# Digging Into IDAPI Part 1

*by John O'Connell*

The Borland Database Engine, or BDE, is the foundation on which Borland's database development products – Paradox, dBase and of course, Delphi – are all built. Applications wishing to use BDE services do so by calling BDE functions via IDAPI, the Integrated Database API, which provides functions to enable the management and control of IDAPI objects such as databases, tables and queries, to name but a few. This article aims to explore the BDE in more detail, to look at how Delphi encapsulates BDE functionality and how we can extend Delphi's database handling capabilities by using calls to IDAPI.

| IDAPI | Purpose | Delphi |
|---|---|---|
| System | Manages all BDE clients on a single machine | |
| Client | Manages the BDE sessions created and used by a BDE application | |
| Session | Manages the database, cursors and query statements created within each session | TSession |
| Driver | Provides connectivity to the data source (Paradox, dBase, Interbase, Oracle etc) for which the driver is written | |
| Database | Collects together the tables created within the database | TDatabase |
| Cursor | Provides access to records in a table or query result | TTable (TDataSet) |
| Query statement | Used by the BDE query engine to return query results from tables | TQuery |

➤ *Table 1: IDAPI objects and their Delphi equivalents*

## Programming IDAPI

A BDE application uses IDAPI function calls to create and manage the various IDAPI objects discussed in last month's *The Late Session* article. IDAPI objects are uniquely identified by a handle: any function which creates an IDAPI object will return a handle to that created object and any function which manipulates that object, or queries its properties, will take (at least) that object's handle as a parameter. Once an IDAPI object is freed, its handle is no longer valid. As far as the BDE is concerned, IDAPI objects are freed when closed. Table 1 lists the IDAPI objects and the applicable Delphi equivalents.

The core of Delphi's use of IDAPI to give us all those wonderful, easy to use data-access components is defined in the DB unit which is used by any Delphi application using the data-access VCL components. Let's check out what happens in the DB unit (and indeed, in any BDE application) when it initialises at run-time. First the BDE is initialised by a call to `DbiInit` which opens an IDAPI system object (if one doesn't already exist) and an IDAPI client object, which in turn opens a default session object. `DbiInit` takes a single parameter: a pointer to a record type which specifies the application's working directory, the BDE configuration file from which IDAPI should obtain configuration details such as driver properties, aliases etc to use, and various other details which we're not really concerned with for the purposes of this article. Passing a `nil` pointer to `DbiInit` causes IDAPI to use the default configuration file as specified in WIN.INI for Delphi 1.0 and in the Registry for Delphi 2.0.

As soon as the BDE has been initialised, the application can open databases and tables within the default session, or open new sessions within which databases and tables can be opened. Note that an application can initialise the BDE only once: successive calls to `DbiInit` will fail. A terminating application must call `DbiExit` to close the client which disconnects the application from IDAPI. `DbiInit` can be called as often as desired provided that the existing client is closed using `DbiExit`. Looking at the DB unit's initialisation section in Delphi 1.0 and 2.0, we see that the default session, encapsulated by the `TSession` class and instantiated as the `Session` object variable, is created by a call to `DbiInit`.

Now the application is ready to do all the usual stuff such as opening databases and tables, performing queries and all the rest. More specifically, a Delphi application is now ready to open `TDatabases` and `TTables`, and execute `TQuery`, `TBatchMove` and `TStoredProc`. Obviously `TDatabase` encapsulates a BDE database handle but a BDE cursor is actually encapsulated by `TDataSet` from which `TTable`, `TQuery`, `TStoredProc` and `TBatchMove` are descended. `TQuery` encapsulates an IDAPI query statement handle and a cursor handle for the query result set. The property `TQuery.StmtHandle` is initialised when the query is prepared, `TQuery.Handle` is initialised when the executed query statement returns a result set.

Using IDAPI to open a database is done by calling `DbiOpenDatabase` which takes an alias name as its first parameter and returns a handle to the database (of type `hDbiDb`) on success. `DbiOpenDatabase` takes other parameters but I'm only going to discuss those parameters that are relevant to the discussion of the particular IDAPI function. A table is opened using `DbiOpenTable` which takes a database handle and

a table name among its parameters and returns a cursor handle of type hDbiCur. The fact that opening a table requires an open database may seem strange to us Delphi developers because a TDatabase component isn't required to open a TTable. In fact the TTable, TQuery and TStoredProc components each have a temporary Database property of type TDatabase which is opened and closed just before and after the TTable, TQuery or TStoredProc is opened and closed. TDatabase has a Temporary run-time property which is True if the database instance will be freed after the table is closed. I'll leave you to guess the IDAPI functions used to close databases and tables.

It's important to note that a Dbi function (you'll have guessed by now that all IDAPI function names start with Dbi so I'll refer to them as such) returns data to the caller through the use of client allocated pointers passed as parameters: in Object Pascal this could be either a var parameter or a pointer to a client-allocated variable. The return value indicates success or the reason for the call's failure.

The use of Dbi functions in Delphi applications requires little effort. For Delphi 1.0, Dbi functions are declared in the DBIPROCS.INT unit interface file, IDAPI data-types are defined in DBITYPES.INT and IDAPI error codes in DBIERRS.INT, all residing in the DELPHI\DOC directory. Delphi 2.0 simplifies matters by providing everything in one file named BDE.INT in the directory DELPHI\DOC. To use IDAPI calls in your code just include the relevant unit(s) in your unit's uses statement. If you're writing code to be portable between Delphi 1.0 and 2.0 it's easier to declare the required Dbi units as for Delphi 1.0 because Delphi 2.0 uses defined unit aliases in order to resolve references to DBIPROCS, DBIERRS and DBITYPES to the BDE unit.

Let's look at how IDAPI does the things normally done within a database application and more.

## What's The Status?

IDAPI provides functions to query the BDE system status, return BDE

```
SYSInfo = record
  iBufferSpace    : Word;  { in K }
  iHeapSpace      : Word;  { in K }
  iDrivers        : Word;  { Active/Loaded drivers }
  iClients        : Word;  { Active clients }
  iSessions       : Word;  { Number of sessions (for all clients) }
  iDatabases      : Word;  { Open databases }
  iCursors        : Word;  { Number of cursors }
end;
```

➤ *Listing 1*

```
SYSVersion = record
  iVersion        : Word;  { Engine version }
  iIntfLevel      : Word;  { Client Interface level }
  dateVer         : Date;  { Version date (Compile/Release) }
  timeVer         : Time;  { Version time (Compile/Release) }
end;
```

➤ *Listing 2*

```
SYSConfig = record
  bLocalShare     : Bool;          { If Local files will be shared }
  iNetProtocol    : Word;          { Net Protocol (35, 40 etc.) }
  bNetShare       : Bool;          { True if connected to network }
  szNetType       : DBINAME;       { Network type }
  szUserName      : DBIUSERNAME;   { Network user name }
  szIniFile       : DBIPATH;       { Configuration file }
  szLangDriver    : DBINAME;       { System language driver }
end;
```

➤ *Listing 3*

engine version and system configuration information. This type of information can be useful when things are going wrong with your database application and you need to know exactly what version and build of the BDE you're running, how many cursors are open etc.

DbiGetSysInfo retrieves system status information and takes a SYSInfo record as a var parameter (Listing 1).

System status information encompasses all BDE applications loaded in the system. The active/loaded drivers referred to are the database drivers (Paradox, dBase etc) loaded for use by IDAPI.

DbiGetSysVersion retrieves the database engine version and takes a SYSVersion record as a var parameter (Listing 2).

The actual engine version is determined from iVersion/100. The Date and Time types are basically long integers containing encoded date and time information. DbiGetSysConfig retrieves BDE configuration information and takes a SYSConfig record as a var parameter (Listing 3).

Some of what's described in SYSConfig can be set via the System page of the BDE Configuration Utility. The setting for Local Share determines whether local files will be shared with non-IDAPI applications. Such a situation would occur where an application's tables are shared with Paradox for DOS or any other Paradox Engine application. The network user name can also be retrieved using DbiGetNetUsername which takes a pointer to a null-terminated string as a parameter in which the user name is returned.

The IDAPINFO application (Figure 1, the code is on this month's disk) uses these functions to display status, version and configuration information in a dialog and can be used to help understand how the BDE/IDAPI works. IDAPINFO allows you to refresh the information retrieved and specify whether or not IDAPI will be initialised for the application (ie is an IDAPI client created). Let's use the application to see what goes on in the world of IDAPI.

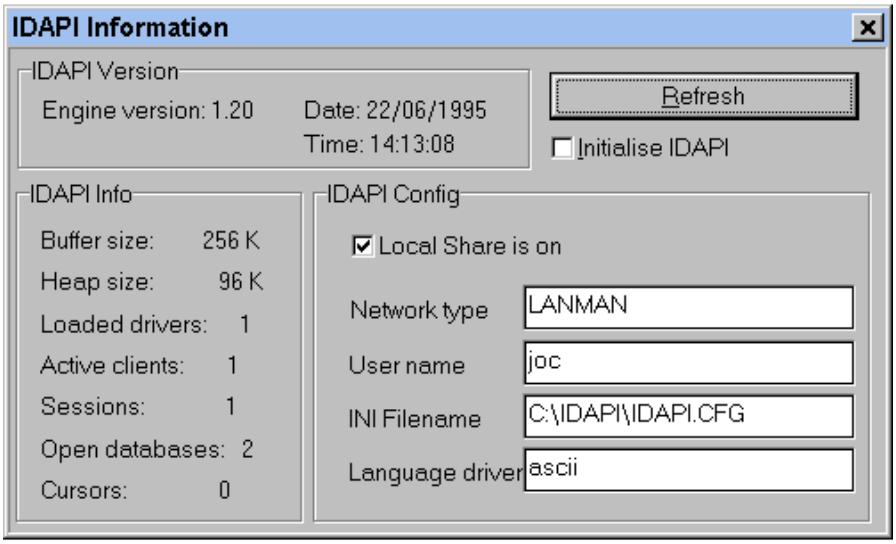
First, boot or restart Windows. Load Delphi 1.0 and run the IDAPINFO project. The version and configuration information displayed requires no explanation but

notice that Active clients, Session and Open databases all have counts of 1, despite the fact that no other BDE applications are loaded and IDAPINFO hasn't initialised IDAPI to register itself as a client. Well, there is another BDE application running – Delphi – which has initialised the BDE on its own behalf; after all, how else would Delphi be able to display records from a TTable or TQuery opened at design time? Now check the Initialise IDAPI check-box and refresh the display. Now the counts have been incremented because IDAPINFO has called DbiInit and created a new client, which created a default session and, it appears, a default database.

Next load Database Desktop and refresh IDAPINFO. Notice that the client, session and database counts rise to three and Loaded drivers becomes 1: Database Desktop loads a table driver on startup. From within DBD open a few Paradox tables in the same database and notice the cursors count increase. Opening tables from a different database will increase the cursors count *and* open databases count. Opening a dBase or SQL table increases the loaded drivers count, which decreases when the SQL table is closed but doesn't decrease any further when the dBase table is closed. It would seem that the Paradox and dBase drivers remain loaded even when not needed. You might have also noticed that the heap size grows and shrinks as tables are opened and closed. The buffer size never shrinks below 256Kb because that is the minimum buffer size specified (MINBUFSIZE) in the BDE configuration file.

Running IDAPINFO as the sole BDE client application reveals that no buffer or heap have been allocated, no drivers are loaded and no BDE clients, sessions, database or cursors have been opened. Now check Initialise IDAPI and refresh the information: both buffer and heap have been allocated, the clients, sessions and database counts are all one.

Note that changes to the 32-bit BDE (version 3.0) mean that




➤ *Figure 1: The IDAPINFO application*

```
procedure Check(Code: DBIRESULT);
var szError: array[0..DBIMAXMSGLEN] of char;
begin
  if Code <> DBIERR_NONE then begin
     DbiGetErrorString(Code, szError);
     raise Exception.Create(StrPas(szError));
  end;
end;
```

➤ *Listing 4*

IDAPINFO cannot be run with Initialise IDAPI not checked if compiled and run under Delphi 2.0 because DbiGetSysVersion, DbiGetSysInfo and DbiGetSysConfig all require that the BDE be initialised. This isn't the case with the 16-bit version of the BDE shipped with Delphi 1.0, so it seems Borland have tightened things up a little.

I've used a few other Dbi functions in IDAPINFO. DbiDateDecode and DbiTimeDecode decode the binary date and time retrieved by DbiGetSysVersion.

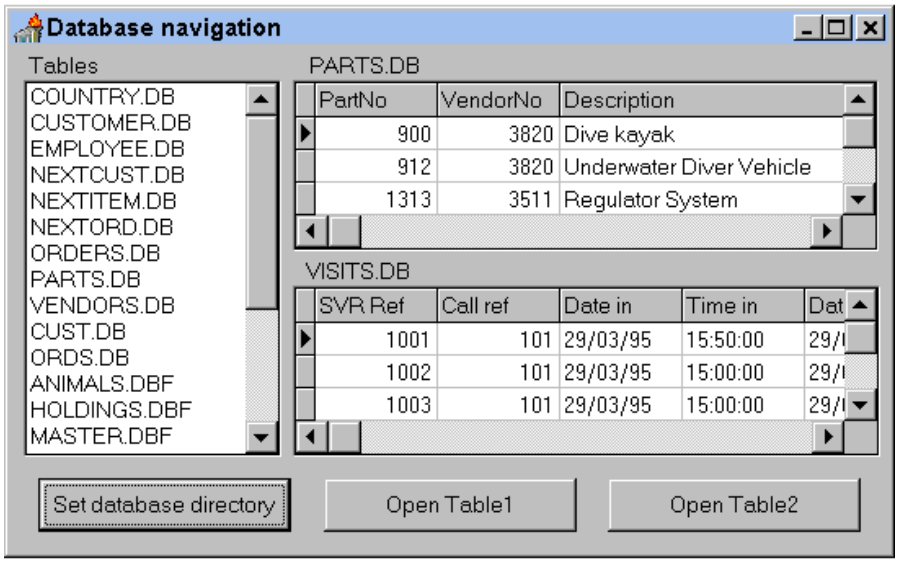
You may have noticed that all Dbi function calls are enclosed in a call to the Check procedure defined in the form unit (Listing 4). This simplifies the handling of non-success (ie non-zero) return codes from Dbi functions. In fact Borland have provided a similar procedure (which does a bit more) in the DB unit which is used with almost every call to IDAPI throughout the VCL.

## Cheating With Databases

Access to a table is gained via an open database of which there are generally two types: the standard database and an SQL database. A standard (or local) database is defined by a disk directory containing the tables managed by that database whose parameters identify the directory path. An SQL database is defined by the SQL database server which identifies the logical location of the database, be it a local or remote database server. It is possible to change the working directory of an open local database at run-time, which makes it possible to cheat a little by concurrently opening tables in different directories using a single TDatabase component. Delphi doesn't provide any way of doing this, but a call to DbiSetDirectory does the trick:

```
Check(DbiSetDirectory(
  Database1.Handle, szDir));
```

where Database1 is a non temporary open database and szDir is a null-terminated string. Passing nil as the second parameter changes the working directory back to the default path for the specified




➤ *Figure 2: The SETWKDIR application*

```
function DbiSetToSeqNo( { Position to a logical record number }
  hCursor : hDBICur;    { Cursor handle }
  iSeqNo  : Longint     { Sequence number }
  ): DBIResult;
```

➤ *Listing 5*

```
function DbiSetToRecordNo( { Position to Physical Rec# }
  hCursor : hDBICur;       { Cursor handle }
  iRecNo  : Longint        { Physical record number }
  ): DBIResult;
```

➤ *Listing 6*

database. Another directory-related function is `DbiSetPrivateDir` which sets the private directory for the current session to that specified by a null-terminated string parameter; passing `nil` resets the private directory to the default startup directory. However, the `TSession`'s `PrivateDir` property provides the same functionality as this function. The demo application SETWKDIR on this month's disk (see Figure 2) demonstrates the use of `DbiSetDirectory` which somewhat violates the standard concept of a database!

## Table Navigation

Obviously, any database engine must provide table navigation functions. Delphi provides most of the table navigation methods you'd ever need, though there are a few things missing. It's not possible to move to a particular record position within a table (Paradox and dBase both provide the means to do so) and it's not possible to retrieve the current record number for a local table. IDAPI provides functions to achieve all of these missing Delphi features but for some reason Borland chose not to implement them. Let's set about putting that right by discussing the relevant `Dbi` functions.

Both `DbiSetToSeqNo` and `DbiSetToRecordNo` move a cursor's record pointer to a specified position, but the choice of which one to use depends on the table type. Only Paradox tables support record sequence numbers and only dBase tables support record numbers, but what is the difference between them? A sequence number is a logical record position: the position of a record within the current view, which may be filtered or indexed, of the table. For instance, with no active secondary index a record's sequence number is 56; a secondary index is then activated which changes that record's sequence number to 10! Also, if a record is inserted before record 56 then that record's sequence number will change to 57 – that gives potential for major trouble if the table is shared and the application is using sequence numbers to keep track of record positions. Obviously sequence numbers are pretty much unsuitable for keeping track of a particular record, but then what are bookmarks for? Use them! dBase record numbers, on the other hand, are much more reliable as an indicator of a record's actual position within the table, because the record number is a physical record number which remains constant regardless of what indexes are applied to the table; even inserting or deleting records will not affect a dBase record's record number. The only time the record number will be changed is after the table has been packed, but that isn't a problem because no one else can have access to the table whilst it's being packed.

So on to `DbiSetToSeqNo` (Listing 5). I don't recommend relying on it to move to a previous record position unless you're certain that the table hasn't changed since in a way which could have changed the record sequence.

This function is easy enough to follow, but bear in mind that if a call to this function attempts to move the record pointer to a sequence number outside the range of records in the table, an error is returned in `DBIResult` to indicate that the start or end of the table (`DBIERR_BOF` or `DBIERR_EOF`) has been reached. The same applies to `DbiSetToRecordNo` (Listing 6).

Whilst we're on the subject of table navigation, let's take a look at how records in a table are seen by IDAPI. Records are conceptually delimited by 'cracks' (see Figure 3) on which a cursor can be positioned.

Whilst this may seem odd it does make life easier for the IDAPI programmer because a table scan can be achieved by moving to the beginning of the table (BOF) after which `DbiGetNextRecord` can be

called repeatedly until DBIERR_EOF is returned after the last record has been read and the cursor pointer tries to advance past the end of the table. Cracks are the reason why, in a Delphi application using a TDBNavigator, it is possible to move to the first record in a table and then still move back a record at which point the 'first record' navigator button becomes disabled because you've hit the BOF crack; pressing the 'refresh' navigator button re-enables the 'first record' pushbutton because DbiGetNextRecord was called.

So what about getting the sequence number or record number of the current record of a Paradox or dBase table? The function DbiGetSeqNo retrieves a record's sequence number but there's no equivalent to retrieve a record number. Ho hum. Let's have a look at DbiGetSeqNo (Listing 7), which returns an error of DBIERR_BOF, DBIERR_EOF or DBIERR_NOCURRREC if the record pointer isn't positioned on a record but on a crack.

What about getting a dBase table's current record number? That involves a little more work and an introduction to IDAPI properties. Remember that the BDE is object-oriented in design and we all know that objects have properties. To find a record's number we need to examine, you've guessed it, the record's properties, which we retrieve by calling DbiGetRecord (Listing 8) where DBILockType is defined as:

```
DBILockType = (dbiNOLOCK,
  dbiWRITELOCK, dbiREADLOCK);
```

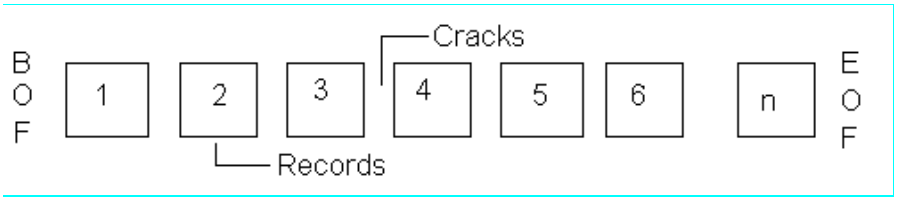
and precProps is defined as shown in Listing 9. I recommend passing eLock as dbiNOLOCK, but a word of caution: if the dataset is in edit/insert mode the current record will be locked but a call to DbiGetRecord with eLock passed as dbiNOLOCK may have undesirable side-effects. You have been warned!

Because we're not interested in actually reading the record we can pass pRecBuff as a nil pointer. We can retrieve the sequence number or record number from RECProps.iSeqNum or RECProps.iPhyRecNum respectively. To determine which is applicable is a simple matter of checking the TTable's TableType property for ttParadox or ttDbase. If it's ttDefault we then need to check the table type (.DB or .DBF) of the TableName property.

But there's a simpler and better way of determining whether you should be concerned with sequence numbers or physical record numbers when dealing with a table: examine the table's cursor properties.

DbiGetCursorProps retrieves a specified cursor's properties and is declared as shown in Listing 10. Rather than discuss cursor properties in detail (I'll leave that until next month) I'll concentrate on just one field of the CURProps record: iSeqNums of type Integer, which is 1 if the cursor uses sequence numbers and 0 if it uses physical record numbers.

This way of determining which record position identifier to use is by far the most reliable, because as new IDAPI table drivers are released they'll have to implement cursor properties. If we had relied upon the fact that tables with a .DB extension support sequence numbers and .DBF tables support physical record numbers, we'd have more work to do to support .COFFEE tables (produced by Java applications) which might be supported by potential future IDAPI drivers!




➤ *Figure 3:Cracks in the IDAPI record layout*

```
function DbiGetSeqNo (  { Get logical record number }
  hCursor    : hDBICur; { Cursor handle }
  var iSeqNo : Longint  { Pointer to sequence number }
  ): DBIResult;
```

➤ *Listing 7*

```
function DbiGetRecord (    { Gets the current record }
  hCursor   : hDBICur;     { Cursor handle }
  eLock     : DBILockType; { Optional lock request }
  pRecBuff  : Pointer;     { Record buffer(client) }
  precProps : pRECProps    { Optional record properties }
  ): DBIResult;
```

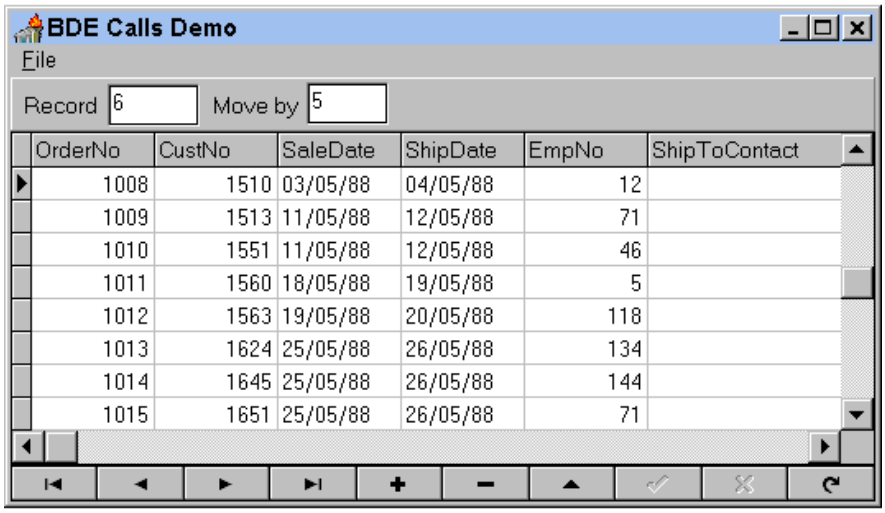
➤ *Listing 8*

```
RECProps = record              { Record properties }
  iSeqNum         : Longint;   { When Seq# supported only }
  iPhyRecNum      : Longint;   { When Phy Rec#s supported only }
  bRecChanged     : Bool;      { Not used }
  bSeqNumChanged  : Bool;      { Not used }
  bDeleteFlag     : Bool;      { When soft delete supported only }
end;
```

➤ *Listing 9*

```
function DbiGetCursorProps ( { Get Cursor properties }
  hCursor      : hDBICur;    { Cursor handle }
  var curProps : CURProps    { Cursor properties }
  ): DBIResult;
```

➤ *Listing 10*




➤ *Figure 4: The CALLBDE application*

```
function GetRecNo(ATable: TTable): LongInt;
var Props:  CURProps;
    RProps: RECProps;
begin
  Result := -1;
  Check(DbiGetCursorProps(ATable.Handle, Props));
  ATable.UpdateCursorPos;
  Check(DbiGetRecord(ATable.Handle, dbiNOLOCK, nil, @RProps));
  if (Props.iSeqNums = 1) then
    Result :=  RProps.iSeqNum
  else
    if (Props.iSeqNums = 0) then
      Result := RProps.iPhyRecNum
end;
```

➤ *Listing 11*

```
function DbiGetRelativeRecord ( { Find/Get a record by record number }
  hCursor    : hDBICur;     { Cursor handle }
  iPosOffset : Longint;     { offset from current position }
  eLock      : DBILockType; { Optional lock request }
  pRecBuff   : Pointer;     { Record buffer(client) }
  precProps  : pRECProps    { Optional record properties }
): DBIResult;
```

➤ *Listing 12*

Listing 11 shows a function to retrieve a `TTable`'s current record number.

The good news for Delphi 2.0 developers is that `TTable` has a new `RecNo` property. By the way, the value of `iSeqNums` can also be used to decide whether `DbiSetToSeqNo` or `DbiSetToRecordNo` should be used to move to a particular record in a local table.

One thing I've forgotten to mention is the `UpdateCursorPos` method of `TTable` which must be called before using any `Dbi` functions which rely on the `TTable`'s current underlying cursor position, such as `DbiGetSeqNo` or `DbiGetRecord` – this must be done because a `TDataSet` (of which `TTable` is an ancestor) buffers records retrieved from the cursor and therefore the `TTable` record position and underlying cursor position might not be in synch. The VCL help states that `TTable.CursorPosChanged` must be called after calling a `Dbi` function which alters the `TTable`'s underlying cursor position but to be honest I've had no joy with that – calling `TTable.Refresh` seems to do the job of re-synchronising the `TTable`'s record pointer with the underlying cursor.

The CALLBDE demo application on the disk (Figure 4) demonstrates the use of the `Dbi` functions discussed so far.

I'll introduce another useful `Dbi` function whose functionality is already provided by the `TTable.MoveBy` method but in which there's a subtle bug with Delphi 1.0. `MoveBy` moves a table's record position by a specified offset (which may be negative) but because this parameter is an integer we're limited to moving no further than 32767 records in either direction from the current position which can be a bit limiting with very large tables. In Delphi 2.0 an integer is 32 bits so there's no such problem with `MoveBy`.

The function `DbiGetRelativeRecord` moves the cursor position by a specified offset (Listing 12). Notice that `iPosOffset` is a `LongInt` which means we can move by 2147483647 records in either direction. Don't forget to call `UpdateCursorPos` before calling this function and `Refresh` afterwards. Strangely, the implementation of `TTable.MoveBy` doesn't make any calls to `DbiGetRelativeRecord`.

## TDBRecLabel

With all this new-found IDAPI knowledge let's create a data-aware control component to display the current sequence/record number of the dataset associated with a particular datasource. The record number can be displayed as `Record 1` or `Record 1 of 500` for example.

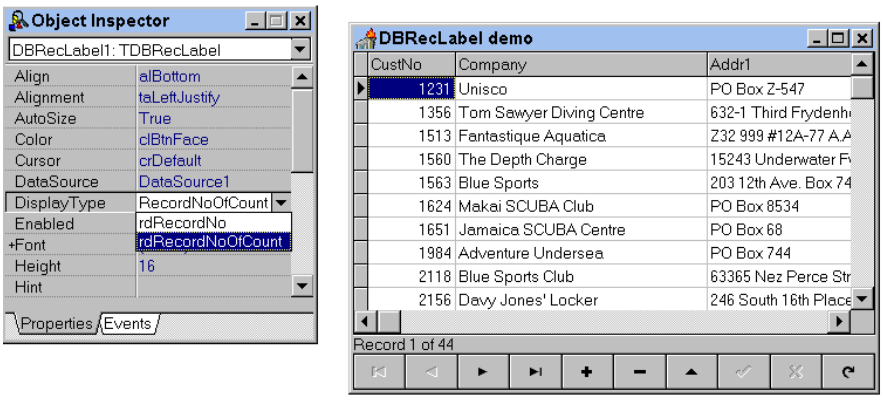
As with any data-aware component we need to declare a private `TFieldDataLink` member which will be linked to a particular `TDataSource` component. As for which component `TDBRecLabel` should be descended we need to choose a `TLabel`-type component which doesn't surface a `Caption` property.

Looking at `TLabel`'s descendant we find `TCustomLabel` has everything we need as the basis for `TDBRecLabel` and it doesn't surface the protected `Caption` property. In fact the VCL provides a number of `TCustomXXXXX` components which surface few or no public or published properties and therefore provide a good foundation for deriving new components, which is exactly the reason Borland included them!

TDBRecLabel doesn't surface the Caption property but uses it internally; however, all other properties surfaced by TLabel have been surfaced by TDBRecLabel. The private TFieldDataLink object FDataLink provides a few useful event handlers which are initialised in the constructor. OnDataChange is triggered for the same reasons as TDataSource.OnDataChange – when the current record in the dataset changes or refreshes for various reasons. OnActiveChange is triggered whenever the active state of the dataset associated with our datalink's DataSource property changes (ie when the dataset is opened or closed) but it's important to note that if the dataset associated with DataSource.DataSet changes from one open dataset to another open dataset, OnActiveChange will not be triggered because the active state of the datasource's dataset property hasn't actually changed.

FDataLink's DataSource property is set via the published DataSource property which has been added to this component's class: see the GetDataSource and SetDataSource property access methods. Note that SetDataSource also retrieves the cursor properties of the datasource's dataset which are used to set the private FSeqNoCapaable and FRecNoCapable fields. The DisplayType property is also an additional property and can be set to rdRecordNo or rdRecordNoOfCount which alters the record number display format as shown in Figure 5.

One small problem with TDBRecLabel concerns painting itself when not connected to an active datasource/dataset in the IDE's form designer. The same problem exists with the TDBText data control component, in that the component's caption text is not displayed when its associated datasource/dataset is inactive when the form is loaded. This occurs because the Caption property is not published and hence isn't stored in the form file. However, once the dataset is opened the caption does get displayed.




➤ *Figure 5: TDBRecLabel in action*

It's worth mentioning the Notification method which basically receives component insert and remove messages from the IDE. Notification is a virtual method of TComponent which receives a pointer to a component and the operation (of type TOperation which is defined in CONTROL.PAS) which can be opInsert or opRemove. This method is important for components having properties which are pointers to other components. If an appropriate Notification method is not defined for a particular component then that component's component property may become a dangling pointer when the referenced component is removed or freed.

However, I can reveal that the Notification method defined for TDBRecLabel is actually unnecessary because a TDataSource's destructor sets all of its associated datalink's datasource properties to nil, but including the method does illustrate an important point regarding component implementation – if our component property wasn't of type TDataSource then the Notification method would be essential. Whenever you override the Notification method be sure to call the inherited method before doing anything else!

## Until Next Month...

For those of you wanting to find out more about IDAPI function calls Borland have published some sources of information.

The *IDAPI Function Reference* is a Windows help file (IDAPI.HLP for BDE 2.0) available for download from CompuServe and from Borland's own web site (which is at www.borland.com) which details all the functions provided by IDAPI. Delphi 2.0 Developer edition includes BDE32.HLP for BDE 3.0 which details all IDAPI functions and lists the new functions introduced in the 32-bit BDE. Whilst these help files are a valuable resource it's a good idea to get a copy of the *BDE User's Guide* if you're really serious about BDE programming as it discusses fundamental BDE/IDAPI concepts which aren't really covered in the help files, although I've covered a fair few of these in this and the previous article. The *Guide* can be difficult to get hold of: in the USA you can get it direct from Borland, but as far as we know other Borland offices don't stock it.Try your local User Group.

Next month we'll move on to discuss bookmarks, cursor properties and introduce some more useful IDAPI functions and structures which open up even up more possibilities for the Delphi database developer.

John O'Connell is a freelance software consultant and developer specialising in Delphi and database application development. He can be reached via email on 73064.74@compuserve.com