYNC or SND_MEMORY));
    finally
      M.Free;
    end;
  finally
    Screen.Cursor := crDefault;
    B.Free;                                       // clean up
```

continues

LISTING 28.2 Continued

```
  end;
end;

procedure TMainForm.sbAddClick(Sender: TObject);
begin
  if OpenDialog.Execute then
  begin
    tblSounds.Append;
    tblSounds['FileName'] := ExtractFileName(OpenDialog.FileName);
    tblSoundsWave.LoadFromFile(OpenDialog.FileName);
    edTitle.SetFocus;
  end;
end;

procedure TMainForm.sbSaveClick(Sender: TObject);
begin
  with SaveDialog do
  begin
    FileName := tblSounds['FileName'];    // initialize file name
    if Execute then                       // execute dialog
      tblSoundsWave.SaveToFile(FileName); // save blob to file
  end;
end;

procedure TMainForm.sbExitClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  Application.OnHint := OnAppHint;
end;

procedure TMainForm.OnAppHint(Sender: TObject);
begin
  stbStatus.SimpleText := Application.Hint;
end;

end.
```

Refreshing the Dataset

If there's one thing you can count on when you create database applications, it's that data contained in a dataset is in a constant state of flux. Records will constantly be added to, removed from, and modified in your dataset, particularly in a networked environment. Because of this, you may occasionally need to reread the dataset information from disk or memory to update the contents of your dataset.

You can update your dataset using `TDataset`'s `Refresh()` method. It functionally does about the same thing as using `Close()` and then `Open()` on the dataset, but `Refresh()` is a bit faster. The `Refresh()` method works with all local tables; however, some restrictions apply when using `Refresh()` with a database from a SQL database server.

`TTable components` connected to SQL databases must have a unique index before the BDE will attempt a `Refresh()` operation. This is because `Refresh()` tries to preserve the current record, if possible. This means that the BDE has to use `Seek()` to go to the current record at some point, which is practical only on a SQL dataset if a unique index is available. `Refresh()` does not work for `TQuery` components connected to SQL databases.

CAUTION

When `Refresh()` is called, it can create some unexpected side effects for the users of your program. For example, if user 1 is viewing a record on a networked table, and that record has been deleted by user 2, a call to `Refresh()` will cause user 1 to see the record disappear for no apparent reason. The fact that data could be changing beneath the user is something you need to keep in mind when you call this function.

Altered States

At some point, you may need to know whether a table is in Edit mode or Append mode, or even if it's active. You can obtain this information by inspecting `TDataset`'s `State` property. The `State` property is of type `TDataSetState`, and it can have any one of the values shown in Table 28.4.

TABLE 28.4 Values for `TDataSet.State`

Value	*Meaning*
`dsBrowse`	The dataset is in Browse (normal) mode.
`dsCalcFields`	The `OnCalcFields` event has been called, and a record value calculation is in progress.
`dsEdit`	The dataset is in Edit mode. This means the `Edit()` method has been called, but the edited record has not yet been posted.
`dsInactive`	The dataset is closed.

continues

TABLE 28.4 Continued

Value	*Meaning*
`dsInsert`	The dataset is in Insert mode. This typically means that `Insert()` has been called but changes haven't been posted.
`dsSetKey`	The dataset is in SetKey mode, meaning that `SetKey()` has been called but `GotoKey()` hasn't yet been called.
`dsNewValue`	The dataset is in a temporary state where the `NewValue` property is being accessed.
`dsOldValue`	The dataset is in a temporary state where the `OldValue` property is being accessed.
`dsCurValue`	The dataset is in a temporary state where the `OldValue` property is being accessed.
`dsFilter`	The dataset is currently processing a record filter, lookup, or some other operation that requires a filter.
`dsBlockRead`	Data is being buffered en masse, so data-aware controls are not updated and events are not triggered when the cursor moves while this member is set.
`dsInternalCalc`	A field value is currently being calculated for a field that has a `FieldKind` of `fkInternalCalc`.
`dsOpening`	DataSet is in the process of opening but has not finished. This state occurs when the dataset is opened for asynchronous fetching.

Filters

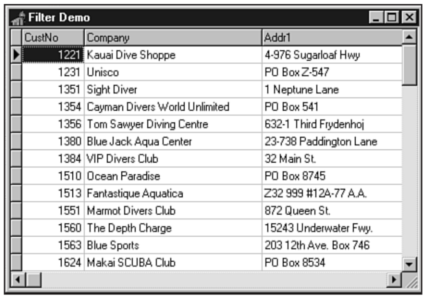
Filters enable you to do simple dataset searching or filtering using only Object Pascal code. The primary advantage of using filters is that they don't require an index or any other preparation on the datasets with which they're used. In many cases, filters can be a bit slower than index-based searching (which is covered later in this chapter), but they're still very usable in almost any type of application.

Filtering a Dataset

One of the more common uses of Delphi's filtering mechanism is to limit a view of a dataset to some specific records only. This is a simple two-step process:

1. Assign a procedure to the dataset's `OnFilterRecord` event. Inside of this procedure, you should write code that accepts records based on the values of one or more fields.
2. Set the dataset's `Filtered` property to `True`.

As an example, Figure 28.11 shows a form containing `TDBGrid`, which displays an unfiltered view of Delphi's `CUSTOMER` table.



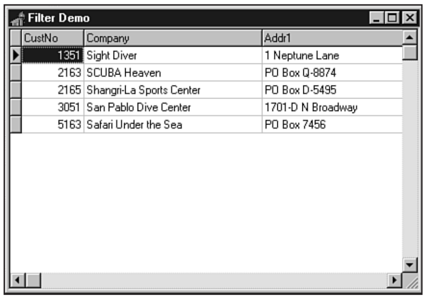
CustNo	Company	Addr1
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy
1231	Unisco	PO Box Z-547
1351	Sight Diver	1 Neptune Lane
1354	Cayman Divers World Unlimited	PO Box 541
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj
1380	Blue Jack Aqua Center	23-738 Paddington Lane
1384	VIP Divers Club	32 Main St.
1510	Ocean Paradise	PO Box 8745
1513	Fantastique Aquatica	Z32 999 #12A-77 A.A.
1551	Marmot Divers Club	872 Queen St.
1560	The Depth Charge	15243 Underwater Fwy.
1563	Blue Sports	203 12th Ave. Box 746
1624	Makai SCUBA Club	PO Box 8534

FIGURE 28.11
An unfiltered view of the `CUSTOMER` *table.*

In step 1, you write a handler for the table's `OnFilterRecord` event. In this case, we'll accept only records whose `Company` field starts with the letter *S*. The code for this procedure is shown here:

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet;
  var Accept: Boolean);
var
  FieldVal: String;
begin
  FieldVal := DataSet['Company'];  // Get the value of the Company field
  Accept := FieldVal[1] = 'S';     // Accept record if field starts with 'S'
end;
```

After following step 2 and setting the table's `Filtered` property to `True`, you can see in Figure 28.12 that the grid displays only those records that meet the filter criteria.



CustNo	Company	Addr1
1351	Sight Diver	1 Neptune Lane
2163	SCUBA Heaven	PO Box Q-8874
2165	Shangri-La Sports Center	PO Box D-5495
3051	San Pablo Dive Center	1701-D N Broadway
5163	Safari Under the Sea	PO Box 7456

FIGURE 28.12
A filtered view of the `CUSTOMER` *table.*

NOTE

The OnFilterRecord event should only be used in cases where the filter cannot be expressed in the Filter property. The reason for this is that it can provide significant performance benefits. On SQL databases, for example, the TTable component will pass the contents of the FILTER property in a WHERE clause to the database, which is generally much faster than the record-by-record search performed in OnFilterRecord.

FindFirst/FindNext

TDataSet also provides methods called FindFirst(), FindNext(), FindPrior(), and FindLast() that employ filters to find records that match a particular search criteria. All these functions work on unfiltered datasets by calling that dataset's OnFilterRecord event handler. Based on the search criteria in the event handler, these functions will find the first, next, previous, or last match, respectively. Each of these functions accepts no parameters and returns a Boolean, which indicates whether a match was found.

Locating a Record

Not only are filters useful for defining a subset view of a particular dataset, but they can also be used to search for records within a dataset based on the value of one or more fields. For this purpose, TDataSet provides a method called Locate(). Once again, because Locate() employs filters to do the searching, it will work irrespective of any index applied to the dataset. The Locate() method is defined as follows:

```
function Locate(const KeyFields: string; const KeyValues: Variant;
  Options: TLocateOptions): Boolean;
```

The first parameter, KeyFields, contains the name of the field(s) on which you want to search. The second parameter, KeyValues, holds the field value(s) you want to locate. The third and last parameter, Options, allows you to customize the type of search you want to perform. This parameter is of type TLocateOptions, which is a set type defined in the DB unit as follows:

```
type
  TLocateOption = (loCaseInsensitive, loPartialKey);
  TLocateOptions = set of TLocateOption;
```

If the set includes the loCaseInsensitive member, a case-insensitive search of the data will be performed. If the set includes the loPartialKey member, the values contained in KeyValues will match even if they're substrings of the field value.

`Locate()` will return `True` if it finds a match. For example, to search for the first occurrence of the value `1356` in the `CustNo` field of `Table1`, use the following syntax:

```
Table1.Locate('CustNo', 1356, []);
```

TIP

You should use `Locate()` whenever possible to search for records, because it will always attempt to use the fastest method possible to find the item, switching indexes temporarily if necessary. This makes your code independent of indexes. Also, if you determine that you no longer need an index on a particular field, or if adding one will make your program faster, you can make that change on the data without having to recode the application.

Using TTable

This section describes the common properties and methods of the `TTable` component and how to use them. In particular, you learn how to search for records, filter records using ranges, and create tables. This section also contains a discussion of `TTable` events.

Searching for Records

When you need to search for records in a table, VCL provides several methods to help you out. When you're working with dBASE and Paradox tables, Delphi assumes that the fields on which you search are indexed. For SQL tables, the performance of your search will suffer if you search on unindexed fields.

Say, for example, you have a table that's keyed on field 1, which is numeric, and on field 2, which is alphanumeric. You can search for a specific record based on those two criteria in one of two ways: using the `FindKey()` technique or the `SetKey()..GotoKey()` technique.

FindKey()

`TTable`'s `FindKey()` method enables you to search for a record matching one or more keyed fields in one function call. `FindKey()` accepts an `array of const` (the search criteria) as a parameter and returns `True` when it's successful. For example, the following code causes the dataset to move to the record where the first field in the index has the value `123` and the second field in the index contains the string `Hello`:

```
if not Table1.FindKey([123, 'Hello']) then MessageBeep(0);
```

If a match is not found, `FindKey()` returns `False` and the computer beeps.

SetKey()..GotoKey()

Calling `TTable`'s `SetKey()` method puts the table in a mode that prepares its fields to be loaded with values representing search criteria. Once the search criteria have been established, use the `GotoKey()` method to do a top-down search for a matching record. The previous example can be rewritten with `SetKey()..GotoKey()`, as follows:

```
with Table1 do begin
  SetKey;
  Fields[0].AsInteger := 123;
  Fields[1].AsString := 'Hello';
  if not GotoKey then MessageBeep(0);
end;
```

The Closest Match

Similarly, you can use `FindNearest()` or the `SetKey..GotoNearest` methods to search for a value in the table that's the closest match to the search criteria. To search for the first record where the value of the first indexed field is closest to (greater than or equal to) `123`, use the following code:

```
Table1.FindNearest([123]);
```

Once again, `FindNearest()` accepts an `array of const` as a parameter that contains the field values for which you want to search.

To search using the longhand technique provided by `SetKey()..GotoNearest()`, you can use this code:

```
with Table1 do begin
  SetKey;
  Fields[0].AsInteger := 123;
  GotoNearest;
end;
```

If the search is successful and the table's `KeyExclusive` property is set to `False`, the record pointer will be on the first matching record. If `KeyExclusive` is `True`, the current record will be the one immediately following the match.

TIP

If you want to search on the indexed fields of a table, use `FindKey()` and `FindNearest()`—rather than `SetKey()..Goto`*`X`*`()`—whenever possible because you type less code and leave less room for human error.

Which Index?

All these searching methods assume that you're searching under the table's primary index. If you want to search using a secondary index, you need to set the table's `IndexName` parameter to the desired index. For instance, if your table had a secondary index on the `Company` field called `ByCompany`, the following code would enable you to search for the company "Unisco":

```
with Table1 do begin
  IndexName := 'ByCompany';
  SetKey;
  FieldValues['Company'] := 'Unisco';
  GotoKey;
end;
```

NOTE

Keep in mind that some overhead is involved in switching indexes while a table is opened. You should expect a delay of a second or more when you set the `IndexName` property to a new value.

Ranges enable you to filter a table so that it contains only records with field values that fall within a certain scope you define. Ranges work similar to key searches, and as with searches, there are several ways to apply a range to a given table—either using the `SetRange()` method or the manual `SetRangeStart()`, `SetRangeEnd()`, and `ApplyRange()` methods.

CAUTION

If you are working with dBASE or Paradox tables, ranges only work with indexed fields. If you're working with SQL data, performance will suffer greatly if you don't have an index on the ranged field.

SetRange()

Like `FindKey()` and `FindNearest()`, `SetRange()` enables you to perform a fairly complex action on a table with one function call. `SetRange()` accepts two `array of const` variables as parameters: The first represents the field values for the start of the range, and the second represents the field values for the end of the range. As an example, the following code filters through only those records where the value of the first field is greater than or equal to 10 but less than or equal to 15:

```
Table1.SetRange([10], [15]);
```

ApplyRange()

To use the `ApplyRange()` method of setting a range, follow these steps:

1. Call the `SetRangeStart()` method and then modify the `Fields[]` array property of the table to establish the starting value of the keyed field(s).
2. Call the `SetRangeEnd()` method and modify the `Fields[]` array property once again to establish the ending value of the keyed field(s).
3. Call `ApplyRange()` to establish the new range filter.

The preceding range example could be rewritten using this technique:

```
with Table1 do begin
  SetRangeStart;
  Fields[0].AsInteger := 10;       // range starts at 10
  SetRangeEnd;
  Fields[0].AsInteger := 15;       // range ends at 15
  ApplyRange;
end;
```

TIP

Use `SetRange()` whenever possible to filter records—your code will be less prone to error when doing so.

To remove a range filter from a table and restore the table to the state it was in before you called `ApplyRange()` or `SetRange()`, just call `TTable`'s `CancelRange()` method.

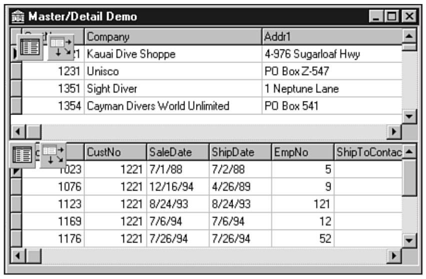
```
Table1.CancelRange;
```

Master/Detail Tables

Very often, when programming databases, you'll find situations where the data to be managed lends itself to being broken up into multiple tables that relate to one another. The classic example is a customer table with one record per customer information and an orders table with one record per order. Because every order would have to be made by one of the customers, a natural relationship forms between the two collections of data. This is called a *one-to-many* relationship, because one customer may have many orders (the customer table being the *master* and the orders table being the *detail*).

Delphi makes it easy to create these types of relationships between tables. In fact, it's all handled at design time through the Object Inspector; therefore, it's not even necessary for you to write any code. Start with an empty project and add two each of the `TTable`, `TDataSource`, and

`TDBGrid` components. `DBGrid1` will hook to `Table1` via `DataSource1`, and `DBGrid2` hooks to `Table2` via `DataSource2`. Using the `DBDEMOS` alias as the `DatabaseName`, `Table1` hooks to the `CUSTOMER.DB` table, and `Table2` hooks to the `ORDERS.DB` table. Your form should look like the one shown in Figure 28.13.

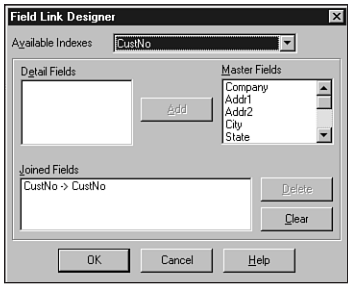


	Company	Addr1
21	Kauai Dive Shoppe	4-976 Sugarloaf Hwy
1231	Unisco	PO Box Z-547
1351	Sight Diver	1 Neptune Lane
1354	Cayman Divers World Unlimited	PO Box 541

	CustNo	SaleDate	ShipDate	EmpNo	ShipToContac
1023	1221	7/1/88	7/2/88	5	
1076	1221	12/16/94	4/26/89	9	
1123	1221	8/24/93	8/24/93	121	
1169	1221	7/6/94	7/6/94	12	
1176	1221	7/26/94	7/26/94	52	

FIGURE 28.13
The master/detail main form in progress.

You now have two unrelated tables sharing the same form. Once you've come this far, the only thing left to do is to create the relationship between the tables using the `MasterSource` and `MasterFields` properties of the detail table. `Table2`'s `MasterSource` property should be set to `DataSource1`. When you attempt to edit the `MasterFields` property, you are presented with a property editor called the Field Link Designer. This is shown in Figure 28.14.



Available Indexes: CustNo

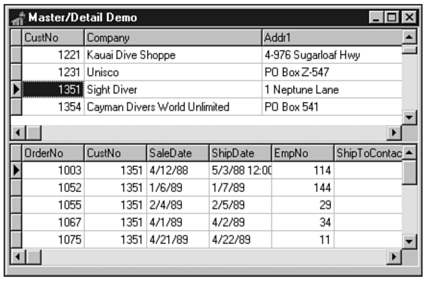
Joined Fields: CustNo -> CustNo

FIGURE 28.14
The Field Link Designer.

In this dialog, you specify which common fields relate the two tables to one another. The field the two tables have in common is `CustNo`—a numeric identifier that represents a customer. Because the `CustNo` field is not a part of the `ORDERS` table's primary index, you'll need to switch to a secondary index that does include the `CustNo` field. You can do that using the Available Indexes drop-down list in the Field Link Designer. Once you've switched to the

`CustNo` index, you can then select the `CustNo` field from both the Detail Fields and Master Fields list boxes and click the Add button to create a link between the tables. Click OK to dismiss the Field Link Designer.

You'll now notice that as you move through the records in `Table1`, the view of `Table2` will be limited to only those records that share the same value in the `CustNo` field as `Table1`. The behavior is shown in the finished application in Figure 28.15.



CustNo	Company	Addr1
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy
1231	Unisco	PO Box Z-547
1351	Sight Diver	1 Neptune Lane
1354	Cayman Divers World Unlimited	PO Box 541

OrderNo	CustNo	SaleDate	ShipDate	EmpNo	ShipToContac
1003	1351	4/12/88	5/3/88 12:0(	114	
1052	1351	1/6/89	1/7/89	144	
1055	1351	2/4/89	2/5/89	29	
1067	1351	4/1/89	4/2/89	34	
1075	1351	4/21/89	4/22/89	11	

FIGURE 28.15
Master/detail demo program.

TTable Events

`TTable` provides you with events that occur before and after a record in the table is deleted, edited, and inserted, whenever a modification is posted or canceled, and whenever the table is opened or closed. This is so that you have full control of your database application. The nomenclature for these events is `Before`*`XXX`* and `After`*`XXX`*, where *`XXX`* stands for `Delete`, `Edit`, `Insert`, `Open`, and so on. These events are fairly self-explanatory, and you'll use them in the database applications in Parts II, "Advanced Techniques," and III, "Component-Based Development," of this book.

`TTable`'s `OnNewRecord` event fires every time a new record is posted to the table. It's ideal to perform various housekeeping tasks in a handler for this event. An example of this would be to keep a running total of records added to a table.

The `OnCalcFields` event occurs whenever the table cursor is moved off the current record or the current record changes. Adding a handler for the `OnCalcFields` event enables you to keep a calculated field current whenever the table is modified.

Creating a Table in Code

Instead of creating all your database tables up front (using the Database Desktop, for example) and deploying them with your application, a time will come when you'll need your program to have the capability to create local tables for you. When this need arises, once again VCL has you covered. `TTable` contains the `CreateTable()` method, which enables you to create tables on disk. Simply follow these steps to create a table:

1. Create an instance of `TTable`.
2. Set the `DatabaseName` property of the table to a directory or existing alias.
3. Give the table a unique name in the `TableName` property.
4. Set the `TableType` property to indicate what type of table you want to create. If you set this property to `ttDefault`, the table type will correspond to the extension of the name provided in the `TableName` property (for example, DB stands for Paradox, and DBF stands for dBASE).
5. Use `Add()` method for `TTable.FieldDefs` to add fields to the table. The `Add()` method takes four parameters:
   - A string indicating the field name.
   - A `TFieldType` variable indicating the field type.
   - A `word` parameter that represents the size of the field. Note that this parameter is only valid for types such as `String` and `Memo`, where the size may vary. Fields such as `Integer` and `Date` are always the same size, so this parameter doesn't apply to them.
   - A Boolean parameter that dictates whether this is a required field. All required fields must have a value before a record can be posted to a table.
6. If you want the table to have an index, use the `Add()` method of `TTable.IndexDefs` to add indexed fields. `IndexDefs.Add()` takes the following three parameters:
   - A string that identifies the index.
   - A string that matches the field name to be indexed. Composite key indexes (indexes on multiple fields) can be specified as a semicolon-delimited list of field names.
   - A set of `TIndexOptions` that determines the index type.
7. Call `TTable.CreateTable()`.

The following code creates a table with `Integer`, `String`, and `Float` fields with an index on the `Integer` field. The table is called `FOO.DB`, and it will live in the `C:\TEMP` directory:

```
begin
  with TTable.Create(Self) do begin       // create TTable object
    DatabaseName := 'c:\temp';            // point to directory or alias
    TableName := 'FOO';                   // give table a name
    TableType := ttParadox;               // make a Paradox table
    with FieldDefs do begin
```

```
      Add('Age', ftInteger, 0, True);     // add an integer field
      Add('Name', ftString, 25, False);   // add a string field
      Add('Weight', ftFloat, 0, False);   // add a floating-point field
    end;
    { create a primary index on the Age field... }
    IndexDefs.Add('', 'Age', [ixPrimary, ixUnique]);
    CreateTable;                          // create the table
  end;
end;
```

NOTE

As mentioned earlier, `TTable.CreateTable()` works only for local tables. For SQL tables, you should use a technique that employs `TQuery` (this is shown in the next chapter).

Data Modules

Data modules enable you to keep all your database rules and relationships in one central location to be shared across projects, groups, or enterprises. Data modules are encapsulated by VCL's `TDataModule` component. Think of `TDataModule` as an invisible form on which you can drop data-access components to be used throughout a project. Creating a `TDataModule` instance is simple: Select File, New from the main menu and then select Data Module from the Object Repository.

The simple justification for using `TDataModule` over just putting data-access components on a form is that it's easier to share the same data across multiple forms and units in your project. In a more complex situation, you would have an arrangement of multiple `TTable`, `TQuery`, and/or `TStoredProc` components. You might have relationships defined between the components and perhaps rules enforced on the field level, such as minimum/maximum values or display formats. Perhaps this assortment of data-access components models the business rules of your enterprise. After taking great pains to set up something so impressive, you wouldn't want to have to do it again for another application, would you? Of course you wouldn't. In such cases, you would want to save your data module to the Object Repository for later use. If you work in a team environment, you might even want to keep the Object Repository on a shared network drive for the use of all the developers on your team.

In the example that follows, you'll create a simple instance of a data module so that many forms have access to the same data. In the database applications shown in several of the later chapters, you'll build more complex relationships into data modules.

The Search, Range, and Filter Demo

Now it's time to create a sample application to help drive home some of the key concepts that were covered in this chapter. In particular, this application will demonstrate the proper use of filters, key searches, and range filters in your applications. This project, called SRF, contains multiple forms. The main form consists mainly of a grid for browsing a table, and other forms demonstrate the different concepts mentioned earlier. Each of these forms will be explained in turn.

The Data Module

Although we're starting a bit out of order, the data module for this project will be covered first. This data module, called `DM`, contains only a `TTable` and a `TDataSource` component. The `TTable`, called `Table1`, is hooked to the `CUSTOMERS.DB` table in the `DBDEMOS` alias. The `TDataSource`, `DataSource1`, is wired to `Table1`. All the data-aware controls in this project will use `DataSource1` as their `DataSource`. `DM` is contained in a unit called `DataMod`, and it's shown in its design-time state in Figure 28.16.

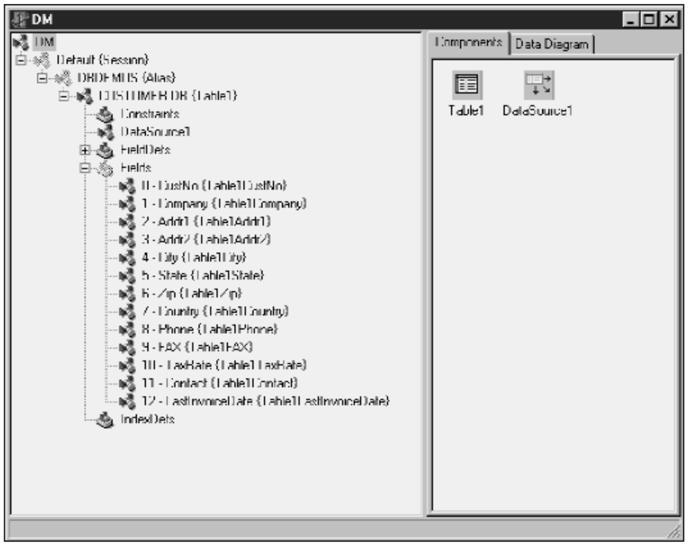


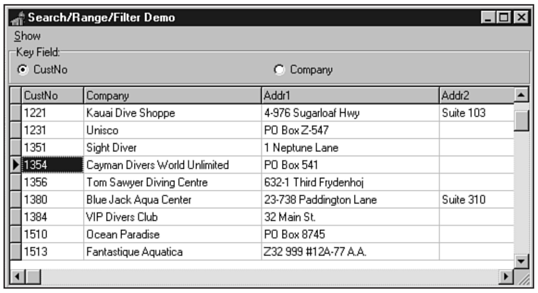
FIGURE 28.16
`DM`, *the data module.*

The Main Form

The main form for SRF, appropriately called `MainForm`, is shown in Figure 28.17. This form is contained in a unit called `Main`. As you can see, it contains a `TDBGrid` control, `DBGrid1`, for browsing a table, and it contains a radio button that enables you to switch between different indexes on the table. `DBGrid1`, as explained earlier, is hooked to `DM.DataSource1` as its data source.

NOTE

In order for `DBGrid1` to be able to hook to `DM.DataSource1` at design time, the `DataMod` unit must be in the `uses` clause of the `Main` unit. The easiest way to do this is to bring up the `Main` unit in the Code Editor and select File, Use Unit from the main menu. You'll then be presented with a list of units in your project from which you can select `DataMod`. You must do this for each of the units from which you want to access the data contained within `DM`.



CustNo	Company	Addr1	Addr2
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103
1231	Unisco	PO Box Z-547	
1351	Sight Diver	1 Neptune Lane	
1354	Cayman Divers World Unlimited	PO Box 541	
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj	
1380	Blue Jack Aqua Center	23-738 Paddington Lane	Suite 310
1384	VIP Divers Club	32 Main St.	
1510	Ocean Paradise	PO Box 8745	
1513	Fantastique Aquatica	Z32 999 #12A-77 A.A.	

FIGURE 28.17
`MainForm` *in the SRF project.*

The radio group, called `RGKeyField`, is used to determine which of the table's two indexes is currently active. The code attached to the `OnClick` event for `RGKeyField` is shown here:

```
procedure TMainForm.RGKeyFieldClick(Sender: TObject);
begin
  case RGKeyField.ItemIndex of
    0: DM.Table1.IndexName := '';           // primary index
    1: DM.Table1.IndexName := 'ByCompany';  // secondary, by company
  end;
end;
```

`MainForm` also contains a `TMainMenu` component, `MainMenu1`, which enables you to open and close each of the other forms. The items on this menu are Key Search, Range, Filter, and Exit. The `Main` unit, in its entirety, is shown in Listing 28.3.

LISTING 28.3 The Source Code for `MAIN.PAS`

```
unit Main;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls, Grids, DBGrids, DB, DBTables,
  Buttons, Mask, DBCtrls, Menus;

type
  TMainForm = class(TForm)
    DBGrid1: TDBGrid;
    RGKeyField: TRadioGroup;
    MainMenu1: TMainMenu;
    Forms1: TMenuItem;
    KeySearch1: TMenuItem;
    Range1: TMenuItem;
    Filter1: TMenuItem;
    N1: TMenuItem;
    Exit1: TMenuItem;
    procedure RGKeyFieldClick(Sender: TObject);
    procedure KeySearch1Click(Sender: TObject);
    procedure Range1Click(Sender: TObject);
    procedure Filter1Click(Sender: TObject);
    procedure Exit1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation

uses DataMod, KeySrch, Rng, Fltr;

{$R *.DFM}

procedure TMainForm.RGKeyFieldClick(Sender: TObject);
begin
  case RGKeyField.ItemIndex of
    0: DM.Table1.IndexName := '';           // primary index
    1: DM.Table1.IndexName := 'ByCompany';  // secondary, by company
```

continues

LISTING 28.3 Continued

```
  end;
end;

procedure TMainForm.KeySearch1Click(Sender: TObject);
begin
  KeySearch1.Checked := not KeySearch1.Checked;
  KeySearchForm.Visible := KeySearch1.Checked;
end;

procedure TMainForm.Range1Click(Sender: TObject);
begin
  Range1.Checked := not Range1.Checked;
  RangeForm.Visible := Range1.Checked;
end;

procedure TMainForm.Filter1Click(Sender: TObject);
begin
  Filter1.Checked := not Filter1.Checked;
  FilterForm.Visible := Filter1.Checked;
end;

procedure TMainForm.Exit1Click(Sender: TObject);
begin
  Close;
end;

end.
```

The Range Form

`RangeForm` is shown in Figure 28.18. `RangeForm` is located in a unit called `Rng`. This form enables you to set a range on the data displayed in `MainForm` to limit the view of the table. Depending on the active index, the items you specify in the Range Start and Range End edit controls can be either numeric (the primary index) or text (the secondary index). Listing 28.4 shows the source code for `RNG.PAS`.

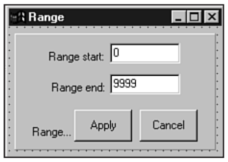


FIGURE 28.18
The `RangeForm` *form.*

LISTING 28.4 The Source Code for RNG.PAS

```
unit Rng;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls;

type
  TRangeForm = class(TForm)
    Panel1: TPanel;
    Label2: TLabel;
    StartEdit: TEdit;
    Label1: TLabel;
    EndEdit: TEdit;
    Label7: TLabel;
    ApplyButton: TButton;
    CancelButton: TButton;
    procedure ApplyButtonClick(Sender: TObject);
    procedure CancelButtonClick(Sender: TObject);
  private
    { Private declarations }
    procedure ToggleRangeButtons;
  public
    { Public declarations }
  end;

var
  RangeForm: TRangeForm;

implementation

uses DataMod;

{$R *.DFM}

procedure TRangeForm.ApplyButtonClick(Sender: TObject);
begin
  { Set range of records in dataset from StartEdit's value to EndEdit's }
  { value.  Strings are again implicitly converted to numerics.         }
  DM.Table1.SetRange([StartEdit.Text], [EndEdit.Text]);
  ToggleRangeButtons;                 // enable proper buttons
end;

procedure TRangeForm.CancelButtonClick(Sender: TObject);
```

continues

LISTING 28.4 Continued

```
begin
  DM.Table1.CancelRange;                 // remove set range
  ToggleRangeButtons;                 // enable proper buttons
end;

procedure TRangeForm.ToggleRangeButtons;
begin
  { Toggle the enabled property of the range buttons }
  ApplyButton.Enabled := not ApplyButton.Enabled;
  CancelButton.Enabled := not CancelButton.Enabled;
end;

end.
```

NOTE

Pay close attention to the following line of code from the `Rng` unit:

```
DM.Table1.SetRange([StartEdit.Text], [EndEdit.Text]);
```

You might find it strange that although the keyed field can be of either a `Numeric` type or `Text` type, you're always passing strings to the `SetRange()` method. Delphi allows this because `SetRange()`, `FindKey()`, and `FindNearest()` will perform the conversion from `String` to `Integer`, and vice versa, automatically.

What this means to you is that you should not bother calling `IntToStr()` or `StrToInt()` in these situations—it will be taken care of for you.

The Key Search Form

`KeySearchForm`, contained in the `KeySrch` unit, provides a means for the user of the application to search for a particular key value in the table. The form enables the user to search for a value in one of two ways. First, when the Normal radio button is selected, the user can search by typing text into the Search For edit control and pressing the Exact or Nearest button to find an exact match or closest match in the table. Second, when the Incremental radio button is selected, the user can perform an incremental search on the table every time he or she changes the text in the Search For edit control. The form is shown in Figure 28.19. The code for the `KeySrch` unit is shown in Listing 28.5.

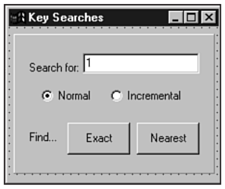


FIGURE 28.19
The `KeySearchForm` *form.*

LISTING 28.5 The Source Code for `KeySrch.PAS`

```
unit KeySrch;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls;

type
  TKeySearchForm = class(TForm)
    Panel1: TPanel;
    Label3: TLabel;
    SearchEdit: TEdit;
    RBNormal: TRadioButton;
    Incremental: TRadioButton;
    Label6: TLabel;
    ExactButton: TButton;
    NearestButton: TButton;
    procedure ExactButtonClick(Sender: TObject);
    procedure NearestButtonClick(Sender: TObject);
    procedure RBNormalClick(Sender: TObject);
    procedure IncrementalClick(Sender: TObject);
  private
    procedure NewSearch(Sender: TObject);
  end;

var
  KeySearchForm: TKeySearchForm;

implementation

uses DataMod;
```

continues

LISTING 28.5 Continued

```
{$R *.DFM}

procedure TKeySearchForm.ExactButtonClick(Sender: TObject);
begin
  { Try to find record where key field matches SearchEdit's Text value. }
  { Notice that Delphi handles the type conversion from the string      }
  { edit control to the numeric key field value.                        }
  if not DM.Table1.FindKey([SearchEdit.Text]) then
    MessageDlg(Format('Match for "%s" not found.', [SearchEdit.Text]),
               mtInformation, [mbOk], 0);
end;

procedure TKeySearchForm.NearestButtonClick(Sender: TObject);
begin
  { Find closest match to SearchEdit's Text value.  Note again the }
  { implicit type conversion.                                      }
  DM.Table1.FindNearest([SearchEdit.Text]);
end;

procedure TKeySearchForm.NewSearch(Sender: TObject);
{ This is the method which is wired to the SearchEdit's OnChange }
{ event whenever the Incremental radio is selected. }
begin
  DM.Table1.FindNearest([SearchEdit.Text]); // search for text
end;

procedure TKeySearchForm.RBNormalClick(Sender: TObject);
begin

  ExactButton.Enabled := True;   // enable search buttons
  NearestButton.Enabled := True;
  SearchEdit.OnChange := Nil;    // unhook the OnChange event
end;

procedure TKeySearchForm.IncrementalClick(Sender: TObject);
begin
  ExactButton.Enabled := False;      // disable search buttons
  NearestButton.Enabled := False;
  SearchEdit.OnChange := NewSearch;  // hook the OnChange event
  NewSearch(Sender);                 // search current text
end;

end.
```

The code for the `KeySrch` unit should be fairly straightforward to you. You might notice that, once again, we can safely pass text strings to the `FindKey()` and `FindNearest()` methods with the knowledge that they will do the right thing with regard to type conversion. You might also appreciate the small trick that's employed to switch to and from incremental searching on the fly. This is accomplished by either assigning a method to or assigning `Nil` to the `OnChange` event of the `SearchEdit` edit control. When assigned a handler method, the `OnChange` event will fire whenever the text in the control is modified. By calling `FindNearest()` inside that handler, an incremental search can be performed as the user types.

The Filter Form

The purpose of `FilterForm`, found in the `Fltr` unit, is two-fold. First, it enables the user to filter the view of the table to a set where the value of the `State` field matches that of the current record. Second, this form enables the user to search for a record where the value of any field in the table is equal to some value he or she has specified. This form is shown in Figure 28.20.

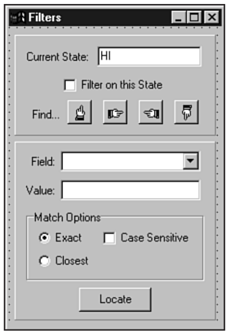


FIGURE 28.20
The `FilterForm` *form.*

The record-filtering functionality actually involves very little code. First, the state of the check box labeled Filter on this State (called `cbFiltered`) determines the setting of `DM.Table1`'s `Filtered` property. This is accomplished with the following line of code attached to `cbFiltered.OnClick`:

```
DM.Table1.Filtered := cbFiltered.Checked;
```

When `DM.Table1.Filtered` is `True`, `Table1` filters records using the following `OnFilterRecord` method, which is actually located in the `DataMod` unit:

```
procedure TDM.Table1FilterRecord(DataSet: TDataSet;
  var Accept: Boolean);
begin
  { Accept record as a part of the filter if the value of the State }
  { field is the same as that of DBEdit1.Text.                      }
  Accept := Table1State.Value = FilterForm.DBEdit1.Text;
end;
```

To perform the filter-based search, the `Locate()` method of `TTable` is employed:

```
DM.Table1.Locate(CBField.Text, EValue.Text, LO);
```

The field name is taken from a combo box called `CBField`. The contents of this combo box are generated in the `OnCreate` event of this form using the following code to iterate through the fields of `Table1`:

```
procedure TFilterForm.FormCreate(Sender: TObject);
var
  i: integer;
begin
  with DM.Table1 do begin
    for i := 0 to FieldCount - 1 do
      CBField.Items.Add(Fields[i].FieldName);
  end;
end;
```

TIP

The preceding code will only work when `DM` is created prior to this form. Otherwise, any attempts to access `DM` before it's created will probably result in an Access Violation error. To make sure that the data module, `DM`, is created prior to any of the child forms, we manually adjusted the creation order of the forms in the Autocreate Forms list on the Forms page of the Project Options dialog (found under Options, Project on the main menu).

The main form must, of course, be the first one created, but other than that, this little trick ensures that the data module gets created prior to any other form in the application.

The complete code for the `Fltr` unit is shown in Listing 28.6.

LISTING 28.6 The Source Code for `Fltr.pas`

```
unit Fltr;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Buttons, Mask, DBCtrls, ExtCtrls;

type
  TFilterForm = class(TForm)
    Panel1: TPanel;
    Label4: TLabel;
    DBEdit1: TDBEdit;
    cbFiltered: TCheckBox;
    Label5: TLabel;
    SpeedButton1: TSpeedButton;
    SpeedButton2: TSpeedButton;
    SpeedButton3: TSpeedButton;
    SpeedButton4: TSpeedButton;
    Panel2: TPanel;
    EValue: TEdit;
    LocateBtn: TButton;
    Label1: TLabel;
    Label2: TLabel;
    CBField: TComboBox;
    MatchGB: TGroupBox;
    RBExact: TRadioButton;
    RBClosest: TRadioButton;
    CBCaseSens: TCheckBox;
    procedure cbFilteredClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure LocateBtnClick(Sender: TObject);
    procedure SpeedButton1Click(Sender: TObject);
    procedure SpeedButton2Click(Sender: TObject);
    procedure SpeedButton3Click(Sender: TObject);
    procedure SpeedButton4Click(Sender: TObject);
  end;

var
  FilterForm: TFilterForm;

implementation

uses DataMod, DB;

{$R *.DFM}

procedure TFilterForm.cbFilteredClick(Sender: TObject);
begin
  { Filter table if checkbox is checked }
  DM.Table1.Filtered := cbFiltered.Checked;
end;

procedure TFilterForm.FormCreate(Sender: TObject);
var
  i: integer;
begin
```

continues

LISTING 28.6 Continued

```
  with DM.Table1 do begin
    for i := 0 to FieldCount - 1 do
      CBField.Items.Add(Fields[i].FieldName);
  end;
end;

procedure TFilterForm.LocateBtnClick(Sender: TObject);
var
  LO: TLocateOptions;
begin
  LO := [];
  if not CBCaseSens.Checked then Include(LO, loCaseInsensitive);
  if RBClosest.Checked then Include(LO, loPartialKey);
  if not DM.Table1.Locate(CBField.Text, EValue.Text, LO) then
    MessageDlg('Unable to locate match', mtInformation, [mbOk], 0);
end;

procedure TFilterForm.SpeedButton1Click(Sender: TObject);
begin
  DM.Table1.FindFirst;
end;

procedure TFilterForm.SpeedButton2Click(Sender: TObject);
begin
  DM.Table1.FindNext;
end;

procedure TFilterForm.SpeedButton3Click(Sender: TObject);
begin
  DM.Table1.FindPrior;
end;

procedure TFilterForm.SpeedButton4Click(Sender: TObject);
begin
  DM.Table1.FindLast;
end;

end.
```

TQuery and TStoredProc: The Other Datasets

Although these components aren't discussed in detail until the next chapter, this section is intended to introduce you to the `TQuery` and `TStoredProc` components as `TDataSet` descendants and siblings of `TTable`.

TQuery

The `TQuery` component enables you to use SQL to obtain specific datasets from one or more tables. Delphi enables you to use the `TQuery` component with both file-oriented server data (that is, Paradox and dBASE) and SQL server data. After assigning the `DatabaseName` property of `TQuery` to an alias or directory, you can enter into the `SQL` property the lines of SQL code you want to execute against the given database. For example, if `Query1` were hooked to the `DBDEMOS` alias, the following code would retrieve all records in the `BIOLIFE` table where the `Length (cm)` field is greater than `100`:

```
select * from BIOLIFE where BIOLIFE."Length (cm)" > 100
```

Like other datasets, the query will execute when its `Active` property is set to `True` or when its `Open()` method is called. If you want to perform a query that doesn't return a result set (an `insert into` query, for example), you should use `ExecSQL()` rather than `Open()` to invoke the query.

Another important property of `TQuery` is `RequestLive`. The `RequestLive` property indicates whether the result set returned is editable. Set this property to `True` when you want to edit the data returned by a query.

NOTE

Simply setting the `RequestLive` property doesn't guarantee a live result set. Depending on the structure of your query, the BDE may not be able to obtain a live result set. For example, queries containing a `HAVING` clause, using the `TO_DATE` function, or containing *abstract data type* (ADT) fields are not editable (see the BDE documentation for a complete list of restrictions). To determine whether a query is live, check the value of the `CanModify` property after opening the query.

In the next chapter, you'll learn more about `TQuery` features such as parameterized queries and SQL optimization.

TStoredProc

The `TStoredProc` component provides you with a means to execute stored procedures on a SQL server. Because this is a server-specific feature—and certainly not for database beginners—we'll save the explanation of this component for the next chapter.

Text File Tables

Delphi provides limited support for using text file tables in your applications. Text tables must consist of two files: a data file, which must end in a `.TXT` extension, and a schema file, which must end in an `.SCH` extension. Each file must have the same name (that is, `FOO.TXT` and `FOO.SCH`). The data file can be of fixed length or delimited. The schema file tells the BDE how to interpret the data file by providing information such as field names, sizes, and types.

The Schema File

The format of a schema file is similar to that of a Windows INI file. The section name is the same as that of the table (minus the extension). Table 28.5 shows the items and possible item values for a schema file.

TABLE 28.5 Schema File Items and Values

Item	*Possible Values*	*Meaning*
`FILETYPE`	`VARYING`	Each field in the file can occupy a variable amount of space. Fields are separated with a special character, and strings are delimited with a special character.
	`FIXED`	Each field can be found at a specific offset from the beginning of the line.
`CHARSET`	*(many)*	Specifies which language driver to use. Most commonly, this will be set to `ASCII`.
`DELIMITER`	*(any char)*	Specifies which character is to be used as a delimiter for `CHAR` fields. Used for `VARYING` tables only.
`SEPARATOR`	*(any char)*	Specifies which character is to be used as a field separator. Used for `VARYING` tables only.

Using the information shown in Table 28.5, the schema file must have an entry for each field in the table. Each entry will be in the following form:

```
FieldX = Field Name, Field Type, Size, Decimal Places, Offset
```

The syntax in the preceding example is explained in the following list:

- *`X`* represents the field number, from 1 to the total number of fields.
- *`Field Name`* can be any string identifier. Do not use quotes or string delimiters.
- *`Field Type`* can be any one of the following values:

Type	*Meaning*
`CHAR`	A character or string field
`BOOL`	A Boolean (`T` or `F`)
`DATE`	A date in the format specified in the BDE Config Tool
`FLOAT`	A 64-bit floating-point number
`LONGINT`	A 32-bit integer

Type	*Meaning*
`NUMBER`	A 16-bit integer
`TIME`	A time in the format specified in the BDE Config Tool
`TIMESTAMP`	A date and time in the format specified in the BDE Config Tool

- *`Size`* refers to the total number of characters or units. This value must be less than or equal to 20 for numeric fields.
- *`Decimal Places`* only has meaning for `FLOAT` fields. It specifies the number of digits after the decimal.
- *`Offset`* is used only for `FIXED` tables. It specifies the character position where a particular field begins.

Now, here's a sample schema file for a fixed table called `OPTeam`:

```
[OPTEAM]
FILETYPE = FIXED
CHARSET = ascii
Field1 = EmpNo,LONGINT,04,00,00
Field2 = Name,CHAR,16,00,05
Field3 = OfficeNo,CHAR,05,00,21
Field4 = PhoneExt,LONGINT,04,00,27
Field5 = Height,FLOAT,05,02,32
```

Here's a schema file for a `VARYING` version of a similar table called `OPTeam2`:

```
[OPTEAM2]
FILETYPE = VARYING
CHARSET = ascii
DELIMITER = "
SEPARATOR = ,
Field1 = EmpNo,LONGINT,04,00,00
Field2 = Name,CHAR,16,00,00
Field3 = OfficeNo,CHAR,05,00,00
Field4 = PhoneExt,LONGINT,04,00,00
Field5 = Height,FLOAT,05,02,00
```

CAUTION

The BDE is very picky about the format of a schema file. If you have one misplaced character or misspelled word, the BDE may not be able to recognize your data at all. If you're having problems getting at your data, scrutinize your schema file.

The Data File

The data file should be a fixed-length (`FIXED`) or delimited (`VARYING`) file that contains one record per line. A sample data file for `OPTeam` can be shown as this:

```
2093 Steve Teixeira  C2121 1234 6.5
3265 Xavier Pacheco  C0001 3456 5.6
2610 Lino Tadros     E2126 5678 5.11
2900 Lance Bullock   C2221 9012 6.5
0007 Greg de Vries   F3169 7890 5.10
1001 Tillman Dickson C3456 0987 5.9
2611 Rory Bannon     E2127 6543 6.0
6908 Karl Santos     A1098 5893 5.6
0909 Mr. T           B0087 1234 5.9
```

A similar data file for `OPTeam2` would look like this:

```
2093,"Steve Teixeira","C2121",1234,6.5
3265,"Xavier Pacheco","C0001",3456,5.6
2610,"Lino Tadros","E2126",5678,5.11
2900,"Lance Bullock","C2221",9012,6.5
0007,"Greg de Vries","F3169",7890,5.10
1001,"Tillman Dickson","C3456",0987,5.9
2611,"Rory Bannon","E2127",6543,6.0
6908,"Karl Santos","A1098",5893,5.6
0909,"Mr. T","B0087",1234,5.9
```

Using the Text Table

You can use text tables with `TTable` components much like any other database type. Set the table's `DatabaseName` property to the alias or directory containing the TXT and SCH files. Set the `TableType` property to `ttASCII`. Now you should be able to view all available text tables by clicking the drop-down button on the `TableName` property. Select one of the tables into the property, and you'll be able to view the fields by hooking up a `TDataSource` and a `TDBGrid`. Figure 28.21 shows a form browsing the `OPTeam` table.

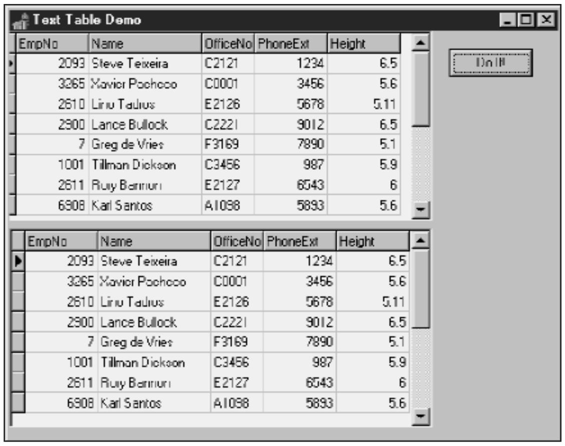
NOTE

If all the fields in your text table appear to be cramped into one field, the BDE is having problems reading your schema file.

Limitations

Borland never intended for text files to be used in lieu of proper database formats. Because of the limitations inherent in text files, we (the authors) seriously advise against using text file tables for anything other than importing data to and exporting data from real database formats. Here's a list of limitations to keep in mind when working with text tables:

- Indexes are not supported, so you can't use any `TTable` method that requires an index.
- You cannot use a `TQuery` component with a text table.
- Deleting records is not supported.
- Inserting records is not supported. Attempts to insert a record will cause the new record to be appended to the end of the table.
- Referential integrity is not supported.
- BLOB data types are not supported.
- Editing is not supported on `VARYING` tables.
- Text tables are always opened with exclusive access. You should, therefore, open your text tables in code rather than during design time.



EmpNo	Name	OfficeNo	PhoneExt	Height
2093	Steve Teixeira	C2121	1234	6.5
3265	Xavier Pacheco	C0001	3456	5.6
2610	Lino Tadros	E2126	5678	5.11
2900	Lance Bullock	C2221	9012	6.5
7	Greg de Vries	F3169	7890	5.1
1001	Tillman Dickson	C3456	987	5.9
2611	Ruy Barruri	E2127	6543	6
6908	Karl Santos	A1098	5893	5.6

EmpNo	Name	OfficeNo	PhoneExt	Height
2093	Steve Teixeira	C2121	1234	6.5
3265	Xavier Pacheco	C0001	3456	5.6
2610	Lino Tadros	E2126	5678	5.11
2900	Lance Bullock	C2221	9012	6.5
7	Greg de Vries	F3169	7890	5.1
1001	Tillman Dickson	C3456	987	5.9
2611	Ruy Barruri	E2127	6543	6
6908	Karl Santos	A1098	5893	5.6

FIGURE 28.21
Browsing a text table.

Text Table Import

As mentioned earlier, about the only reasonable use for text tables is in converting them to a real database format. With that in mind, what follows is a set of step-by-step instructions for using a `TBatchMove` component to copy a table from text format to a Paradox table. Assume a form containing two `TTable` objects and one `TBatchMove` component. The `TTable` object that represents the text table is called `TextTbl`, and the `TTable` object that represents the target Paradox table is called `PDoxTbl`. The `TBatchMove` component is called `BM`. Here are the steps:

1. Hook TextTbl to the text table you want to import (as described earlier).
2. Set the DatabaseName property of PDoxTbl to the target alias or directory. Set the TableName property to the desired table name. Set the TableType property to ttParadox.
3. Set the Source property of BM to TextTbl. Set the Destination property to PDoxTbl. Set the Mode property to batCopy.
4. Right click BM and select Execute from the local menu.
5. Voilà! You have just copied your text table to a Paradox table.

Connecting with ODBC

It's a given that the BDE can only provide native support for a limited subset of databases in the world. What happens, then, when your situation requires that you connect to a database type—such as Btrieve, for example—that's not directly supported by the BDE? Can you still use Delphi? Of course. The BDE provides an ODBC socket so that you can use an *Open Database Connectivity* (ODBC) driver to access databases not directly supported by the BDE; the capability to take advantage of this feature is built into the Professional and Client/Server Suite editions of Delphi. ODBC is a standard developed by Microsoft for product-independent database driver support.

Where to Find an ODBC Driver

The best place to obtain an ODBC driver is through the vendor who distributes the database format you want to access. When you do venture out to obtain an ODBC driver, bear in mind that there's a difference between 16- and 32-bit ODBC drivers, and that Delphi requires the 32-bit drivers. In addition to the vendor of your particular database, there are a number of vendors who produce ODBC drivers for many different types of databases. In particular, you can obtain ODBC drivers for Access, Excel, SQL Server, and FoxPro from Microsoft. These drivers are available either in the ODBC Driver Pack, or you can often find them on the MS Developer Network CD-ROMs.

CAUTION

Not all ODBC drivers are created equal! Many ODBC drivers are "brain deadened" to work with only one particular software package or to have their functionality otherwise limited. Examples of these types of drivers are ones that have shipped with past versions of Microsoft Office products (which are intended to work only with MS Office). Make sure that the ODBC driver you purchase is certified for application development, not just to work with some existing package.

An ODBC Example: Connecting to MS Access

Assuming you've obtained the necessary 32-bit ODBC driver from Microsoft or another vendor, this section takes you step by step from configuring the driver to making it work with a Delphi `TTable` object. Although Access is directly supported by the BDE, that's beside the point—this section is intended to serve as an example of using the BDE's ODBC socket. This demonstration assumes that you do not yet have an Access database on your hard disk, and it takes you through the steps for creating one:

1. Install the driver using the vendor-provided disk. Once it's installed, run the Windows Control Panel, and you should see an icon for ODBC Data Sources (32bit), as shown in Figure 28.22. Double-click the icon and you'll be presented with the ODBC Data Source Administrator dialog, as shown in Figure 28.23.

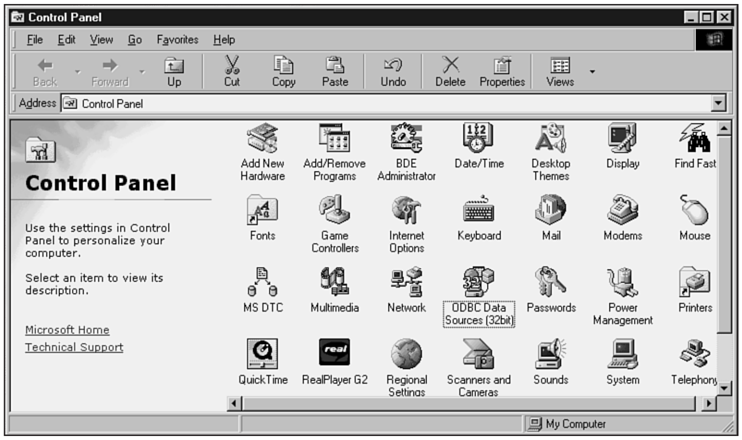
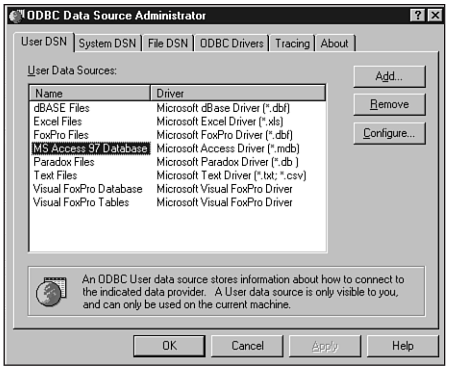


FIGURE 28.22
The Windows Control Panel containing the ODBC Data Sources (32 bit) icon.

2. Click the Add button in the Data Source Administrator dialog, and you'll be presented with the Create New Data Source dialog, as shown in Figure 28.24. From this dialog, select "Microsoft Access Driver (*.mdb)" and click Finish.
3. You'll now be presented with a dialog similar to the ODBC Microsoft Access Setup dialog shown in Figure 28.25. You can give the data source any name and description you choose. In this case, we'll call it `AccessDB`, and the description will read `DDG Test for Access`.
4. Click the Create button in the ODBC Microsoft Access Setup dialog, and you'll be presented with the New Database dialog, where you can choose a name for your new database and a directory in which to store the database file. Click the OK button after you

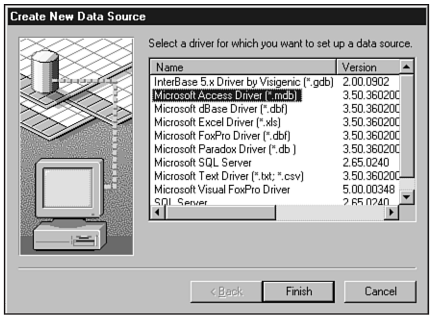
choose a file and path. Figure 28.25 shows a picture of the ODBC Microsoft Access Setup dialog with steps 3 and 4 completed. Click OK to dismiss this dialog, and then click Close to dismiss the Data Sources dialog. The data source is now configured, and you're ready to create a BDE alias that maps to this data source.



Name	Driver
dBASE Files	Microsoft dBase Driver (*.dbf)
Excel Files	Microsoft Excel Driver (*.xls)
FoxPro Files	Microsoft FoxPro Driver (*.dbf)
MS Access 97 Database	Microsoft Access Driver (*.mdb)
Paradox Files	Microsoft Paradox Driver (*.db)
Text Files	Microsoft Text Driver (*.txt; *.csv)
Visual FoxPro Database	Microsoft Visual FoxPro Driver
Visual FoxPro Tables	Microsoft Visual FoxPro Driver

An ODBC User data source stores information about how to connect to the indicated data provider. A User data source is only visible to you, and can only be used on the current machine.

FIGURE 28.23
The ODBC Data Source Administrator dialog.



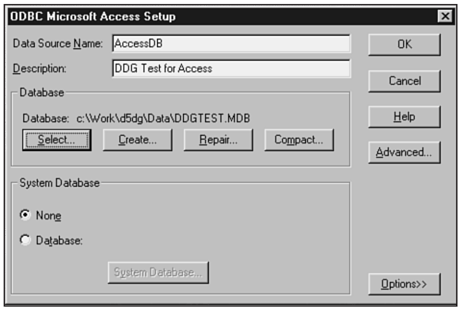
Select a driver for which you want to set up a data source.

Name	Version
InterBase 5.x Driver by Visigenic (*.gdb)	2.00.0902
Microsoft Access Driver (*.mdb)	3.50.36020
Microsoft dBase Driver (*.dbf)	3.50.36020
Microsoft Excel Driver (*.xls)	3.50.36020
Microsoft FoxPro Driver (*.dbf)	3.50.36020
Microsoft Paradox Driver (*.db)	3.50.36020
Microsoft SQL Server	2.65.0240
Microsoft Text Driver (*.txt; *.csv)	3.50.36020
Microsoft Visual FoxPro Driver	5.00.00348
SQL Server	2.65.0240

FIGURE 28.24
The Create New Data Source dialog.

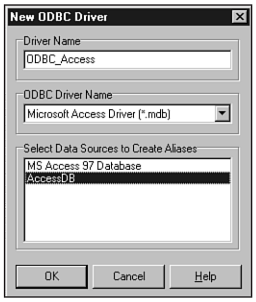
5. Close all applications that use the BDE. Run the BDE Administrator tool that comes with Delphi and change to the Configuration page on the left pane. Expand the Drivers branch of the tree view, right-click ODBC, and select New from the local menu. This will invoke the New ODBC Driver dialog. Driver Name can be anything you like. For the sake of this example, we'll use `ODBC_Access`. ODBC Driver Name will be "Microsoft Access Driver (*.mdb)" (the same driver name as step 2). Default Data Source Name should come up automatically as `AccessDB` (the same name as step 3). The completed

dialog is shown in Figure 28.26. Select OK, and you'll return to the BDE Administrator main window.



Data Source Name: AccessDB
Description: DDG Test for Access
Database
Database: c:\Work\d5dg\Data\DDGTEST.MDB
System Database
(● None) ( ) Database:

FIGURE 28.25
The ODBC Microsoft Access Setup dialog.

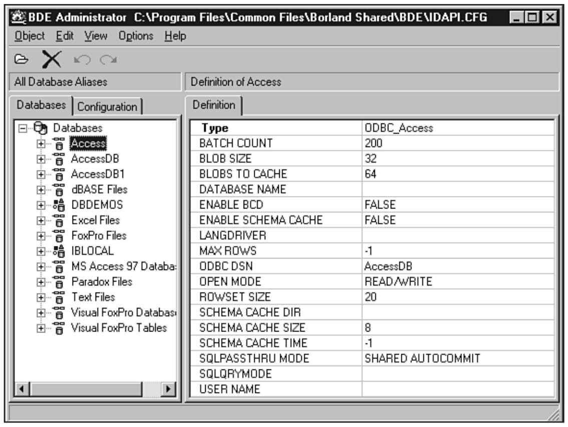


Driver Name: ODBC_Access
ODBC Driver Name: Microsoft Access Driver (*.mdb)
Select Data Sources to Create Aliases:
MS Access 97 Database
AccessDB

FIGURE 28.26
The completed New ODBC Driver dialog.

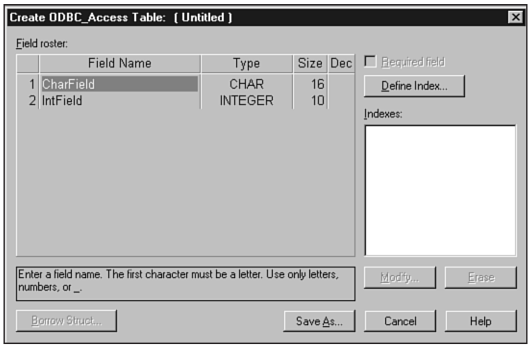
6. Change to the Databases page in the left pane of the BDE Administrator and select Object, New from the main menu. This will invoke the New Database Alias dialog. In this dialog, select `ODBC_Access` (from step 5) as the database driver name and click OK. You can then give the alias any name you like—we'll use `Access` in this case. The completed alias is shown in Figure 28.27. Select OK to dismiss the dialog and then select Object, Apply from the BDE Administrator main window. The alias has now been created, and you may now close the BDE Administrator tool. The next step is to create a table for the database.
7. You'll use the Database Desktop application that ships with Delphi to create tables for your Access database. Select File, New, Table from the main menu, and you'll be presented with the Create Table dialog. Choose `ODBC_Access` (same as steps 5 and 6) as the table type, and the Create ODBC_Access Table dialog will come up.

8. Assuming you're familiar with creating tables in Database Desktop (if you're not, refer to the Delphi documentation), the Create ODBC_Access Table dialog works the same as the "create table" dialogs for other database types. For demonstration purposes, add one field of type `CHAR` and one of type `INTEGER`. Figure 28.28 shows the completed dialog.



Type	ODBC_Access
BATCH COUNT	200
BLOB SIZE	32
BLOBS TO CACHE	64
DATABASE NAME	
ENABLE BCD	FALSE
ENABLE SCHEMA CACHE	FALSE
LANGDRIVER	
MAX ROWS	-1
ODBC DSN	AccessDB
OPEN MODE	READ/WRITE
ROWSET SIZE	20
SCHEMA CACHE DIR	
SCHEMA CACHE SIZE	8
SCHEMA CACHE TIME	-1
SQLPASSTHRU MODE	SHARED AUTOCOMMIT
SQLQRYMODE	
USER NAME	

FIGURE 28.27
The new Access alias in BDE Administrator.



	Field Name	Type	Size	Dec
1	CharField	CHAR	16	
2	IntField	INTEGER	10	

Enter a field name. The first character must be a letter. Use only letters, numbers, or _.

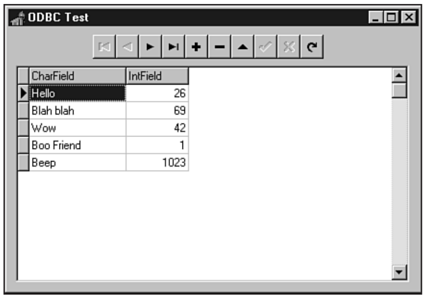
FIGURE 28.28
The completed Create ODBC_Access Table dialog.

9. Click the Save As button, and you'll be prompted with the Save Table As dialog. In this dialog, first set Alias to `Access` (from step 6). At this point, you'll be presented with a database login dialog—just click OK to dismiss the dialog, because no user name or password have been specified. Now give the table a name (do not use an extension) in the File Name edit control. We'll use `TestTable` in this case. Click OK, and the table will be stored to the database. You're now ready to access this database with Delphi.

NOTE

MS Access tables that comprise a database are stored in one MDB file. Although this is in contrast to Paradox and dBASE, which store each table as a separate file, it is similar to SQL server databases.

10. Create a new project in Delphi. The main form should contain one each of the `TTable`, `TDataSource`, and `TDBGrid` components. `DBGrid1` hooks to `Table1` via `DataSource1`. Select Access (from steps 6 and 9) into `Table1`'s `DatabaseName` property. Click `Table1`'s `TableName` property, and you'll be presented with a login dialog. Simply click the OK button (no password has been configured) and you can choose an available table from the Access database. Because `TestTable` is the only table you created, choose that table. Now set `Table1`'s `Active` property to `True`, and you'll see the field names appear in `DBGrid1`. Run the application, and you'll be able to edit the table. Figure 28.29 shows the completed application.



CharField	IntField
Hello	26
Blah blah	69
Wow	42
Boo Friend	1
Beep	1023

FIGURE 28.29
Browsing an ODBC table in Delphi.

ActiveX Data Objects (ADO)

One of the marquee new features added to Delphi 5 is the ability to access data directly through Microsoft's ADO. This is accomplished via a suite of new components in Delphi Enterprise collectively known as *ADOExpress* and found on the ADO page of the Component Palette. By leveraging the abstract `TDataSet` class mentioned earlier in this chapter, the ADOExpress components are able to provide ADO connectivity directly, without having to go through the BDE. This means simplified deployment, fewer dependencies on code you don't have control over, and improved performance.

The Who's Who of Microsoft Data Access

Microsoft has backed a number of data-access strategies over the years, so don't feel bad if the letters *A*, *D*, *O* tend to fall illegibly into an alphabet soup of other acronyms, such as ODBC, DAO, RDS, and UDA. To help put things into perspective, it's worth taking the time to review this collection of terms and acronyms that deal with the various Microsoft data-access strategies. In doing so, you'll hopefully gain a little perspective on how ADO fits into the picture.

- *UDA* (Universal Data Access) is the umbrella term Microsoft gives to its whole data access strategy, including ADO, OLE DB, and ODBC. It's interesting to note that UDA doesn't refer strictly to databases but can be applicable to other data-store technologies, such as directory services, Excel spreadsheet data, and Exchange server data.
- *ODBC* (Open Database Connectivity) is the most well-established Microsoft data-connectivity technology. The ODBC architecture involves a generic SQL-based API, upon which drivers can be developed to access specific databases. Because of the large market presence and proven track record of ODBC, you can still find ODBC drivers for nearly any database. Because of this, ODBC will continue to be used extensively for some time to come, even if it is a bit long in the tooth.
- *RDO* (Remote Data Objects) provides a COM wrapper for ODBC. The goal of RDO is to simplify ODBC development and open ODBC development to Visual Basic and VBA programmers.
- *Jet* is the name of the database engine built into Microsoft Access. Jet supports both Access's native MDB databases and ODBC.
- *DAO* (Data Access Objects) is yet another COM-based API for data access. DAO provides encapsulations for both Jet and ODBC.
- *ODBCDirect* is the technology Microsoft added later to DAO to provide direct access to ODBC, rather than supporting ODBC through Jet.
- *OLE DB* is a generic and simplified COM-based specification and API for data access. OLE DB was designed to be independent of any particular database back end and is the underlying architecture for Microsoft's latest data-connectivity solutions. Drivers, known as *OLE DB providers*, can be written to connect to virtually any data store through OLE DB.
- *ADO* (ActiveX Data Objects) provides a more developer-friendly wrapper for OLE DB.
- *RDS* (Remote Data Services) is an ADO-based technology that enables remote access of ADO data sources in order to build multitier systems. RDS was formerly known as *ADC* (Advanced Data Connector).
- *MDAC* (Microsoft Data Access Components) is the practical implementation and file distribution for UDA. MDAC includes four distinct technologies: ODBC, OLE DB, ADO, and RDS.

ADOExpress Components

Six components make up ADOExpress. Here, we categorize them into three groups: connectivity, ADO access, and compatibility.

Connectivity Components

The `TADOConnection` component is used to establish a connection with an ADO data store. You can hook multiple ADO dataset and command components to a single `TADOConnection` component in order to share the connection for the purposes of executing commands, retrieving data, and operating on metadata. This component is similar to the `TDataBase` component for BDE-based applications, and it's not necessary for simple applications.

The `TRDSConnection` component encapsulates a remote RDS connection by exposing the functionality of RDS's `DataSpace` object. `TRDSConnection` is used by specifying the name of the RDS server machine in the `ComputerName` parameter and the ProgID of the RDS server in the `ServerName` property.

ADO Access Components

`TADODataSet` and `TADOCommand` make up the group of ADO access components. This group gets its name because the components provide their data-manipulation capability using more of an ADO style than the traditional BDE style with which Delphi developers are generally more familiar.

The `TADODataSet` component is the primary component used to retrieve and operate on ADO data. This component has the ability to manipulate tables and execute SQL queries and stored procedures and can connect directly to a data store or connection through a `TADOConnection` component. In VCL terms, `TADODataSet` encapsulates the functionality that the `TTable`, `TQuery`, and `TStoredProc` components provide for BDE-based applications.

The `TADOCommand` component is used to execute SQL statements that do not return result sets, much like `TQuery.Execute()` and `TStoredProc.ExecProc()` in BDE-based applications. Like `TADODataSet`, this component can connect directly to a data store or connect through a `TADOConnection`. `TADOCommand` can also be used to execute SQL that returns a result set, but the result set must be manipulated using a `TADODataSet` component. The following line of code shows how to pipe the result set of a `TADOCommand` query into a `TADODataSet`:

```
ADODataSet.RecordSet := ADOCommand.Execute;
```

Compatibility Components

We consider `TADOTable`, `TADOQuery`, `TADOStoredProc` to be compatibility components because they provide developers with the separate table, query, and stored procedure components that they may already be familiar with. Developers are free to use these or the ADO access components described previously, although using these components may make it a bit easier to port

BDE-based applications to ADO. Like `TADODataSet` and `TADOCommand`, the compatibility components have the ability to connect directly to a data store or connect through a `TADOConnection` component.

As you might have guessed, `TADOTable` is used to retrieve and operate on a dataset produced by a single table. `TADOQuery` can be used to retrieve and operate on a dataset produced by a SQL statement or execute *Data Definition Language* (DDL) SQL statements, such as `CREATE TABLE`. `TADOStoredProc` is used to execute stored procedures, whether or not they return result sets.

Connecting to an ADO Data Store

The `TADOConnection` component and each of the ADO access and compatibility components contain a property called `ConnectionString` that specifies the connection to an ADO data store and its attributes. The simplest way to provide a value for this property is by using the property editor, which you can invoke by clicking the ellipses next to the property value in the Object Inspector. You'll then be presented with a property editor dialog like the one shown in Figure 28.30.

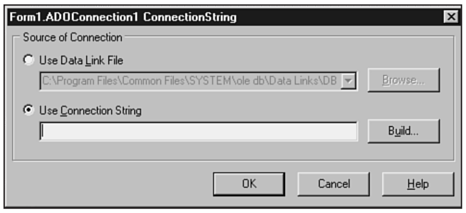
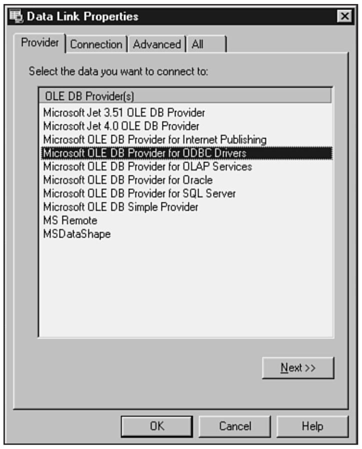

Form1.ADOConnection1 ConnectionString
Source of Connection
Use Data Link File
C:\Program Files\Common Files\SYSTEM\ole db\Data Links\DB
Browse...
Use Connection String
Build...
OK
Cancel
Help

FIGURE 28.30
The `ConnectString` *property editor.*

In this dialog, you have the option of choosing a data link file or a connection string for the property value. A data link file is a file on disk, typically with a `.UDL` extension, in which a connection string is stored. Assuming you want to build a new connection string rather than use a UDL file, you should select the Use Connection String radio button and click the Build button. This will invoke the Data Link Properties window shown in Figure 28.31.

Building UDL Files

If you want to build UDL files in order to create connection strings that can be reused many times, you can do so fairly easily in the Windows Explorer, as long as MDAC has been installed on your machine (Delphi 5 installs MDAC). Just open an Explorer window to the folder in which to want to create a new UDL file and then right-click. Then select New, Microsoft Data Link from the local menu. This will create a new UDL file, which you can name. Then right-click the icon for the UDL file and select Properties from the local menu. You'll then be presented with the Data Link Properties window as described in this section.



Select the data you want to connect to:

OLE DB Provider(s)
Microsoft Jet 3.51 OLE DB Provider
Microsoft Jet 4.0 OLE DB Provider
Microsoft OLE DB Provider for Internet Publishing
Microsoft OLE DB Provider for ODBC Drivers
Microsoft OLE DB Provider for OLAP Services
Microsoft OLE DB Provider for Oracle
Microsoft OLE DB Provider for SQL Server
Microsoft OLE DB Simple Provider
MS Remote
MSDataShape

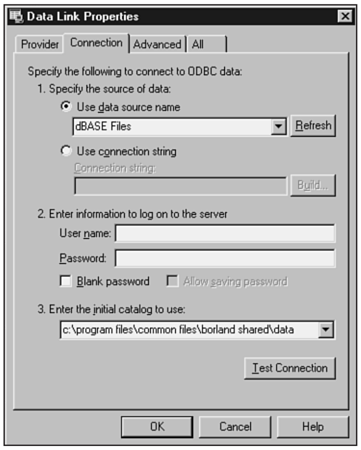
FIGURE 28.31
The Provider page of the Data Link Properties window.

The first page, Provider, of this dialog enables you to choose the OLE DB provider to which you want to connect. For example, you may choose the Microsoft OLE DB provider for ODBC drivers in order to connect to an ODBC driver via OLE DB.

After selecting the provider, you can click the Next button or the Connection tab in order to be taken to the Connection page shown in Figure 28.32. On this page, you'll configure the driver to connect to a particular database. For this example, we want to connect to a dBASE table, so select the dBASE ODBC data source from the Use Data Source Name drop-down list in part 1 of the page. You can skip part 2 of the page because the dBASE table is not password protected. In part 3 of the dialog, we need to set the initial catalog name to the directory containing the dBASE tables. For testing purposes, we set it to the directory containing the Borland sample data.

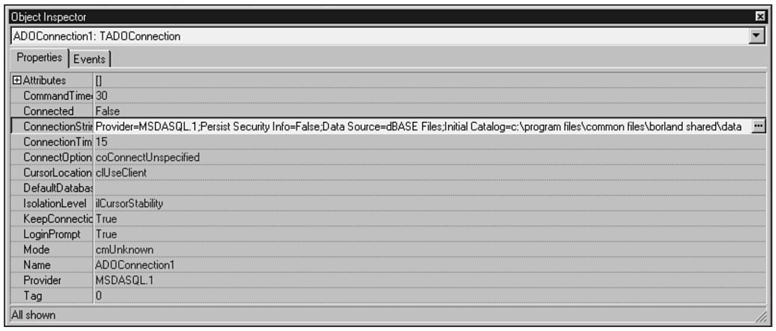
To ensure that the connection is valid, click the Test Connection button, and you'll receive a confirmation of a valid connection or an error if the directory you entered was invalid.

The Advanced and All pages of the Data Link Properties window enable you to set various properties on the connection, such as Connect Timeout, Access Permissions, Locale ID, and so on. For our purposes, we don't need to edit these pages and can use the defaults. Clicking OK in this window and then again in the property editor dialog will cause the connection string to be created and placed in the Object Inspector, as shown in Figure 28.33.



Specify the following to connect to ODBC data:
1. Specify the source of data:
(●) Use data source name: dBASE Files
( ) Use connection string
Connection string:
2. Enter information to log on to the server
User name:
Password:
☐ Blank password ☐ Allow saving password
3. Enter the initial catalog to use: c:\program files\common files\borland shared\data

FIGURE 28.32
The Connection page of the Data Link Properties window.



ADOConnection1: TADOConnection

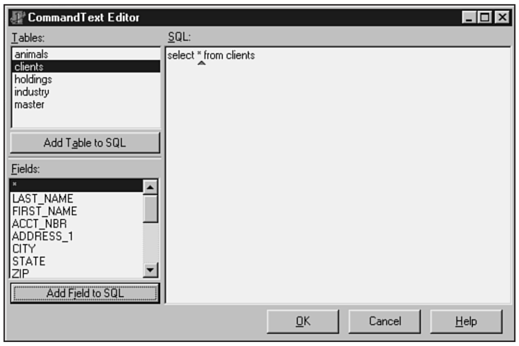
⊞Attributes	[]
CommandTime	30
Connected	False
ConnectionStri	Provider=MSDASQL.1;Persist Security Info=False;Data Source=dBASE Files;Initial Catalog=c:\program files\common files\borland shared\data
ConnectionTim	15
ConnectOption	coConnectUnspecified
CursorLocation	clUseClient
DefaultDatabas	
IsolationLevel	ilCursorStability
KeepConnectic	True
LoginPrompt	True
Mode	cmUnknown
Name	ADOConnection1
Provider	MSDASQL.1
Tag	0

FIGURE 28.33
The completed `ConnectString` *property in the Object Inspector.*

Example: Connecting via ADO

Now that you know how to create a new connection string, you already know the hardest part about accessing data via ADO. To take it to the next step in Delphi, you can view the data in the connection you just created. To accomplish this, we'll use only a `TADODataSet` component. Follow the steps outlined previously for setting the `ConnectString` property of the

TADODataSet. Then use the property editor for the CommandText property to create a SQL statement that enables you to view the contents of a table, such as that shown in Figure 28.34. Then click OK to close the dialog.



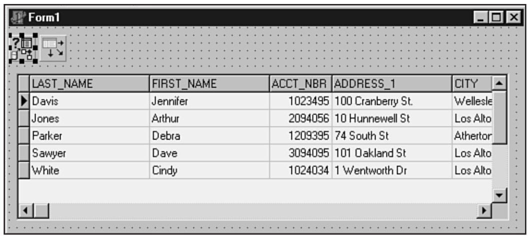
animals
clients
holdings
industry
master

```
select * from clients
```

*
LAST_NAME
FIRST_NAME
ACCT_NBR
ADDRESS_1
CITY
STATE
ZIP

FIGURE 28.34
Editing the CommandText *property.*

Once you've set the CommandText property, you can set the Active property of the TADODataSet to True. The component is now actively viewing the data. In order to see it, you can drop down a TDataSource component, which you'll connect to the TADODataSet, and a TDBGrid component, which you'll connect to the TDataSource, as you learned earlier in this chapter. The result is shown in Figure 28.35.



LAST_NAME	FIRST_NAME	ACCT_NBR	ADDRESS_1	CITY
Davis	Jennifer	1023495	100 Cranberry St.	Wellesle
Jones	Arthur	2094056	10 Hunnewell St	Los Alto
Parker	Debra	1209395	74 South St	Atherton
Sawyer	Dave	3094095	101 Oakland St	Los Alto
White	Cindy	1024034	1 Wentworth Dr	Los Alto

FIGURE 28.35
Accessing data using the TADODataSet *component.*

ADO Deployment

In order to deploy ADO-based solutions on Windows 95, 98, and NT, remember that the MDAC must be installed on the target systems. You'll find the redistributable files in the \MDAC directory of the Delphi 5 CD-ROM. Windows 2000 includes MDAC, so redistribution of MDAC isn't necessary if your application will be running on a Windows 2000 machine.

Summary

After reading this chapter, you should be ready for just about any type of non-SQL database programming with Delphi. You learned the ins and outs of Delphi's `TDataSet` component, which is the ancestor of the `TTable`, `TQuery`, and `TStoredProc` components. You also learned techniques for manipulating `TTable` objects, how to manage fields, and how to work with text tables. Along with all this how-to information, you also learned about the various data-access strategies, including BDE, ODBC, and ADO. As you've seen, VCL offers a pretty tight object-oriented wrapper around the procedural BDE in addition to an extensible framework than can accommodate other engines, such as ADO.

The next chapter, "Developing Client/Server Applications," focuses a bit more on client/server technology and using related VCL classes such as `TQuery` and `TStoredProc` components.