CHAPTER

30

Extending Database VCL

IN THIS CHAPTER

Out of the box, Visual Component Library's (VCL's) database architecture is equipped to communicate primarily by means of the Borland Database Engine (BDE)—feature-rich and reliable database middleware. What's more, VCL serves as a kind of insulator between you and your databases, allowing you to access different types of databases in much the same manner. Although all this adds up to reliability, scalability, and ease of use, there is a downside: database-specific features provided both within and outside the BDE are generally not provided for in the VCL database framework. This chapter provides you with the insight you'll need to extend VCL by communicating directly with the BDE and other data sources to obtain database functionality not otherwise available in Delphi.

Using the BDE

When you're writing applications that make direct calls to the BDE, there are a few rules of thumb to keep in mind. This section presents the general information you need to get into the BDE API from your Delphi applications.

The BDE Unit

All BDE functions, types, and constants are defined in the `BDE` unit. This unit will need to be in the `uses` clause of any unit from which you want to make BDE calls. Additionally, the interface portion of the `BDE` unit is available in the `BDE.INT` file, which you'll find in the `..\Delphi5\Doc` directory. You can use this file as a reference to the functions and records available to you.

TIP

For additional assistance on programming using the BDE API, take a look at the `BDE32.hlp` help file provided in your `BDE` directory (the default path for this directory is `\Program Files\Borland\Common Files\BDE`). This file contains detailed information on all BDE API functions and very good examples in both Object Pascal and C.

Check()

All BDE functions return a value of type `DBIRESULT`, which indicates the success or failure of the function call. Rather than going through the cumbersome process of checking the result of every BDE function call, Delphi defines a procedure called `Check()`, which accepts a `DBIRESULT` as a parameter. This procedure will raise an exception when the `DBIRESULT` indicates any value except success. The following code shows how to, and how not to, make a BDE function call:

```
// !!Don't do this:
var
```

```
  Rez: DBIRESULT;
  A: array[0..dbiMaxUserNameLen] of char;
begin
  Rez := dbiGetNetUserName(A);  // make BDE call
  if Rez <> DBIERR_NONE then    // handle error
    // handle error here
  else begin
    // continue with function
  end;
end;

// !!Do do this:
var
  A: array[0..dbiMaxUserNameLen] of char;
begin
  { Handle error and make BDE call at one time. }
  { Exception will be raised in case of error.  }
  Check(dbiGetNetUserName(A));
  // continue with function
end;
```

Cursors and Handles

Many BDE functions accept as parameters handles to cursors or databases. Roughly speaking, a *cursor handle* is a BDE object that represents a particular set of data positioned at some particular row in that data. The data type of a cursor handle is `hDBICur`. Delphi surfaces this concept as the current record in a particular table, query, or stored procedure. The `Handle` properties of `TTable`, `TQuery`, and `TStoredProc` hold this cursor handle. Remember to pass the `Handle` of one of these objects to any BDE function that requires an `hDBICur`.

Some BDE functions also require a handle to a database. A BDE database handle is of type `hDBIDb`, and it represents some particular open database—either a local or networked directory in the case of dBASE or Paradox, or a server database file in the case of a SQL server database. You can obtain this handle from a `TDatabase` through its `Handle` property. If you're not connecting to a database using a `TDatabase` object, the `DBHandle` properties of `TTable`, `TQuery`, and `TStoredProc` also contain this handle.

Synching Cursors

It's been established that an open Delphi dataset has the concept of a current record, whereas the underlying BDE maintains the concept of a cursor that points to some particular record in a dataset. Because of the way Delphi performs record caching to optimize performance, sometimes the Delphi current record is not in sync with the underlying BDE cursor. Normally, this is not a problem because this behavior is business as usual for VCL's database framework.

However, if you want to make a direct call to a BDE function that expects a cursor as a parameter, you need to ensure that Delphi's current cursor position is synchronized with the underlying BDE cursor. It might sound like a daunting task, but it's actually quite easy to do. Simply call the `UpdateCursorPos()` method of a `TDataSet` descendant to perform this synchronization.

In a similar vein, after making a BDE call that modifies the position of the underlying cursor, you need to inform VCL that it needs to resynchronize its own current record position with that of the BDE. To do this, you must call the `CursorPosChanged()` method of `TDataSet` descendants immediately after calling into the BDE. The following code demonstrates how to use these cursor-synchronization functions:

```
procedure DoSomethingWithTable(T: TTable);
begin
  T.UpdateCursorPos;
  // call BDE function(s) which modifies cursor position
  T.CursorPosChanged;
end;
```

dBASE Tables

dBASE tables have a number of useful capabilities that are not directly supported by Delphi. These features include, among other things, the maintenance of a unique physical record number for each record, the capability to "soft-delete" records (delete records without removing them from the table), the capability to undelete soft-deleted records, and the capability to pack a table to remove soft-deleted records. In this section, you'll learn about the BDE functions involved in performing these actions, and you'll create a `TTable` descendant called `TdBaseTable` that incorporates these features.

Physical Record Number

dBASE tables maintain a unique physical record number for each record in a table. This number represents a record's physical position relative to the beginning of the table (regardless of any index currently applied to the table). To obtain a physical record number, you must call the BDE's `DbiGetRecord()` function, which is defined as follows:

```
function DbiGetRecord(hCursor: hDBICur; eLock: DBILockType;
    pRecBuff: Pointer; precProps: pRECProps): DBIResult stdcall;
```

`hCursor` is the cursor handle. Usually, this is the `Handle` property of the `TDataSet` descendant.

`eLock` is an optional request for the type of lock to place on the record. This parameter is of type `DBILockType`, which is an enumerated type defined as follows:

```
type
  DBILockType = (
    dbiNOLOCK,             // No lock (Default)
    dbiWRITELOCK,          // Write lock
    dbiREADLOCK);          // Read lock
```

In this case you don't want to place a lock on the record because you're not intending to modify the record content; therefore, `dbiNOLOCK` is the appropriate choice.

`pRecBuff` is a pointer to a record buffer. Because you want to obtain only the record properties and not the data, you should pass `Nil` for this parameter.

`pRecProps` is a pointer to a `RECProps` record. This record is defined as follows:

```
type
  pRECProps = ^RECProps;
  RECProps = packed record       // Record properties
    iSeqNum         : Longint;   // When Seq# supported only
    iPhyRecNum      : Longint;   // When Phy Rec#s supported only
    iRecStatus      : Word;      // Delayed Updates Record Status
    bSeqNumChanged  : WordBool;  // Not used
    bDeleteFlag     : WordBool;  // When soft delete supported only
  end;
```

As you can see, you can obtain a variety of information from this record. In this case, you're concerned only with the `iPhyRecNum` field, which is valid only in the case of dBASE and FoxPro tables.

Putting this all together, the following code shows a method of `TdBaseTable` that returns the physical record number of the current record:

```
function TdBaseTable.GetRecNum: Longint;
{ Returns the physical record number of the current record. }
var
  RP: RECProps;
begin
  UpdateCursorPos;              // update BDE from Delphi
  { Get current record properties }
  Check(DbiGetRecord(Handle, dbiNOLOCK, Nil, @RP));
  Result := RP.iPhyRecNum;     // return value from properties
end;
```

Viewing Deleted Records

Viewing records that have been soft-deleted in a dBASE table is as easy as making one BDE API call. The function to call is `DbiSetProp()`, which is a very powerful function that enables you to modify the different properties of multiple types of BDE objects. For a complete description of this function and how it works, your best bet is to check out the "Properties—Getting and Setting" topic in the BDE help. This function is defined as follows:

```
function DbiSetProp(hObj: hDBIObj; iProp: Longint;
  iPropValue: Longint): DBIResult stdcall;
```

The `hObj` parameter holds a handle to some type of BDE object. In this case, it will be a cursor handle.

The `iProp` parameter will contain the identifier of the property to be set. You'll find a complete list of these in the aforementioned topic in the BDE help. For the purposes of enabling or disabling the view of deleted records, use the `curSOFTDELETEON` identifier.

`iPropValue` is the new value for the given property. In this case, it's a Boolean value (`0` meaning *off*; `1` meaning *on*).

The following code shows the `SetViewDeleted()` method of `TdBaseTable`:

```
procedure TdBaseTable.SetViewDeleted(Value: Boolean);
{ Allows the user to toggle between viewing and not viewing }
{ deleted records. }
begin
  { Table must be active }
  if Active and (FViewDeleted <> Value) then begin
    DisableControls;     // avoid flicker
    try
      { Magic BDE call to toggle view of soft deleted records }
      Check(DbiSetProp(hDBIObj(Handle), curSOFTDELETEON, Longint(Value)));
    finally
      Refresh;           // update Delphi
      EnableControls;    // flicker avoidance complete
    end;
    FViewDeleted := Value
  end;
end;
```

This method first performs a test to ensure that the table is open and that the value to be set is different than the value the `FViewDeleted` field in the object already contains. It then calls `DisableControls()` to avoid flicker of any data-aware controls attached to the table. The `DbiSetProp()` function is called next (notice the necessary typecast of `hDBICur`'s `Handle` parameter to an `hDBIObj`). Think of `hDBIObj` as an untyped handle to some type of BDE object. After that, the dataset is refreshed and any attached controls are reenabled.

TIP

Whenever you use `DisableControls()` to suspend a dataset's connection to data-aware controls, you should always use a `try..finally` block to ensure that the subsequent call to `EnableControls()` takes place whether or not an error occurs.

Testing for a Deleted Record

When viewing a dataset that includes deleted records, you'll probably need to determine as you navigate through the dataset which records are deleted and which aren't. Actually, you've already learned how to perform this check. You can obtain this information using the `DbiGetRecord()` function that you used to obtain the physical record number. The following code shows this procedure. The only material difference between this procedure and `GetRecNum()` is the checking of the `bDeletedFlag` field rather than the `iPhyRecNo` field of the `RECProps` record. Here's the code:

```
function TdBaseTable.GetIsDeleted: Boolean;
{ Returns a boolean indicating whether or not the current record }
{ has been soft deleted. }
var
  RP: RECProps;
begin
  if not FViewDeleted then     // don't bother if they aren't viewing
    Result := False            // deleted records
  else begin
    UpdateCursorPos;           // update BDE from Delphi
    { Get current record properties }
    Check(DbiGetRecord(Handle, dbiNOLOCK, Nil, @RP));
    Result := RP.bDeleteFlag;  // return flag from properties
  end;
end;
```

Undeleting a Record

So far, you've learned how to view deleted records as well as determine whether a record has been deleted, and, of course, you already know how to delete a record. The only other thing you need to learn regarding record deletion is how to undelete a record. Fortunately, the BDE makes this an easy task thanks to the `DbiUndeleteRecord()` function, which is defined as follows:

```
function DbiUndeleteRecord(hCursor: hDBICur): DBIResult stdcall;
```

The lone parameter is a cursor handle for the current dataset. Using this function, you can create an `UndeleteRecord()` method for `TdBaseTable` as shown here:

```
procedure TdBaseTable.UndeleteRecord;
begin
  if not IsDeleted then
    raise EDatabaseError.Create('Record is not deleted');
  Check(DbiUndeleteRecord(Handle));
  Refresh;
end;
```

Packing a Table

To remove soft-deleted records from a dBASE table, that table must go through a process called *packing*. For this, the BDE provides a function called `DbiPackTable()`, which is defined as follows:

```
function DbiPackTable(hDb: hDBIDb; hCursor: hDBICur;
    pszTableName: PChar; pszDriverType: PChar;
    bRegenIdxs: Bool): DBIResult stdcall;
```

`hDb` is a handle to a database. You should pass the `DBHandle` property of a `TDataSet` descendant or the `Handle` property of a `TDatabase` component in this parameter.

`hCursor` is a cursor handle. You should pass the `Handle` property of a `TDataSet` descendant in this parameter. You may also pass `Nil` if you want to instead use the `pszTableName` and `pszDriverType` parameters to identify the table.

`pszTableName` is a pointer to a string containing the name of the table.

`pszDriverType` is a pointer to a string representing the driver type of the table. If `hCursor` is `Nil`, this parameter must be set to `szDBASE`. As a side note, it's unusual that this parameter is required because this function is supported only for dBASE tables—we don't make the rules, we just play by them.

`bRegenIdxs` indicates whether or not you want to rebuild all out-of-date indexes associated with the table.

Here's the `Pack()` method for the `TdBaseTable` class:

```
procedure TdBaseTable.Pack(RegenIndexes: Boolean);
{ Packs the table in order to removed soft deleted records }
{ from the file. }
const
  SPackError = 'Table must be active and opened exclusively';
begin
  { Table must be active and opened exclusively }
  if not (Active and Exclusive) then
    raise EDatabaseError.Create(SPackError);
  try
    { Pack the table }
    Check(DbiPackTable(DBHandle, Handle, Nil, Nil, RegenIndexes));
  finally
    { update Delphi from BDE }
    CursorPosChanged;
    Refresh;
  end;
end;
```

The complete listing of the `TdBaseTable` object is provided in Listing 30.1, later in this chapter.

Paradox Tables

Paradox tables don't have as many nifty features, such as soft deletion, but they do carry the concept of a record number and table pack. In this section, you'll learn how to extend a `TTable` to perform these Paradox-specific tasks and to create a new `TParadoxTable` class.

Sequence Number

Paradox tables do not have the concept of a physical record number in the dBASE sense. They do, however, maintain the concept of a sequence number for each record in a table. The sequence number differs from the physical record number in that the sequence number is dependent on whatever index is currently applied to the table. The sequence number of a record is the order in which the record appears based on the current index.

The BDE makes it pretty easy to obtain a sequence number using the `DbiGetSeqNo()` function, which is defined as follows:

```
function DbiGetSeqNo(hCursor: hDBICur; var iSeqNo: Longint): DBIResult;
  stdcall;
```

`hCursor` is a cursor handle for a Paradox table, and the `iSeqNo` parameter will be filled in with the sequence number of the current record. The following code shows the `GetRecNum()` function for `TParadoxTable`:

```
cfunction TParadoxTable.GetRecNum: Longint;
{ Returns the sequence number of the current record. }
begin
  UpdateCursorPos;              // update BDE from Delphi
  { Get sequence number of current record into Result }
  Check(DbiGetSeqNo(Handle, Result));
end;
```

Table Packing

Table packing in Paradox has a different meaning than in dBASE because Paradox does not support soft deletion of records. When a record is deleted in Paradox, the record is removed from the table, but a "hole" is left in the database file where the record used to be. To compress these holes left by deleted records and make the table smaller and more efficient, you must pack the table.

Unlike with dBASE tables, there's no obvious BDE function you can use to pack a Paradox table. Instead, you must use `DbiDoRestructure()` to restructure the table and specify that the table should be packed as it's restructured. `DbiDoRestructure()` is defined as follows:

```
function DbiDoRestructure(hDb: hDBIDb; iTblDescCount: Word;
    pTblDesc: pCRTblDesc; pszSaveAs, pszKeyviolName,
    pszProblemsName: PChar; bAnalyzeOnly: Bool): DBIResult stdcall;
```

hDb is the handle to a database. However, because this function will not work when Delphi has the table open, you won't be able to use the DBHandle property of a TDataSet. To overcome this, the sample code (shown a bit later) that uses this function demonstrates how to create a temporary database.

iTblDescCount is the number of table descriptors. This parameter must be set to 1 because the current version of the BDE supports only one table descriptor per call.

pTblDesc is a pointer to a CRTblDesc record. This is the record that identifies the table and specifies how the table is to be restructured. This record is defined as follows:

```
type
  pCRTblDesc = ^CRTblDesc;
  CRTblDesc = packed record     // Create/Restruct Table descr
    szTblName    : DBITBLNAME;  // TableName incl. optional path & ext
    szTblType    : DBINAME;     // Driver type (optional)
    szErrTblName : DBIPATH;     // Error Table name (optional)
    szUserName   : DBINAME;     // User name (if applicable)
    szPassword   : DBINAME;     // Password (optional)
    bProtected   : WordBool;    // Master password supplied in szPassword
    bPack        : WordBool;    // Pack table (restructure only)
    iFldCount    : Word;        // Number of field defs supplied
    pecrFldOp    : pCROpType;   // Array of field ops
    pfldDesc     : pFLDDesc;    // Array of field descriptors
    iIdxCount    : Word;        // Number of index defs supplied
    pecrIdxOp    : pCROpType;   // Array of index ops
    pidxDesc     : PIDXDesc;    // Array of index descriptors
    iSecRecCount : Word;        // Number of security defs supplied
    pecrSecOp    : pCROpType;   // Array of security ops
    psecDesc     : pSECDesc;    // Array of security descriptors
    iValChkCount : Word;        // Number of val checks
    pecrValChkOp : pCROpType;   // Array of val check ops
    pvchkDesc    : pVCHKDesc;   // Array of val check descs
    iRintCount   : Word;        // Number of ref int specs
    pecrRintOp   : pCROpType;   // Array of ref int ops
    printDesc    : pRINTDesc;   // Array of ref int specs
    iOptParams   : Word;        // Number of optional parameters
    pfldOptParams: pFLDDesc;    // Array of field descriptors
    pOptData     : Pointer;     // Optional parameters
  end;
```

For Paradox table packing, it's only necessary to specify values for the szTblName, szTblType, and bPack fields.

pszSaveAs is an optional string pointer that identifies the destination table if it is different than the source table.

pszKeyviolName is an optional string pointer that identifies the table to which records that cause key violations during the restructure will be sent.

pszProblemsName is an optional string pointer that identifies the table to which records that cause problems during the restructure will be sent.

bAnalyzeOnly is unused.

The following code shows the Pack() method of TParadoxTable. You can see from the code how the CRTblDesc record is initialized and how the temporary database is created using the DbiOpenDatabase() function. Also note the finally block, which ensures that the temporary database is cleaned up after use.

```
procedure TParadoxTable.Pack;
var
  TblDesc: CRTblDesc;
  TempDBHandle: HDBIDb;
  WasActive: Boolean;
begin
  { Initialize TblDesc record }
  FillChar(TblDesc, SizeOf(TblDesc), 0); // fill with 0s
  with TblDesc do begin
    StrPCopy(szTblName, TableName);      // set table name
    StrCopy(szTblType, szPARADOX);      // set table type
    bPack := True;                       // set pack flag
  end;
  { Store table active state.  Must close table to pack. }
  WasActive := Active;
  if WasActive then Close;
  try
    { Create a temporary database.  Must be read-write/exclusive }
    Check(DbiOpenDatabase(PChar(DatabaseName), Nil, dbiREADWRITE,
          dbiOpenExcl, Nil, 0, Nil, Nil, TempDBHandle));
    try
      { Pack the table }
      Check(DbiDoRestructure(TempDBHandle, 1, @TblDesc, Nil, Nil, Nil,
            False));
    finally
      { Close the temporary database }
      DbiCloseDatabase(TempDBHandle);
    end;
  finally
    { Reset table active state }
    Active := WasActive;
  end;
end;
```

Listing 30.1 shows the `DDGTbls` unit in which the `TdBaseTable` and `TParadoxTable` objects are defined.

LISTING 30.1 The `DDGTbls.pas` Unit

```
unit DDGTbls;

interface

uses DB, DBTables, BDE;

type
  TdBaseTable = class(TTable)
  private
    FViewDeleted: Boolean;
    function GetIsDeleted: Boolean;
    function GetRecNum: Longint;
    procedure SetViewDeleted(Value: Boolean);
  protected
    function CreateHandle: HDBICur; override;
  public
    procedure Pack(RegenIndexes: Boolean);
    procedure UndeleteRecord;
    property IsDeleted: Boolean read GetIsDeleted;
    property RecNum: Longint read GetRecNum;
    property ViewDeleted: Boolean read FViewDeleted write SetViewDeleted;
  end;

  TParadoxTable = class(TTable)
  private
  protected
    function CreateHandle: HDBICur; override;
    function GetRecNum: Longint;
  public
    procedure Pack;
    property RecNum: Longint read GetRecNum;
  end;

implementation

uses SysUtils;

{ TdBaseTable }

function TdBaseTable.GetIsDeleted: Boolean;
{ Returns a boolean indicating whether or not the current record }
{ has been soft deleted. }
```

```
var
  RP: RECProps;
begin
  if not FViewDeleted then     // don't bother if they aren't viewing
    Result := False            // deleted records
  else begin
    UpdateCursorPos;           // update BDE from Delphi
    { Get current record properties }
    Check(DbiGetRecord(Handle, dbiNOLOCK, Nil, @RP));
    Result := RP.bDeleteFlag;  // return flag from properties
  end;
end;

function TdBaseTable.GetRecNum: Longint;
{ Returns the physical record number of the current record. }
var
  RP: RECProps;
begin
  UpdateCursorPos;             // update BDE from Delphi
  { Get current record properties }
  Check(DbiGetRecord(Handle, dbiNOLOCK, Nil, @RP));
  Result := RP.iPhyRecNum;     // return value from properties
end;

function TdBaseTable.CreateHandle: HDBICur;
{ Overridden from ancestor in order to perform a check to }
{ ensure that this is a dBASE table. }
var
  CP: CURProps;
begin
  Result := inherited CreateHandle;         // do inherited
  if Result <> Nil then begin
    { Get cursor properties, and raise exception if the }
    { table isn't using the dBASE driver. }
    Check(DbiGetCursorProps(Result, CP));
    if not (CP.szTableType = szdBASE) then
      raise EDatabaseError.Create('Not a dBASE table');
  end;
end;

procedure TdBaseTable.Pack(RegenIndexes: Boolean);
{ Packs the table in order to removed soft deleted records }
{ from the file. }
const
  SPackError = 'Table must be active and opened exclusively';
begin
```

continues

LISTING 30.1 Continued

```
  { Table must be active and opened exclusively }
  if not (Active and Exclusive) then
    raise EDatabaseError.Create(SPackError);
  try
    { Pack the table }
    Check(DbiPackTable(DBHandle, Handle, Nil, Nil, RegenIndexes));
  finally
    { update Delphi from BDE }
    CursorPosChanged;
    Refresh;
  end;
end;

procedure TdBaseTable.SetViewDeleted(Value: Boolean);
{ Allows the user to toggle between viewing and not viewing }
{ deleted records. }
begin
  { Table must be active }
  if Active and (FViewDeleted <> Value) then begin
    DisableControls;     // avoid flicker
    try
      { Magic BDE call to toggle view of soft deleted records }
      Check(DbiSetProp(hdbiObj(Handle), curSOFTDELETEON, Longint(Value)));
    finally
      Refresh;           // update Delphi
      EnableControls;    // flicker avoidance complete
    end;
    FViewDeleted := Value
  end;
end;

procedure TdBaseTable.UndeleteRecord;
begin
  if not IsDeleted then
    raise EDatabaseError.Create('Record is not deleted');
  Check(DbiUndeleteRecord(Handle));
  Refresh;
end;

function TParadoxTable.CreateHandle: HDBICur;
{ Overridden from ancestor in order to perform a check to }
{ ensure that this is a Paradox table. }
var
  CP: CURProps;
begin
  Result := inherited CreateHandle;         // do inherited
  if Result <> Nil then begin
    { Get cursor properties, and raise exception if the }
    { table isn't using the Paradox driver. }
    Check(DbiGetCursorProps(Result, CP));
    if not (CP.szTableType = szPARADOX) then
      raise EDatabaseError.Create('Not a Paradox table');
  end;
end;

function TParadoxTable.GetRecNum: Longint;
{ Returns the sequence number of the current record. }
begin
  UpdateCursorPos;             // update BDE from Delphi
  { Get sequence number of current record into Result }
  Check(DbiGetSeqNo(Handle, Result));
end;

procedure TParadoxTable.Pack;
var
  TblDesc: CRTblDesc;
  TempDBHandle: HDBIDb;
  WasActive: Boolean;
begin
  { Initialize TblDesc record }
  FillChar(TblDesc, SizeOf(TblDesc), 0); // fill with 0s
  with TblDesc do begin
    StrPCopy(szTblName, TableName);      // set table name
    szTblType := szPARADOX;              // set table type
    bPack := True;                       // set pack flag
  end;
  { Store table active state.  Must close table to pack. }
  WasActive := Active;
  if WasActive then Close;
  try
    { Create a temporary database.  Must be read-write/exclusive }
    Check(DbiOpenDatabase(PChar(DatabaseName), Nil, dbiREADWRITE,
          dbiOpenExcl, Nil, 0, Nil, Nil, TempDBHandle));
    try
      { Pack the table }
      Check(dbiDoRestructure(TempDBHandle, 1, @TblDesc, Nil, Nil, Nil,
            False));
    finally
      { Close the temporary database }
      DbiCloseDatabase(TempDBHandle);
    end;
  finally
```

continues

LISTING 30.1 Continued

```
    { Reset table active state }
    Active := WasActive;
  end;
end;

end.
```

Limiting TQuery Result Sets

Here's a classic SQL programming *faux pas*: Your application issues a SQL statement to the server that returns a result set consisting of a gazillion rows, thereby making the application user wait forever for the query to return and tying up precious server and network bandwidth. Conventional SQL wisdom dictates that one shouldn't issue queries that are so general that they cause so many records to be fetched. However, this is sometimes unavoidable, and `TQuery` doesn't seem to help matters much, because it doesn't provide a means for restricting the number of records in a result set to be fetched from the server. Fortunately, the BDE does provide this capability, and it's not very difficult to surface in a `TQuery` descendant.

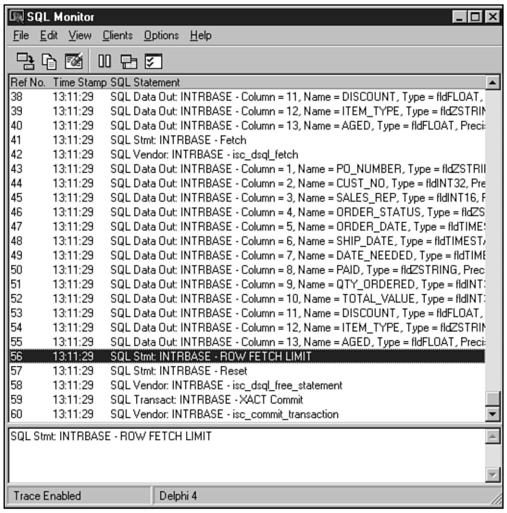
The BDE API call that performs this magic is the catchall `DbiSetProp()` function, which was explained earlier in this chapter. In this case, the first parameter to `DbiSetProp()` is the cursor handle for the query, the second parameter must be `curMAXROWS`, and the final parameter should be set to the maximum number of rows to which you want to restrict the result set.

The ideal place to make the call to this function is in the `PrepareCursor()` method of `TQuery`, which is called immediately after the query is opened. Listing 30.2 shows the `ResQuery` unit, in which the `TRestrictedQuery` component is defined.

LISTING 30.2 The `ResQuery.pas` Unit

```
unit ResQuery;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, DB, DBTables, BDE;

type
  TRestrictedQuery = class(TQuery)
  private
    FMaxRowCount: Longint;
  protected
    procedure PrepareCursor; override;
  published
    property MaxRowCount: Longint read FMaxRowCount write FMaxRowCount;
  end;

procedure Register;

implementation

procedure TRestrictedQuery.PrepareCursor;
begin
  inherited PrepareCursor;
  if FMaxRowCount > 0 then
    Check(DbiSetProp(hDBIObj(Handle), curMAXROWS, FMaxRowCount));
end;

procedure Register;
begin
  RegisterComponents('DDG', [TRestrictedQuery]);
end;

end.
```

You can limit the result set of a query by simply setting the `MaxRowCount` property to a value greater than zero. To further illustrate the point, Figure 30.1 shows the result of a query restricted to three rows, as shown in SQL Monitor.



```
Ref No. Time Stamp SQL Statement
38      13:11:29   SQL Data Out: INTRBASE - Column = 11, Name = DISCOUNT, Type = fldFLOAT,
39      13:11:29   SQL Data Out: INTRBASE - Column = 12, Name = ITEM_TYPE, Type = fldZSTRIN
40      13:11:29   SQL Data Out: INTRBASE - Column = 13, Name = AGED, Type = fldFLOAT, Preci
41      13:11:29   SQL Stmt: INTRBASE - Fetch
42      13:11:29   SQL Vendor: INTRBASE - isc_dsql_fetch
43      13:11:29   SQL Data Out: INTRBASE - Column = 1, Name = PO_NUMBER, Type = fldZSTRI
44      13:11:29   SQL Data Out: INTRBASE - Column = 2, Name = CUST_NO, Type = fldINT32, Pre
45      13:11:29   SQL Data Out: INTRBASE - Column = 3, Name = SALES_REP, Type = fldINT16, F
46      13:11:29   SQL Data Out: INTRBASE - Column = 4, Name = ORDER_STATUS, Type = fldZS
47      13:11:29   SQL Data Out: INTRBASE - Column = 5, Name = ORDER_DATE, Type = fldTIME
48      13:11:29   SQL Data Out: INTRBASE - Column = 6, Name = SHIP_DATE, Type = fldTIMEST
49      13:11:29   SQL Data Out: INTRBASE - Column = 7, Name = DATE_NEEDED, Type = fldTIME
50      13:11:29   SQL Data Out: INTRBASE - Column = 8, Name = PAID, Type = fldZSTRING, Prec
51      13:11:29   SQL Data Out: INTRBASE - Column = 9, Name = QTY_ORDERED, Type = fldINT
52      13:11:29   SQL Data Out: INTRBASE - Column = 10, Name = TOTAL_VALUE, Type = fldINT
53      13:11:29   SQL Data Out: INTRBASE - Column = 11, Name = DISCOUNT, Type = fldFLOAT,
54      13:11:29   SQL Data Out: INTRBASE - Column = 12, Name = ITEM_TYPE, Type = fldZSTRIN
55      13:11:29   SQL Data Out: INTRBASE - Column = 13, Name = AGED, Type = fldFLOAT, Preci
56      13:11:29   SQL Stmt: INTRBASE - ROW FETCH LIMIT
57      13:11:29   SQL Stmt: INTRBASE - Reset
58      13:11:29   SQL Vendor: INTRBASE - isc_dsql_free_statement
59      13:11:29   SQL Transact: INTRBASE - XACT Commit
60      13:11:29   SQL Vendor: INTRBASE - isc_commit_transaction

SQL Stmt: INTRBASE - ROW FETCH LIMIT

Trace Enabled   Delphi 4
```

FIGURE 30.1
A restricted query viewed from SQL Monitor.

BDE Miscellany

Through our development of database applications, we've found a few common development tasks that could serve to be automated a bit. Some of these miscellaneous tasks include performing SQL aggregate functions on a table, copying tables, and obtaining a list of Paradox users for a particular session.

SQL Aggregate Functions

Generally speaking, SQL aggregate functions are functions built into the SQL language that perform some arithmetic operation on one or more columns from one or more rows. Some common examples of this are `sum()`, which adds columns from multiple rows, `avg()`, which calculates the average value of columns from multiple rows, `min()`, which finds the minimum value of columns in multiple rows, and `max()`, which (as you might guess) determines the maximum value of columns within multiple rows.

Aggregate functions such as these can sometimes be inconvenient to use in Delphi. For example, if you're working with `TTables` to access data, using these functions involves creating a `TQuery`, formulating the correct SQL statement for the table and column in question, executing the query, and obtaining the result from the query. Clearly this is a process crying out to be automated, and the code in Listing 30.3 does just that.

LISTING 30.3 Automating SQL Aggregate Functions

```
type
  TSQLAggFunc = (safSum, safAvg, safMin, safMax);

const
  // SQL aggregate functions
  SQLAggStrs: array[TSQLAggFunc] of string = (
    'select sum(%s) from %s',
    'select avg(%s) from %s',
    'select min(%s) from %s',
    'select max(%s) from %s');

function CreateQueryFromTable(T: TTable): TQuery;
// returns a query hooked to the same database and session as table T
begin
  Result := TQuery.Create(nil);
  try
    Result.DatabaseName := T.DatabaseName;
    Result.SessionName := T.SessionName;
  except
    Result.Free;
    Raise;
  end;
end;

function DoSQLAggFunc(T: TTable; FieldNames: string;
  Func: TSQLAggFunc): Extended;
begin
  with CreateQueryFromTable(T) do
  begin
    try
      SQL.Add(Format(SQLAggStrs[Func], [FieldNames, T.TableName]));
      Open;
      Result := Fields[0].AsFloat;
    finally
      Free;
    end;
  end;
end;

function SumField(T: TTable; Field: String): Extended;
begin
  Result := DoSQLAggFunc(T, Field, safSum);
end;

function AvgField(T: TTable; Field: String): Extended;
begin
  Result := DoSQLAggFunc(T, Field, safAvg);
end;

function MinField(T: TTable; Field: String): Extended;
begin
  Result := DoSQLAggFunc(T, Field, safMin);
end;

function MaxField(T: TTable; Field: string): Extended;
begin
  Result := DoSQLAggFunc(T, Field, safMax);
end;
```

As you can see from the listing, each of the individual aggregate function wrappers call into the `DoSQLAggFun()` function. In this function, the `CreateQueryFromTable()` function creates and returns a `TQuery` component that uses the same database and session as the `TTable` passed in the `T` parameter. The proper SQL string is then formatted from an array of strings, the query is executed, and the query's result is returned from the function.

Quick Table Copy

If you want to make a copy of a table, traditional wisdom might dictate a few different courses of action. The first might be to use the Win32 API's `CopyFile()` function to physically copy the table file(s) from one location to another. Another option is to use a `TBatchMove` component to copy one `TTable` to another. Yet another option is to use `TTable`'s `BatchMove()` method to perform the copy.

However, there are problems with each of these traditional alternatives: A brute-force file copy using the `CopyFile()` API function may not work if the table files are open by another process or user, and it certainly will not work if the table exists within some type of database file on a SQL server. A file copy might become a very complex task if you consider that you may also have to copy associated index, BLOB, or value files. The use of `TBatchMove` would solve these problems, but only if you submit to the disadvantage of the complexity involved in using this component. An additional drawback is the fact that the batch move process is much slower than a direct file copy. Using `TTable.BatchMove()` does help to alleviate the issue of complexity in performing the table copy, but it doesn't overcome the performance shortcomings inherent in the batch move process.

Fortunately, the BDE developers also recognized this issue and made a BDE API function available that provides the best of both worlds: speed and ease of use. The function in question is `DbiCopyTable()`, and it's declared as shown here:

```
function DbiCopyTable (            { Copy one table to another }
      hDb           : hDBIDb;      { Database handle }
      bOverWrite    : Bool;        { True, to overwrite existing file }
      pszSrcTableName : PChar;     { Source table name }
      pszSrcDriverType : PChar;    { Source driver type }
      pszDestTableName : PChar     { Destination table name }
   ): DBIResult stdcall;
```

Because the BDE API function can't deal directly with VCL `TTable` components, the following procedure wraps `DbiCopyTable()` into a nifty routine to which you can pass a `TTable` and a destination table name:

```
procedure QuickCopyTable(T: TTable; DestTblName: string;
  Overwrite: Boolean);
// Copies TTable T to an identical table with name DestTblName.
// Will overwrite existing table with name DestTblName if Overwrite is
// True.
var
  DBType: DBINAME;
  WasOpen: Boolean;
  NumCopied: Word;
begin
```

```
  WasOpen := T.Active;        // save table active state
  if not WasOpen then T.Open; // ensure table is open
  // Get driver type string
  Check(DbiGetProp(hDBIObj(T.Handle), drvDRIVERTYPE, @DBType,
    SizeOf(DBINAME), NumCopied));
  // Copy the table
  Check(DbiCopyTable(T.DBHandle, Overwrite, PChar(T.TableName), DBType,
    PChar(DestTblName)));
  T.Active := WasOpen;        // restore active state
end;
```

NOTE

For local databases (Paradox, dBASE, Access, and FoxPro), all files associated with the table—index and BLOB files, for example—are copied to the destination table. For tables residing in a SQL database, only the table will be copied, and it's up to you to ensure that the necessary indexes and other elements are applied to the destination table.

Paradox Session Users

If your application uses Paradox tables, you may come across a situation where you need to determine which users are currently using a particular Paradox table. You can accomplish this with the `DbiOpenUserList()` BDE API function. This function provides a BDE cursor for a list of users for the current session. The following procedure demonstrates how to use this function effectively:

```
procedure GetPDoxUsersForSession(Sess: TSession; UserList: TStrings);
// Clears UserList and adds each user using the same netfile as session
// Sess to the list.  If Sess = nil, then procedure works for default
// net file.
var
  WasActive: Boolean;
  SessHand: hDBISes;
  ListCur: hDBICur;
  User: UserDesc;
begin
  if UserList = nil then Exit;
  UserList.Clear;
  if Assigned(Sess) then
  begin
    WasActive := Sess.Active;
    if not WasActive then Sess.Open;
    Check(DbiStartSession(nil, SessHand, PChar(Sess.NetFileDir)));
  end
  else
    Check(DbiStartSession(nil, SessHand, nil));
  try
    Check(DbiOpenUserList(ListCur));
    try
      while DbiGetNextRecord(ListCur, dbiNOLOCK, @User, nil) =
        DBIERR_NONE do
        UserList.Add(User.szUserName);
    finally
      DbiCloseCursor(ListCur);  // close "user list table" cursor
    end;
  finally
    DbiCloseSession(SessHand);
    if Assigned(Sess) then Sess.Active := WasActive;
  end;
end;
```

The interesting thing about the `DbiOpenUserList()` function is that it creates a cursor for a table that's manipulated in the same manner as any other BDE table cursor. In this case, `DbiGetNextRecord()` is called repeatedly until the end of the table is reached. The record buffer for this table follows the format of the `UserDesc` record, which is defined in the BDE unit as follows:

```
type
  pUSERDesc = ^USERDesc;
  USERDesc = packed record           { User description }
    szUserName   : DBIUSERNAME;      { User Name }
    iNetSession  : Word;             { Net level session number }
    iProductClass: Word;             { Product class of user }
    szSerialNum  : packed array [0..21] of Char; { Serial number }
  end;
```

Each call to `DbiGetNextRecord()` fills a `UserDesc` record called `User`, and the `szUserName` field of that record is added to the `UserList string list`.

TIP

Note the use of the `try..finally` resource protection blocks in the `GetPDoxUsersForSession()` procedure. These ensure that both the BDE resources associated with the session and cursor are properly released.

Writing Data-Aware VCL Controls

Chapter 21, "Writing Delphi Custom Components," and Chapter 22, "Advanced Component Techniques," provided you with thorough coverage of component-building techniques and methodologies. One large topic that wasn't covered, however, is data-aware controls. Actually, there isn't much more to creating a data-aware control than there is to creating a regular VCL control, but a typical data-aware component is different in four key respects:

- Data-aware controls maintain an internal data link object. A descendant of `TDataLink`, this object provides the means by which the control communicates with a `TDataSource`. For data-aware controls that connect to a single field of a dataset, this is usually a `TFieldDataLink`. The control should handle the `OnDataChange` event of the data link in order to receive notifications when the field or record data has changed.
- Data-aware controls must handle the `CM_GETDATALINK` message. The typical response to this message is to return the data link object in the message's `Result` field.
- Data-aware controls should surface a property of type `TDataSource` so the control can be connected to a data source by which it will communicate with a dataset. By convention, this property is called `DataSource`. Controls that connect to a single field should also surface a string property to hold the name of the field to which it is connected. By convention, this property is called `DataField`.
- Data-aware controls should override the `Notification()` method of `TComponent`. By overriding this method, the data-aware control can be notified if the data source component connected to the control has been deleted from the form.

To demonstrate the creation of a simple data-aware control, Listing 30.4 shows the `DBSound` unit. This unit contains the `TDBWavPlayer` component, a component that plays WAV sounds from a BLOB field in a dataset.

LISTING 30.4 The `DBSound.pas` Unit

```
unit DBSound;

interface

uses Windows, Messages, Classes, SysUtils, Controls, Buttons, DB,
  DBTables, DbCtrls;

type
  EDBWavError = class(Exception);

  TDBWavPlayer = class(TSpeedButton)
  private
```

continues

Listing 30.4 Continued

```
    FAutoPlay: Boolean;
    FDataLink: TFieldDataLink;
    FDataStream: TMemoryStream;
    FExceptOnError: Boolean;
    procedure DataChange(Sender: TObject);
    function GetDataField: string;
    function GetDataSource: TDataSource;
    function GetField: TField;
    procedure SetDataField(const Value: string);
    procedure SetDataSource(Value: TDataSource);
    procedure CMGetDataLink(var Message: TMessage); message
      CM_GETDATALINK;
    procedure CreateDataStream;
    procedure PlaySound;
  protected
    procedure Notification(AComponent: TComponent;
      Operation: TOperation); override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure Click; override;
    property Field: TField read GetField;
  published
    property AutoPlay: Boolean read FAutoPlay write FAutoPlay
      default False;
    property ExceptOnError: Boolean read FExceptOnError
      write FExceptOnError;
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource
      write SetDataSource;
  end;

implementation

uses MMSystem;

const
  // Error strings
  SNotBlobField = 'Field "%s" is not a blob field';
  SPlaySoundErr = 'Error attempting to play sound';

constructor TDBWavPlayer.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);             // call inherited
  FDataLink := TFieldDataLink.Create;   // create field data link
  FDataLink.OnDataChange := DataChange; // get data link notifications
  FDataStream := TMemoryStream.Create;  // create worker memory stream
end;

destructor TDBWavPlayer.Destroy;
begin
  FDataStream.Free;
  FDataLink.Free;
  FDataLink := Nil;
  inherited Destroy;
end;

procedure TDBWavPlayer.Click;
begin
  inherited Click; // do default behavior
  PlaySound;       // play the sound
end;

procedure TDBWavPlayer.CreateDataStream;
// creates memory stream from wave file in blob field
var
  BS: TBlobStream;
begin
  // make sure it's a blob field
  if not (Field is TBlobField) then
    raise EDBWavError.CreateFmt(SNotBlobField, [DataField]);
  // create a blob stream
  BS := TBlobStream.Create(TBlobField(Field), bmRead);
  try
    // copy from blob stream to memory stream
    FDataStream.SetSize(BS.Size);
    FDataStream.CopyFrom(BS, BS.Size);
  finally
    BS.Free;  // free blob stream
  end;
end;

procedure TDBWavPlayer.PlaySound;
// plays wave sound loaded in memory stream
begin
  // make sure we are hooked to a dataset and field
  if (DataSource <> nil) and (DataField <> '') then
  begin
    // make sure data stream is created
    if FDataStream.Size = 0 then CreateDataStream;
    // Play the sound in the memory stream, raise exception on error
```

continues

LISTING 30.4 Continued

```
    if (not MMSystem.PlaySound(FDataStream.Memory, 0, SND_ASYNC or
      SND_MEMORY)) and FExceptOnError then
      raise EDBWavError.Create(SPlaySoundErr);
  end;
end;

procedure TDBWavPlayer.DataChange(Sender: TObject);
// OnChange handler FFieldDataLink.DataChange
begin
  // deallocate memory occupied by previous wave file
  with FDataStream do if Size <> 0 then SetSize(0);
  // if AutoPlay is on, the play the sound
  if FAutoPlay then PlaySound;
end;

procedure TDBWavPlayer.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  // do some required housekeeping
  if (Operation = opRemove) and (FDataLink <> nil) and
    (AComponent = DataSource) then DataSource := nil;
end;

function TDBWavPlayer.GetDataSource: TDataSource;
begin
  Result := FDataLink.DataSource;
end;

procedure TDBWavPlayer.SetDataSource(Value: TDataSource);
begin
  FDataLink.DataSource := Value;
  if Value <> nil then Value.FreeNotification(Self);
end;

function TDBWavPlayer.GetDataField: string;
begin
  Result := FDataLink.FieldName;
end;

procedure TDBWavPlayer.SetDataField(const Value: string);
begin
  FDataLink.FieldName := Value;
end;
```

```
function TDBWavPlayer.GetField: TField;
begin
  Result := FDataLink.Field;
end;

procedure TDBWavPlayer.CMGetDataLink(var Message: TMessage);
begin
  Message.Result := Integer(FDataLink);
end;

end.
```

This component is a `TSpeedButton` descendant that, when pressed, can play a WAV sound residing in a database BLOB field. The `AutoPlay` property can also be set to `True`, which will cause the sound to play every time the user navigates to a new record in the table. When this property is set, it might also make sense to set the `Visible` property of the component to `False` so that a button doesn't appear visually on the form.

In the `FDataLink.OnChange` handler, `DataChange()`, the component works by extracting the BLOB field using a `TBlobStream` and copying the BLOB stream to a memory stream, `FDataStream`. When the sound is in a memory stream, you can play it using the `PlaySound()` Win32 API function.

Extending TDataSet

One of the marquee features of the database VCL is the abstract `TDataSet`, which provides the capability to manipulate non-BDE data sources within the database VCL framework.

In the Olden Days…

In previous versions of Delphi, the VCL database architecture was closed, making it was very difficult to manipulate non-BDE data sources using VCL components. Figure 30.2 illustrates the BDE-centric data set architecture found in Delphi 1 and 2.

As Figure 30.2 shows, `TDataSet` is essentially hard-coded for the BDE, and there's no room in this architecture for non-BDE data sources. Developers wanting to use non-BDE data sources within VCL had two choices:

- Creating a DLL that looks to VCL like the BDE but talks to a different type of data
- Throwing `TDataSet` out the window and writing their own dataset class and data-aware controls

Clearly, either of these options involves a very significant amount of work, and neither is a particularly elegant solution. Something had to be done.

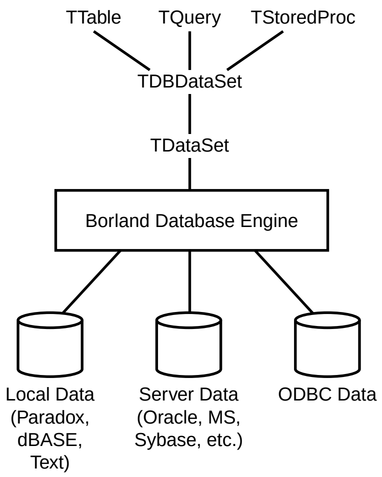


Figure 30.2
Delphi 1 and 2 VCL dataset architecture.

Modern Times

Recognizing these issues and the strong customer demand for easier access to non-BDE data sources, the Delphi development team made it a priority to extend VCL's data set architecture in Delphi 3. The idea behind the new architecture was to make the `TDataSet` class an abstraction of a VCL dataset and to move the BDE-specific data set code into the new `TBDEDataSet` class. Figure 30.3 provides an illustration of this new architecture.

When you understand how `TDataSet` was uncoupled from the BDE, the challenge becomes how to employ this concept to create a `TDataSet` descendant that manipulates some type of non-BDE data. And we're not using the term *challenge* loosely; creating a functional `TDataSet` descendant is not a task for the faint of heart. This is a fairly demanding task that requires familiarity with VCL database architecture and component writing.

Tip

Delphi provides two examples of creating a `TDataSet` descendant—one very simple and one very complicated. The simple example is the `TTextDataSet` class found in the `TextData` unit in the `\Delphi5\Demos\Db\TextData` directory. This example encapsulates `TStringList` as a one-field dataset. The complex example is the `TBDEDataSet` class found in the `DbTables` unit in the VCL source. As mentioned earlier, this class maps VCL's dataset architecture to the BDE.

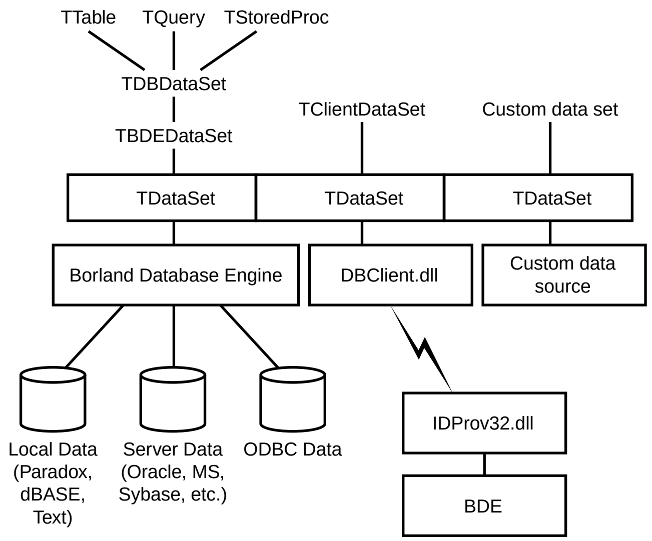


FIGURE 30.3
Delphi 3 and higher VCL dataset architecture.

Creating a TDataSet Descendant

Most dataset implementations will fall in between `TTextDataSet` and `TBDEDataSet` in terms of complexity. To provide an example of this, we'll demonstrate how to create a `TDataSet` descendant that manipulates an Object Pascal `file of record` (for a description of `file of record`, see Chapter 12, "Working With Files"). The following record and file type will be used for this example:

```
type
  // arbitrary-length array of char used for name field
  TNameStr = array[0..31] of char;

  // this record info represents the "table" structure:
  PDDGData = ^TDDGData;
  TDDGData = record
    Name: TNameStr;
    Height: Double;
    ShoeSize: Integer;
  end;

  // Pascal file of record which holds "table" data:
  TDDGDataFile = file of TDDGData;
```

An Object Pascal `file of record` can provide a convenient and efficient way to store information, but the format is inherently limited by its inability to insert records into or delete records from the middle of the file. For this reason, we'll use a two-file scheme to track the "table" information: the first, *data file*, being the `file of record`; the second, *index file*, maintaining a list of integers that represent seek values into the first file. This means that a record's position in the data file doesn't necessarily coincide with its position in the dataset. A record's position in the dataset is controlled by the order of the index file; the first integer in the index file contains the seek value of the first record into the data file, the second integer in the index file contains the next seek value into the data file, and so on.

In this section we'll discuss what's necessary to create a `TDataSet` descendant called `TDDGDataSet`, which communicates to this `file of record`.

TDataSet Abstract Methods

`TDataSet`, being an abstract class, is useless until you override the methods necessary for manipulation of some particular type of dataset. In particular, you must at least override each of `TDataSet`'s 23 abstract methods and perhaps some optional methods. For the sake of discussion, we've divided them into six logical groupings: record buffer methods, navigational methods, bookmark methods, editing methods, miscellaneous methods, and optional methods.

The following code shows an edited version of `TDataSet` as it is defined in `Db.pas`. For clarity, only the methods mentioned thus far are shown, and the methods are categorized based on the logical groupings we discussed. Here's the code:

```
type
  TDataSet = class(TComponent)
  { ... }
  protected
  { Record buffer methods }
    function AllocRecordBuffer: PChar; virtual; abstract;
    procedure FreeRecordBuffer(var Buffer: PChar); virtual; abstract;
    procedure InternalInitRecord(Buffer: PChar); virtual; abstract;
    function GetRecord(Buffer: PChar; GetMode: TGetMode;
      DoCheck: Boolean):
      TGetResult; virtual; abstract;
    function GetRecordSize: Word; virtual; abstract;
    function GetFieldData(Field: TField; Buffer: Pointer): Boolean;
      override;
    procedure SetFieldData(Field: TField; Buffer: Pointer); virtual;
      abstract;
  { Bookmark methods }
    function GetBookmarkFlag(Buffer: PChar): TBookmarkFlag; override;
    procedure SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag);
      override;
    procedure GetBookmarkData(Buffer: PChar; Data: Pointer); override;
    procedure SetBookmarkData(Buffer: PChar; Data: Pointer); override;
    procedure InternalGotoBookmark(Bookmark: Pointer); override;
    procedure InternalSetToRecord(Buffer: PChar); override;
  { Navigational methods }
    procedure InternalFirst; virtual; abstract;
    procedure InternalLast; virtual; abstract;
  { Editing methods }
    procedure InternalAddRecord(Buffer: Pointer; Append: Boolean);
      virtual; abstract;
    procedure InternalDelete; virtual; abstract;
    procedure InternalPost; virtual; abstract;
  { Miscellaneous methods }
    procedure InternalClose; virtual; abstract;
    procedure InternalHandleException; virtual; abstract;
    procedure InternalInitFieldDefs; virtual; abstract;
    procedure InternalOpen; virtual; abstract;
    function IsCursorOpen: Boolean; virtual; abstract;
  { optional methods }
    function GetRecordCount: Integer; virtual;
    function GetRecNo: Integer; virtual;
    procedure SetRecNo(Value: Integer); virtual;
  { ... }
  end;
```

Record Buffer Methods

You must override a number of methods that deal with record buffers. Actually, VCL does a pretty good job of hiding the gory details of its record buffer implementation; `TDataSet` will create and manage groups of buffers, so your job is primarily to decide what goes in the buffers and to move data between different buffers. Because it's a requirement for all `TDataSet` descendants to implement bookmarks, we'll store bookmark information after the record data in the record buffer. The record we'll use to describe bookmark information is as follows:

```
type
  // Bookmark information record to support TDataset bookmarks:
  PDDGBookmarkInfo = ^TDDGBookmarkInfo;
  TDDGBookmarkInfo = record
    BookmarkData: Integer;
    BookmarkFlag: TBookmarkFlag;
  end;
```

The `BookmarkData` field will represent a simple seek value into the data file. The `BookmarkFlag` field is used to determine whether the buffer contains a valid bookmark, and it will contain special values when the dataset is positioned on the BOF and EOF cracks.

NOTE

Keep in mind that this implementation of bookmarks and record buffers is specific to this solution. If you were creating a TDataSet descendant to manipulate some other type of data, you might choose to implement your record buffer or bookmarks differently. For example, the data source you're trying to encapsulate may natively support bookmarks.

Before examining the record buffer–specific methods, first take a look at the constructor for the `TDDGDataSet` class:

```
constructor TDDGDataSet.Create(AOwner: TComponent);
begin
  FIndexList := TIndexList.Create;
  FRecordSize := SizeOf(TDDGData);
  FBufferSize := FRecordSize + SizeOf(TDDGBookmarkInfo);
  inherited Create(AOwner);
end;
```

This constructor does three important things: First, it creates the `TIndexList` object. This list object is used as the index file described earlier to maintain order in the dataset. Next, the `FRecordSize` and `FBufferSize` fields are initialized. `FRecordSize` holds the size of the data record, and `FBufferSize` represents the total size of the record buffer (the data record size plus the size of the bookmark information record). Finally, this method calls the inherited constructor to perform the default `TDataSet` setup.

The following are `TDataSet` methods that deal with record buffers that must be overridden in a descendant. Except for `GetFieldData()`, all are declared as abstract in the base class:

```
function AllocRecordBuffer: PChar; override;
procedure FreeRecordBuffer(var Buffer: PChar); override;
procedure InternalInitRecord(Buffer: PChar); override;
function GetRecord(Buffer: PChar; GetMode: TGetMode;
  DoCheck: Boolean): TGetResult; override;
function GetRecordSize: Word; override;
function GetFieldData(Field: TField; Buffer: Pointer): Boolean; override;
procedure SetFieldData(Field: TField; Buffer: Pointer); override;
```

AllocRecordBuffer()

The `AllocRecordBuffer()` method is called to allocate memory for a single record buffer. In this implementation of the method, the `AllocMem()` function is used to allocate enough memory to hold both the record data and the bookmark data:

```
function TDDGDataSet.AllocRecordBuffer: PChar;
begin
  Result := AllocMem(FBufferSize);
end;
```

FreeRecordBuffer()

As you might expect, `FreeRecordBuffer()` must free the memory allocated by the `AllocRecordBuffer()` method. It's implemented using the `FreeMem()` procedure, as shown here:

```
procedure TDDGDataSet.FreeRecordBuffer(var Buffer: PChar);
begin
  FreeMem(Buffer);
end;
```

InternalInitRecord()

The `InternalInitRecord()` method is called to initialize a record buffer. In this method, you can do things such as setting default field values and performing some type of initialization of custom record buffer data. In this case, we simply zero-initialize the record buffer:

```
procedure TDDGDataSet.InternalInitRecord(Buffer: PChar);
begin
  FillChar(Buffer^, FBufferSize, 0);
end;
```

GetRecord()

The primary function of the `GetRecord()` method is to retrieve the record data for either the previous, current, or next record in the dataset. The return value of this function is of type `TGetResult`, which is defined in the `Db` unit as follows:

```
TGetResult = (grOK, grBOF, grEOF, grError);
```

The meaning of each of the enumerations is pretty much self-explanatory: `grOk` means success, `grBOF` means the dataset is at the beginning, `grEOF` means the dataset is at the end, and `grError` means an error has occurred.

The implementation of this method is as follows:

```
function TDDGDataSet.GetRecord(Buffer: PChar; GetMode: TGetMode;
  DoCheck: Boolean): TGetResult;
var
  IndexPos: Integer;
begin
 if FIndexList.Count < 1 then
    Result := grEOF
  else begin
```

```
    Result := grOk;
    case GetMode of
      gmPrior:
        if FRecordPos <= 0 then
        begin
          Result := grBOF;
          FRecordPos := -1;
        end
        else
          Dec(FRecordPos);
      gmCurrent:
        if (FRecordPos < 0) or (FRecordPos >= RecordCount) then
           Result := grError;
      gmNext:
        if FRecordPos >= RecordCount-1 then
          Result := grEOF
        else
          Inc(FRecordPos);
    end;
    if Result = grOk then
    begin
      IndexPos := Integer(FIndexList[FRecordPos]);
      Seek(FDataFile, IndexPos);
      BlockRead(FDataFile, PDDGData(Buffer)^, 1);
      with PDDGBookmarkInfo(Buffer + FRecordSize)^ do
      begin
        BookmarkData := FRecordPos;
        BookmarkFlag := bfCurrent;
      end;
    end
    else if (Result = grError) and DoCheck then
      DatabaseError('No records');
  end;
end;
```

The `FRecordPos` field tracks the current record position in the dataset. You'll notice that `FRecordPos` is incremented or decremented, as appropriate, when `GetRecord()` is called to obtain the next or previous record. If `FRecordPos` contains a valid record number, `FRecordPos` is used as an index into `FIndexList`. The number at that index is a seek value into the data file, and the record data is read from that position in the data file into the buffer specified by the `Buffer` parameter.

`GetRecord()` also has one additional job: When the `DoCheck` parameter is `True` and `grError` is the potential return value, an exception should be raised.

GetRecordSize()

The GetRecordSize() method should return the size, in bytes, of the record data portion of the record buffer. Be careful not to return the size of the entire record buffer; just return the size of the data portion. In this implementation, we return the value of the FRecordSize field:

```
function TDDGDataSet.GetRecordSize: Word;
begin
  Result := FRecordSize;
end;
```

GetFieldData()

The GetFieldData() method is responsible for copying data from the active record buffer (as provided by the ActiveBuffer property) into a field buffer. This is often accomplished most expediently using the Move() procedure. You can differentiate which field to copy using Field's Index or Name property. Also, be sure to copy from the correct offset into ActiveBuffer because ActiveBuffer contains a complete record's data and Buffer only holds one field's data. This implementation copies the fields from the internal buffer structure to its respective TField:

```
function TDDGDataSet.GetFieldData(Field: TField; Buffer: Pointer):
  Boolean;
begin
  Result := True;
  case Field.Index of
    0:
      begin
        Move(ActiveBuffer^, Buffer^, Field.Size);
        Result := PChar(Buffer)^ <> #0;
      end;
    1: Move(PDDGData(ActiveBuffer)^.Height, Buffer^, Field.DataSize);
    2: Move(PDDGData(ActiveBuffer)^.ShoeSize, Buffer^, Field.DataSize);
  end;
end;
```

Both this method and SetFieldData() can become much more complex if you want to support more advanced features such as calculated fields and filters.

SetFieldData()

The purpose of SetFieldData() is inverse to that of GetFieldData(); SetFieldData() copies data from a field buffer into the active record buffer. As you can see from the following code, the implementations of these two methods are very similar:

```
procedure TDDGDataSet.SetFieldData(Field: TField; Buffer: Pointer);
begin
  case Field.Index of
```

```
    0: Move(Buffer^, ActiveBuffer^, Field.Size);
    1: Move(Buffer^, PDDGData(ActiveBuffer)^.Height, Field.DataSize);
    2: Move(Buffer^, PDDGData(ActiveBuffer)^.ShoeSize, Field.DataSize);
  end;
  DataEvent(deFieldChange, Longint(Field));
end;
```

After the data is copied, the `DataEvent()` method is called to signal that a field has changed and fire the `OnChange` event of the field.

Bookmark Methods

We mentioned earlier that bookmark support is required for `TDataSet` descendants. The following abstract methods of `TDataSet` are overridden to provide this support:

```
function GetBookmarkFlag(Buffer: PChar): TBookmarkFlag; override;
procedure SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag); override;
procedure GetBookmarkData(Buffer: PChar; Data: Pointer); override;
procedure SetBookmarkData(Buffer: PChar; Data: Pointer); override;
procedure InternalGotoBookmark(Bookmark: Pointer); override;
procedure InternalSetToRecord(Buffer: PChar); override;
```

For `TDDGDataSet`, you'll see that the implementations of these methods revolve mostly around manipulating the bookmark information tacked onto the end of the record buffer.

GetBookmarkFlag() and SetBookmarkFlag()

Bookmark flags are used internally by `TDataSet` to determine whether a particular record is the first or last in the dataset. For this purpose, you must override the `GetBookmarkFlag()` and `SetBookmarkFlag()` methods. The `TDDGDataSet` implementation of these methods reads from and writes to the record buffer to keep track of this information, as shown here:

```
function TDDGDataSet.GetBookmarkFlag(Buffer: PChar): TBookmarkFlag;
begin
  Result := PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkFlag;
end;

procedure TDDGDataSet.SetBookmarkFlag(Buffer: PChar;
  Value: TBookmarkFlag);
begin
  PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkFlag := Value;
end;
```

GetBookmarkData() and SetBookmarkData()

The `GetBookmarkData()` and `SetBookmarkData()` methods provide a means for `TDataSet` to manipulate a record's bookmark data without repositioning the current record. As you can see, these methods are implemented in a manner similar to the methods described in the preceding example:

```
procedure TDDGDataSet.GetBookmarkData(Buffer: PChar; Data: Pointer);
begin
  PInteger(Data)^ :=PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkData;
end;

procedure TDDGDataSet.SetBookmarkData(Buffer: PChar; Data: Pointer);
begin
  PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkData :=PInteger(Data)^;
end;
```

InternalGotoBookmark()

The `InternalGotoBookmark()` method is called to reposition the current record to that represented by the `Bookmark` parameter. Because a bookmark value is the same as the record number for `TDDGDataSet`, the implementation of this method is straightforward:

```
procedure TDDGDataSet.InternalGotoBookmark(Bookmark: Pointer);
begin
  FRecordPos := Integer(Bookmark);
end;
```

InternalSetToRecord()

`InternalSetToRecord()` is similar to `InternalGotoBookmark()` except that it receives as a parameter a record buffer instead of a bookmark value. The job of this method is to position the dataset to the record provided in the `Buffer` parameter. This implementation of a record buffer contains the bookmark information because the bookmark value is the same as the record position; therefore, the implementation of this method is a one-liner:

```
procedure TDDGDataSet.InternalSetToRecord(Buffer: PChar);
begin
  // bookmark value is the same as an offset into the file
  FRecordPos := PDDGBookmarkInfo(Buffer + FRecordSize)^.Bookmarkdata;
end;
```

Navigational Methods

You must override several abstract navigational methods in `TDataSet` in order to position the dataset on the first or last record:

```
procedure InternalFirst; override;
procedure InternalLast; override;
```

The implementations of these methods are quite simple; `InternalFirst()` sets the `FRecordPos` value to `-1` (the BOF crack value), and `InternalLast()` sets the record position to the record count. Because the record index is zero based, the count is 1 greater than the last index (the EOF crack). Here's an example:

```
procedure TDDGDataSet.InternalFirst;
begin
```

```
  FRecordPos := -1;
end;

procedure TDDGDataSet.InternalLast;
begin
  FRecordPos := FIndexList.Count;
end;
```

Editing Methods

Three abstract `TDataSet` methods must be overridden in order to allow for the editing, appending, inserting, and deleting of records:

```
procedure InternalAddRecord(Buffer: Pointer; Append: Boolean); override;
procedure InternalDelete; override;
procedure InternalPost; override;
```

InternalAddRecord()

`InternalAddRecord()` is called when a record is inserted or appended to the dataset. The `Buffer` parameter points to the record buffer to be added to the dataset, and the `Append` parameter is `True` when a record is being appended and `False` when a record is being inserted. The `TDDGDataSet` implementation of this method seeks to the end of the data file, writes the record data to the file, and then adds or inserts the data file seek value into the appropriate position in the index list:

```
procedure TDDGDataSet.InternalAddRecord(Buffer: Pointer;
  Append: Boolean);
var
RecPos: Integer;
begin
  Seek(FDataFile, FileSize(FDataFile));
  BlockWrite(FDataFile, PDDGData(Buffer)^, 1);
  if Append then
  begin
    FIndexList.Add(Pointer(FileSize(FDataFile) - 1));
    InternalLast;
  end
  else begin
    if FRecordPos = -1 then RecPos := 0
    else RecPos := FRecordPos;
    FIndexList.Insert(RecPos, Pointer(FileSize(FDataFile) - 1));
  end;
  FIndexList.SaveToFile(FIdxName);
end;
```

InternalDelete()

The `InternalDelete()` method deletes the current record from the dataset. Because it's not practical to remove a record from the middle of the data file, the current record is deleted from the index list. This, in effect, orphans the deleted record in the data file by removing the index entry for a data record. Here's an example:

```
procedure TDDGDataSet.InternalDelete;
begin
  FIndexList.Delete(FRecordPos);
  if FRecordPos >= FIndexList.Count then Dec(FRecordPos);
end;
```

NOTE

This method of deletion means that the data file will not shrink in size even as records are deleted (similar to dBASE files). If you intend to use this type of dataset for commercial work, a good addition would be a file-pack method, which removes orphaned records from the data file.

InternalPost()

The `InternalPost()` method is called by `TDataSet.Post()`. In this method, you should write the data from the active record buffer to the data file. You'll note that the implementation of this method is quite similar to that of `InternalAddRecord()`, as shown here:

```
procedure TDDGDataSet.InternalPost;
var
  RecPos, InsPos: Integer;
begin
  if FRecordPos = -1 then
    RecPos := 0
  else begin
    if State = dsEdit then RecPos := Integer(FIndexList[FRecordPos])
    else RecPos := FileSize(FDataFile);
  end;
  Seek(FDataFile, RecPos);
  BlockWrite(FDataFile, PDDGData(ActiveBuffer)^, 1);
  if State <> dsEdit then
  begin
    if FRecordPos = -1 then InsPos := 0
    else InsPos := FRecordPos;
    FIndexList.Insert(InsPos, Pointer(RecPos));
  end;
  FIndexList.SaveToFile(FIdxName);
end;
```

Miscellaneous Methods

Several other abstract methods must be overridden in order to create a working `TDataSet` descendant. These are general housekeeping methods, and because these methods can't be pigeonholed into a particular category, we'll call them miscellaneous methods. These methods are as follows:

```
procedure InternalClose; override;
procedure InternalHandleException; override;
procedure InternalInitFieldDefs; override;
procedure InternalOpen; override;
function IsCursorOpen: Boolean; override;
```

InternalClose()

`InternalClose()` is called by `TDataSet.Close()`. In this method, you should deallocate all resources associated with the dataset that were allocated by `InternalOpen()` or that were allocated throughout the course of using the dataset. In this implementation, the data file is closed, and we ensure that the index list has been persisted to disk. Additionally, the `FRecordPos` is set to the BOF crack, and the data file record is zeroed out:

```
procedure TDDGDataSet.InternalClose;
begin
  if TFileRec(FDataFile).Mode <> 0 then
    CloseFile(FDataFile);
  FIndexList.SaveToFile(FIdxName);
  FIndexList.Clear;
  if DefaultFields then
    DestroyFields;
  FRecordPos := -1;
  FillChar(FDataFile, SizeOf(FDataFile), 0);
end;
```

InternalHandleException()

`InternalHandleException()` is called if an exception is raised while this component is being read from or written to a stream. Unless you have a specific need to handle these exceptions, you should implement this method as follows:

```
procedure TDDGDataSet.InternalHandleException;
begin
  // standard implementation for this method:
  Application.HandleException(Self);
end;
```

InternalInitFieldDefs()

In the `InternalInitFieldDefs()` method is where you should define the fields contained in the dataset. This is done by instantiating `TFieldDef` objects, passing the `TDataSet`'s `FieldDefs` property as the `Owner`. In this case, three `TFieldDef` objects are created, representing the three fields in this dataset:

```
procedure TDDGDataSet.InternalInitFieldDefs;
begin
  // create FieldDefs which map to each field in the data record
  FieldDefs.Clear;
  TFieldDef.Create(FieldDefs, 'Name', ftString, SizeOf(TNameStr), False,
   1);
  TFieldDef.Create(FieldDefs, 'Height', ftFloat, 0, False, 2);
  TFieldDef.Create(FieldDefs, 'ShoeSize', ftInteger, 0, False, 3);
end;
```

InternalOpen()

The `InternalOpen()` method is called by `TDataSet.Open()`. In this method, you should open the underlying data source, initialize any internal fields or properties, create the field defs if necessary, and bind the field defs to the data. The following implementation of this method opens the data file, loads the index list from a file, initializes the `FRecordPos` field and the `BookmarkSize` property, and creates and binds the field defs. You'll see from the following code that this method also gives the user a chance to create the database files if they aren't found on disk:

```
procedure TDDGDataSet.InternalOpen;
var
  HFile: THandle;
begin
  // make sure table and index files exist
  FIdxName := ChangeFileExt(FTableName, feDDGIndex);
  if not (FileExists(FTableName) and FileExists(FIdxName)) then
  begin
    if MessageDlg('Table or index file not found.  Create new table?',
      mtConfirmation, [mbYes, mbNo], 0) = mrYes then
    begin
      HFile := FileCreate(FTableName);
      if HFile = INVALID_HANDLE_VALUE then
        DatabaseError('Error creating table file');
      FileClose(HFile);
      HFile := FileCreate(FIdxName);
      if HFile = INVALID_HANDLE_VALUE then
        DatabaseError('Error creating index file');
      FileClose(HFile);
    end
    else
      DatabaseError('Could not open table');
  end;
  // open data file
  FileMode := fmShareDenyNone or fmOpenReadWrite;
  AssignFile(FDataFile, FTableName);
  Reset(FDataFile);
  try
    FIndexList.LoadFromFile(FIdxName); //initialize index TList from file
    FRecordPos := -1;                  //initial record pos before BOF
    BookmarkSize := SizeOf(Integer);   //initialize bookmark size for VCL
    InternalInitFieldDefs;             //initialize FieldDef objects
    // Create TField components when no persistent fields have been
    // created
    if DefaultFields then CreateFields;
    BindFields(True);                  //bind FieldDefs to actual data
  except
    CloseFile(FDataFile);
    FillChar(FDataFile, SizeOf(FDataFile), 0);
    raise;
  end;
end;
```

NOTE

Any resource allocations made in `InternalOpen()` should be freed in `InternalClose()`.

IsCursorOpen()

The `IsCursorOpen()` method is called internal to `TDataSet` while the dataset is being opened in order to determine whether data is available even though the dataset is inactive. The `TDDGData` implementation of this method returns `True` only if the data file has been opened, as shown here:

```
function TDDGDataSet.IsCursorOpen: Boolean;
begin
  // "Cursor" is open if data file is open.  File is open if FDataFile's
  // Mode includes the FileMode in which the file was open.
  Result := TFileRec(FDataFile).Mode <> 0;
end;
```

TIP

The preceding method illustrates an interesting feature of Object Pascal: a *file of record* or untyped file can be typecast to a `TFileRec` in order to obtain low-level information about the file. `TFileRec` is described in Chapter 12, “Working with Files.”

Optional Record Number Methods

If you want to take advantage of `TDBGrid`'s ability to scroll relative to the cursor position in the dataset, you must override three methods:

```
function GetRecordCount: Integer; override;
function GetRecNo: Integer; override;
procedure SetRecNo(Value: Integer); override;
```

Although this feature makes sense for this implementation, in many cases this capability isn't practical or even possible. For example, if you're working with a huge amount of data, it might not be practical to obtain a record count, or if you're communicating with a SQL server, this information might not even be available.

This `TDataSet` implementation is fairly simple, and these methods are appropriately straightforward to implement:

```
function TDDGDataSet.GetRecordCount: Integer;
begin
  Result := FIndexList.Count;
end;

function TDDGDataSet.GetRecNo: Integer;
begin
  UpdateCursorPos;
  if (FRecordPos = -1) and (RecordCount > 0) then
    Result := 1
  else
    Result := FRecordPos + 1;
end;

procedure TDDGDataSet.SetRecNo(Value: Integer);
begin
  if (Value >= 0) and (Value <= FIndexList.Count-1) then
  begin
    FRecordPos := Value - 1;
    Resync([]);
  end;
end;
```

TDDGDataSet

Listing 30.5 shows the `DDG_DS` unit, which contains the complete implementation of the `TDDGDataSet` unit.

LISTING 30.5 The `DDG_DS.pas` Unit

```
unit DDG_DS;

interface

uses Windows, Db, Classes, DDG_Rec;

type

  // Bookmark information record to support TDataset bookmarks:
  PDDGBookmarkInfo = ^TDDGBookmarkInfo;
  TDDGBookmarkInfo = record
    BookmarkData: Integer;
    BookmarkFlag: TBookmarkFlag;
  end;

  // List used to maintain access to file of record:
  TIndexList = class(TList)
  public
    procedure LoadFromFile(const FileName: string); virtual;
    procedure LoadFromStream(Stream: TStream); virtual;
    procedure SaveToFile(const FileName: string); virtual;
    procedure SaveToStream(Stream: TStream); virtual;
  end;

  // Specialized DDG TDataset descendant for our "table" data:
  TDDGDataSet = class(TDataSet)
  private
    function GetDataFileSize: Integer;
  public
    FDataFile: TDDGDataFile;
    FIdxName: string;
    FIndexList: TIndexList;
    FTableName: string;
    FRecordPos: Integer;
    FRecordSize: Integer;
    FBufferSize: Integer;
    procedure SetTableName(const Value: string);
  protected
    { Mandatory overrides }
    // Record buffer methods:
    function AllocRecordBuffer: PChar; override;
    procedure FreeRecordBuffer(var Buffer: PChar); override;
    procedure InternalInitRecord(Buffer: PChar); override;
    function GetRecord(Buffer: PChar; GetMode: TGetMode;
      DoCheck: Boolean): TGetResult; override;
    function GetRecordSize: Word; override;
    procedure SetFieldData(Field: TField; Buffer: Pointer); override;
    // Bookmark methods:
    procedure GetBookmarkData(Buffer: PChar; Data: Pointer); override;
    function GetBookmarkFlag(Buffer: PChar): TBookmarkFlag; override;
    procedure InternalGotoBookmark(Bookmark: Pointer); override;
    procedure InternalSetToRecord(Buffer: PChar); override;
    procedure SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag);
      override;
    procedure SetBookmarkData(Buffer: PChar; Data: Pointer); override;
    // Navigational methods:
    procedure InternalFirst; override;
    procedure InternalLast; override;
    // Editing methods:
    procedure InternalAddRecord(Buffer: Pointer; Append: Boolean);
      override;
    procedure InternalDelete; override;
    procedure InternalPost; override;
    // Misc methods:
    procedure InternalClose; override;
    procedure InternalHandleException; override;
    procedure InternalInitFieldDefs; override;
    procedure InternalOpen; override;
    function IsCursorOpen: Boolean; override;
    { Optional overrides }
    function GetRecordCount: Integer; override;
    function GetRecNo: Integer; override;
    procedure SetRecNo(Value: Integer); override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    function GetFieldData(Field: TField; Buffer: Pointer): Boolean;
      override;

    // Additional procedures
    procedure EmptyTable;
  published
    property Active;
    property TableName: string read FTableName write SetTableName;
    property BeforeOpen;
    property AfterOpen;
    property BeforeClose;
    property AfterClose;
    property BeforeInsert;
    property AfterInsert;
    property BeforeEdit;
```

continues

LISTING 30.5 Continued

```
    property AfterEdit;
    property BeforePost;
    property AfterPost;
    property BeforeCancel;
    property AfterCancel;
    property BeforeDelete;
    property AfterDelete;
    property BeforeScroll;
    property AfterScroll;
    property OnDeleteError;
    property OnEditError;

    // Additional Properties
    property DataFileSize: Integer read GetDataFileSize;
  end;

procedure Register;

implementation

uses BDE, DBTables, SysUtils, DBConsts, Forms, Controls, Dialogs;

const
  feDDGTable = '.ddg';
  feDDGIndex = '.ddx';
  // note that file is not being locked!

{ TIndexList }

procedure TIndexList.LoadFromFile(const FileName: string);
var
  F: TFileStream;
begin
  F := TFileStream.Create(FileName, fmOpenRead or fmShareDenyWrite);
  try
    LoadFromStream;
  finally
    F.Free;
  end;
end;

procedure TIndexList.LoadFromStream(Stream: TStream);
var
  Value: Integer;
begin
```

```
  while Stream.Position < Stream.Size do
  begin
    Stream.Read(Value, SizeOf(Value));
    Add(Pointer(Value));
  end;
  ShowMessage(IntToStr(Count));
end;

procedure TIndexList.SaveToFile(const FileName: string);
var
  F: TFileStream;
begin
  F := TFileStream.Create(FileName, fmCreate or fmShareExclusive);
  try
    SaveToStream(F);
  finally
    F.Free;
  end;
end;

procedure TIndexList.SaveToStream(Stream: TStream);
var
  i: Integer;
  Value: Integer;
begin
  for i := 0 to Count - 1 do
  begin
    Value := Integer(Items[i]);
    Stream.Write(Value, SizeOf(Value));
  end;
end;

{ TDDGDataSet }

constructor TDDGDataSet.Create(AOwner: TComponent);
begin
  FIndexList := TIndexList.Create;
  FRecordSize := SizeOf(TDDGData);
  FBufferSize := FRecordSize + SizeOf(TDDGBookmarkInfo);
  inherited Create(AOwner);
end;

destructor TDDGDataSet.Destroy;
begin
  inherited Destroy;
  FIndexList.Free;
```

continues

LISTING 30.5 Continued

```
end;

function TDDGDataSet.AllocRecordBuffer: PChar;
begin
  Result := AllocMem(FBufferSize);
end;

procedure TDDGDataSet.FreeRecordBuffer(var Buffer: PChar);
begin
  FreeMem(Buffer);
end;

procedure TDDGDataSet.InternalInitRecord(Buffer: PChar);
begin
  FillChar(Buffer^, FBufferSize, 0);
end;

function TDDGDataSet.GetRecord(Buffer: PChar; GetMode: TGetMode;
  DoCheck: Boolean): TGetResult;
var
  IndexPos: Integer;
begin
 if FIndexList.Count < 1 then
    Result := grEOF
  else begin
    Result := grOk;
    case GetMode of
      gmPrior:
        if FRecordPos <= 0 then
        begin
          Result := grBOF;
          FRecordPos := -1;
        end
        else
          Dec(FRecordPos);
      gmCurrent:
        if (FRecordPos < 0) or (FRecordPos >= RecordCount) then
           Result := grError;
      gmNext:
        if FRecordPos >= RecordCount-1 then
          Result := grEOF
        else
          Inc(FRecordPos);
    end;
    if Result = grOk then
    begin
      IndexPos := Integer(FIndexList[FRecordPos]);
      Seek(FDataFile, IndexPos);
      BlockRead(FDataFile, PDDGData(Buffer)^, 1);
      with PDDGBookmarkInfo(Buffer + FRecordSize)^ do
      begin
        BookmarkData := FRecordPos;
        BookmarkFlag := bfCurrent;
      end;
    end
    else if (Result = grError) and DoCheck then
      DatabaseError('No records');
  end;
end;

function TDDGDataSet.GetRecordSize: Word;
begin
  Result := FRecordSize;
end;

function TDDGDataSet.GetFieldData(Field: TField; Buffer: Pointer):
  Boolean;
begin
  Result := True;
  case Field.Index of
    0:
      begin
        Move(ActiveBuffer^, Buffer^, Field.Size);
        Result := PChar(Buffer)^ <> #0;
      end;
    1: Move(PDDGData(ActiveBuffer)^.Height, Buffer^, Field.DataSize);
    2: Move(PDDGData(ActiveBuffer)^.ShoeSize, Buffer^, Field.DataSize);
  end;
end;

procedure TDDGDataSet.SetFieldData(Field: TField; Buffer: Pointer);
begin
  case Field.Index of
    0: Move(Buffer^, ActiveBuffer^, Field.Size);
    1: Move(Buffer^, PDDGData(ActiveBuffer)^.Height, Field.DataSize);
    2: Move(Buffer^, PDDGData(ActiveBuffer)^.ShoeSize, Field.DataSize);
  end;
  DataEvent(deFieldChange, Longint(Field));
end;

procedure TDDGDataSet.GetBookmarkData(Buffer: PChar; Data: Pointer);
```

continues

LISTING 30.5 Continued

```
begin
  PInteger(Data)^ :=PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkData;
end;

function TDDGDataSet.GetBookmarkFlag(Buffer: PChar): TBookmarkFlag;
begin
  Result := PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkFlag;
end;

procedure TDDGDataSet.InternalGotoBookmark(Bookmark: Pointer);
begin
  FRecordPos := Integer(Bookmark);
end;

procedure TDDGDataSet.InternalSetToRecord(Buffer: PChar);
begin
  // bookmark value is the same as an offset into the file
  FRecordPos := PDDGBookmarkInfo(Buffer + FRecordSize)^.Bookmarkdata;
end;

procedure TDDGDataSet.SetBookmarkData(Buffer: PChar; Data: Pointer);
begin
  PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkData :=PInteger(Data)^;
end;

procedure TDDGDataSet.SetBookmarkFlag(Buffer: PChar;
  Value: TBookmarkFlag);
begin
  PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkFlag := Value;
end;

procedure TDDGDataSet.InternalFirst;
begin
  FRecordPos := -1;
end;

procedure TDDGDataSet.InternalInitFieldDefs;
begin
  // create FieldDefs which map to each field in the data record
  FieldDefs.Clear;
  TFieldDef.Create(FieldDefs, 'Name', ftString, SizeOf(TNameStr), False,
    1);
  TFieldDef.Create(FieldDefs, 'Height', ftFloat, 0, False, 2);
  TFieldDef.Create(FieldDefs, 'ShoeSize', ftInteger, 0, False, 3);
end;
```

```
procedure TDDGDataSet.InternalLast;
begin
  FRecordPos := FIndexList.Count;
end;

procedure TDDGDataSet.InternalClose;
begin
  if TFileRec(FDataFile).Mode <> 0 then
    CloseFile(FDataFile);
  FIndexList.SaveToFile(FIdxName);
  FIndexList.Clear;
  if DefaultFields then
    DestroyFields;
  FRecordPos := -1;
  FillChar(FDataFile, SizeOf(FDataFile), 0);
end;

procedure TDDGDataSet.InternalHandleException;
begin
  // standard implementation for this method:
  Application.HandleException(Self);
end;

procedure TDDGDataSet.InternalDelete;
begin
  FIndexList.Delete(FRecordPos);
  if FRecordPos >= FIndexList.Count then Dec(FRecordPos);
end;

procedure TDDGDataSet.InternalAddRecord(Buffer: Pointer;
  Append: Boolean);
var
  RecPos: Integer;
begin
  Seek(FDataFile, FileSize(FDataFile));
  BlockWrite(FDataFile, PDDGData(Buffer)^, 1);
  if Append then
  begin
    FIndexList.Add(Pointer(FileSize(FDataFile) - 1));
    InternalLast;
  end
  else begin
    if FRecordPos = -1 then RecPos := 0
    else RecPos := FRecordPos;
    FIndexList.Insert(RecPos, Pointer(FileSize(FDataFile) - 1));
```

continues

LISTING 30.5 Continued

```
  end;
  FIndexList.SaveToFile(FIdxName);
end;

procedure TDDGDataSet.InternalOpen;
var
  HFile: THandle;
begin
  // make sure table and index files exist
  FIdxName := ChangeFileExt(FTableName, feDDGIndex);
  if not (FileExists(FTableName) and FileExists(FIdxName)) then
  begin
    if MessageDlg('Table or index file not found.  Create new table?',
      mtConfirmation, [mbYes, mbNo], 0) = mrYes then
    begin
      HFile := FileCreate(FTableName);
      if HFile = INVALID_HANDLE_VALUE then
        DatabaseError('Error creating table file');
      FileClose(HFile);
      HFile := FileCreate(FIdxName);
      if HFile = INVALID_HANDLE_VALUE then
        DatabaseError('Error creating index file');
      FileClose(HFile);
    end
    else
      DatabaseError('Could not open table');
  end;
  // open data file
  FileMode := fmShareDenyNone or fmOpenReadWrite;
  AssignFile(FDataFile, FTableName);
  Reset(FDataFile);
  try
    FIndexList.LoadFromFile(FIdxName); //initialize index TList from file
    FRecordPos := -1;                  //initial record pos before BOF
    BookmarkSize := SizeOf(Integer);   //initialize bookmark size for VCL
    InternalInitFieldDefs;             //initialize FieldDef objects
    // Create TField components when no persistent fields have been
    // created
    if DefaultFields then CreateFields;
    BindFields(True);                  //bind FieldDefs to actual data
  except
    CloseFile(FDataFile);
    FillChar(FDataFile, SizeOf(FDataFile), 0);
    raise;
  end;
end;
```

```
procedure TDDGDataSet.InternalPost;
var
  RecPos, InsPos: Integer;
begin
  if FRecordPos = -1 then
    RecPos := 0
  else begin
    if State = dsEdit then RecPos := Integer(FIndexList[FRecordPos])
    else RecPos := FileSize(FDataFile);
  end;
  Seek(FDataFile, RecPos);
  BlockWrite(FDataFile, PDDGData(ActiveBuffer)^, 1);
  if State <> dsEdit then
  begin
    if FRecordPos = -1 then InsPos := 0
    else InsPos := FRecordPos;
    FIndexList.Insert(InsPos, Pointer(RecPos));
  end;
  FIndexList.SaveToFile(FIdxName);
end;

function TDDGDataSet.IsCursorOpen: Boolean;
begin
  // "Cursor" is open if data file is open.  File is open if FDataFile's
  // Mode includes the FileMode in which the file was open.
  Result := TFileRec(FDataFile).Mode <> 0;
end;

function TDDGDataSet.GetRecordCount: Integer;
begin
  Result := FIndexList.Count;
end;

function TDDGDataSet.GetRecNo: Integer;
begin
  UpdateCursorPos;
  if (FRecordPos = -1) and (RecordCount > 0) then
    Result := 1
  else
    Result := FRecordPos + 1;
end;

procedure TDDGDataSet.SetRecNo(Value: Integer);
begin
  if (Value >= 0) and (Value <= FIndexList.Count-1) then
  begin
```

continues

LISTING 30.5 Continued

```
    FRecordPos := Value - 1;
    Resync([]);
  end;
end;

procedure TDDGDataSet.SetTableName(const Value: string);
begin
  CheckInactive;
  FTableName := Value;
  if ExtractFileExt(FTableName) = '' then
    FTableName := FTableName + feDDGTable;

  FIdxName := ChangeFileExt(FTableName, feDDGIndex);

end;

procedure Register;
begin
  RegisterComponents('DDG', [TDDGDataSet]);
end;


function TDDGDataSet.GetDataFileSize: Integer;
begin
  Result := FileSize(FDataFile);
end;

procedure TDDGDataSet.EmptyTable;
var
  HFile: THandle;
begin
  Close;

  DeleteFile(FTableName);
  HFile := FileCreate(FTableName);
  FileClose(HFile);

  DeleteFile(FIdxName);
  HFile := FileCreate(FIdxName);
  FileClose(HFile);

  Open;
end;

end.
```

Summary

This chapter demonstrated how to extend your Delphi database applications to incorporate features that aren't encapsulated by VCL. Additionally, you learned some of the rules and processes for making direct calls into the BDE from Delphi applications. You also learned the specifics for extending the behavior of `TTable` with regard to dBASE and Paradox tables. Finally, you went step by step through the challenging process of creating a working `TDataSet` descendant. In the next chapter, "Internet-Enabling your Applications with WebBroker," you'll learn how to create server-side applications for the Web and deliver data to Web clients in real time.