Digging Into IDAPI Part 3

by John O'Connell

Last month I promised we'd have a look at table and record locking, as well as in-memory tables and their uses. We'll also continue developing the TjocTable component and a few more components which give us some useful information about BDE objects.

Lock Me Up Before You Go-Go...

Table and record locks are pretty much left out in the cold as far as Delphi's documentation is concerned. The exclusive table lock does get a mention as implemented in the TTable.Exclusive property. In the VCL source you'll find the undocumented TTable methods LockTable, which applies a read lock or a write lock to a table, and UnlockTable, which releases a read or write lock. We encountered these methods last month.

Let's compare the types of table locks we know about so far. The exclusive table lock prevents any other users from opening the table: obviously it's impossible to apply an exclusive lock to a table already opened by another session. Cases where an exclusive lock is necessary are when a table is being restructured, packed and renamed, in which case the most restrictive type of table lock is required.

A read lock prevents all other users modifying the table. This is more commonly known as a "repeatable read" which can be placed on a table by any number of users. A write lock prevents all other users except the lock owner from modifying the table: only one user at a time can place a write lock on a table which cannot have been read locked any other users. However, where allowed, a user can place a read lock on a table and then place a write lock without needing to release the read lock: when the write lock is released the read lock will still be in effect. Table 1 summarises the relationship between the various lock types.

User A applies:	User B applies:			
	Exclusive lock	Read lock	Write lock	No table lock
Exclusive lock	Fails	Fails	Fails	Fails
Read lock	Fails	Succeeds[1]	Fails	Succeeds
Write lock	Fails	Fails	Fails	Succeeds
No table lock	Fails	Succeeds	Succeeds	Succeeds

[1] *Succeeds if Paradox table, fails if dBase table*

➤ *Table 1: Table locks interaction*

We can see that the table read lock is the least restrictive and the exclusive lock is the most restrictive. Record locks aren't even half as varied as table locks (a record is either write locked or not locked at all), there is no such thing as a read record lock for local tables. Further, if any of these table locks are in effect no record in the table can be locked. So how are record locks implemented in Delphi? When a dataset's state switches to dsEdit the current record is locked. There's no other way of locking a record unless a BDE function is used. Remember DbiGetRecord? The function does a number of things: it retrieves the current record, optionally locks that record and optionally retrieves that record's properties. Here's the function prototype again:

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

where DBILockType is defined as:

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

Now those of you who are paying attention will cry "I thought you said there was no such thing as a read record lock?" Well, I wasn't quite straight with you. A read record lock is always promoted to a write record lock (unless the table driver says otherwise, which for Paradox and dBase is currently not the case). To lock the current record in Table1 we use:

```
Table1.UpdateCursorPos;
Check(DbiGetRecord(
  Table1.Handle, dbiWriteLock,
  nil, nil));
```

The first statement synchronises the record pointer, the second just locks the record: we don't want the record data or properties. With Delphi, Database Desktop and Paradox, a record lock is released when the record is posted or cancelled, but a record locked using DbiGetRecord stays locked until the table is closed or the function DbiReleaseRecordLock is called:

```
function DbiRelRecordLock(
  { Cursor handle }
  hCursor     : hDBICur;
  { True for all locks }
  bAll        : Bool
  ): DBIResult;
```

If the bAll parameter is True the function will release all record locks placed on the table identified by hCursor by the current session. If bAll is False only the current record is unlocked: this function fails if the current record is not locked. We can check if the current record is locked by using:

```
function DbiIsRecordLocked(
  { Cursor handle }
  hCursor     : hDBICur;
  { Lock status }
  var bLocked : Bool
  ): DBIResult;
```

The record's lock status is returned in bLocked. What about table locks? How can we tell if a table is locked? IDAPI provides:

```
function DbiIsTableLocked(
  { Cursor handle }
  hCursor    : hDBICur;
  { Lock type to verify }
  epdxLock   : DBILockType;
  { Num of locks of given type}
  var iLocks : Word
  ): DBIResult;
```

which returns the number of specified locks on the table (identified by hCursor) in the iLocks parameter. We can determine if a table is shared by other users by calling:

```
function DbiIsTableShared(
  { Cursor handle }
  hCursor     : hDBICur;
  { Shared status }
  var bShared : Bool
   ): DBIResult;
```

which will return False in bShared if the table has an exclusive lock or is not shared.

We can count the number of cursors open on a table with a call to:

```
function DbiGetTableOpenCount(
  { Database }
  hDb             : hDBIDb;
  { Table name }
  pszTableName    : PChar;
  { Driver type }
  pszDriverType   : PChar;
  { returned number of cursors}
  var iOpenCount : Word
   ): DBIResult;
```

where the number of cursors open is returned in iOpenCount. If pszTableName is passed with a file extension or if hDb identifies an SQL database, pszDriverType can be passed as nil, otherwise either szPARADOX or szDBASE must be passed. This function can be used before attempting to open a table for exclusive use. Notice that a cursor handle isn't required so the relevant table doesn't have to be open when we call this function.

All the IDAPI functions mentioned here so far have been implemented in TjocTable (the full source is included on this month's disk) as the CountTableLocks, LockRecord and UnlockRecord methods and as the IsRecordLocked, IsShared and OpenCount properties.

A useful locks-related function is DbiSetLockRetry which allows you to set a lock retry period for the current session (important note for Delphi 2.0 developers) during which a failed lock is retried until the retry period expires. The prototype for this function is:

```
function DbiSetLockRetry(
  iWait: Integer): DBIResult;
```

If iWait is zero then no retry is attempted, a negative iWait causes infinite retries and a positive value specifies the retry period in seconds. Lock retries only occur if the table is open; if the table is closed, any failed attempt to open it exclusively will not be retried. If you decide to use lock retries bear in mind that your application may run sluggishly if you set the retry period too high where there's a good chance that a record to be edited will be locked. A sensible maximum figure is five seconds. After all, you don't want your application's users thinking that their system has hung just because you've set an excessive lock retry period.

We can retrieve all the information we want about all types of locks on a particular table using the IDAPI function:

```
function DbiOpenLockList(
  { Cursor handle }
  hCursor        : hDBICur;
  { True, for all Users locks }
  bAllUsers      : Bool;
  { True, for all lock types }
  bAllLockTypes : Bool;
  { Returned cursor on Lock list }
  var hLocks     : hDBICur
  ): DBIResult;
```

which returns a cursor, in hLocks, to an in-memory table named LOCKS containing the required information, where hCursor identifies the table whose locks we're interested in, bAllUsers specifies whether we want locks for all users/sessions or the current session (again Delphi 2.0 developers take note) and bAllLockTypes specifies whether we want all types of locks or just record locks.

Property name	Value
szTableType	'INMEMORY'
eOpenMode	dbiReadOnly
bBookMarkStable	True
eShareMode	dbiOPENSHARED
iSeqNums	1 (uses sequence numbers)
exltMode	xltRECORD

➤ *Table 2: Notable in-memory cursor properties*

In-Memory Of...

An in-memory table exists purely in memory. There are a few other differences between an in-memory table and a standard table. Let's examine some of the more telling cursor properties listed in Table 2. The table type comes as no surprise but the share mode does – how can an in-memory table be shared? The translation mode xltRecord is new as we've only encountered translation types xltNone and xltField. We'll examine cursor translation modes a bit later. The fact that the open mode is read-only means that records can't be inserted or deleted. The most important difference between in-memory and standard tables is that once closed, an in-memory table no longer exists.

For the structure of the LOCKS table we can refer to the lock descriptor of type LOCKDesc as defined in DBITYPES (see Listing 1).

The constant lckGROUPLOCK refers to Paradox for DOS group locks. lckIMGAREA refers to Paradox image locks, which simply indicate that Paradox or Database Desktop has opened the table for view. lckTABLEREG is a table open lock which isn't really a lock but indicates that a cursor is open on the table; by counting these locks you could determine how many users have the table open instead of using DbiGetTableOpenCount. The net level session number refers to the network session identifier, the IDAPI session number refers to the local session identifier and will be zero for any locks not owned by the current session. The iInfo field doesn't concern us.

How do we get at the information in an in-memory table? Using IDAPI functions of course! We retrieve records using DbiGetNextRecord, DbiGetRecord, DbiGetPriorRecord, DbiSetToBegin and DbiSetToEnd. The last two functions position the cursor to the start and end of the table respectively. The code in Listing 2 opens and navigates the LOCKS in-memory table. The LOCKLIST.DPR application on the disk lists the contents of the LOCKS table in a grid.

There is a more convenient way of getting at the contents of the LOCKS in-memory table: we can use a TDataset derived component to encapsulate the LOCKS table, which can then be used in the IDE just like a TTable. The key to achieving this is to override the CreateHandle method of TDataset or any of its descendants. This method, called when the dataset is opened or its Active property is set to True, supplies the IDAPI cursor handle encapsulated by the dataset. We'll use TDataset as the basis for our new component TTableLocks; see the file INFOTAB.PAS on the disk for the full source and Listing 3 for the component definition.

You may be wondering why TTableLocks publishes a DataSource property rather than a Table or DataSet property of type TTable. The main reason is that the DataSource property can be assigned to the DataSource property of TTableLocks' own private FDataLink field which has events to handle changes in the state of the datasource and changes in the Active property of the datasource's dataset. All of which means that TTableLocks will know if the dataset has been closed in which case TTableLocks is closed. The published DataSource property's write access method checks the dataset to ensure it's a TTable or a TTable

➤ *Listing 1*

```
LOCKDesc = record
  iType          : Word;          { Lock type (0 for rec lock) }
  szUserName     : DBIUSERNAME;   { Lock owner }
  iNetSession    : Word;          { Net level Session number }
  iSession       : Word;          { Idapi session#, if our lock }
  iRecNum        : Longint;       { If a record lock }
  iInfo          : Word;          { Info for table locks }
end;

{ iType can take on one of the following values: }
lckRECLOCK        = 0;            { Normal Record lock (Write) }
lckRRECLOCK       = 1;            { Special Pdox Record lock (Read) }
lckGROUPLOCK      = 2;            { Pdox Group lock }
lckIMGAREA        = 3;            { Pdox Image area }
lckTABLEREG       = 4;            { Table registration/Open }
lckTABLEREAD      = 5;            { Table Read lock }
lckTABLEWRITE     = 6;            { Table Write lock }
lckTABLEEXCL      = 7;            { Table Exclusive lock }
lckUNKNOWN        = 9;            { Unknown lock }
```

➤ *Listing 2: Opening and navigating the LOCKS in-memory table*

```
const
  LockStr: array[0..9] of string[30] =
    ('Record lock (write)', 'Record lock (read)',
     'Paradox Group lock',  'Paradox Image lock',
     'Table open lock',     'Table read lock',
     'Table write lock',    'Exclusive lock',
     'No such lock',        'Unknown lock');
var
  LckDesc:  LOCKDesc;
  LckCur:   HDbiCur;
  Msg:  String;
  Table1:   TTable;
  UserName: string;
begin
  Table1 := TTable.Create(Self); {need to set properties and open the table}
  ...
  Check(DbiOpenLockList(Table1.Handle, True, True, LckCur));
  Check(DbiSetToBegin(LckCur);
  while (DbiGetNextRecord(LckCur, dbiNOLOCK, @LckDesc, nil)=DBIERR_NONE) do
    with LckDesc do begin
      NativeToAnsi(Table1.Locale, szUserName, Username);
      Msg := LockStr[iType] + #10;
      Msg := Msg + Username + #10;
      Msg := Msg + IntToStr(iNetSession) + #10;
      Msg := Msg + IntToStr(iSession) + #10;
      Msg := Msg + IntToStr(iRecNum);
      if MessageDlg(Msg, mtConfirmation, mbOkCancel, 0) = mrCancel then
        break;
    end;
  Check(DbiCloseCursor(LckCur));
  ...
end;
```

➤ *Listing 3: TTableLocks definition*

```
TTableLocks = class(TDataset)
private
  FAllUsers: Boolean;
  FAllLockTypes: Boolean;
  FDataLink: TFieldDataLink;
  procedure SetDataSource(const Value: TDataSource);
  function GetDataSource: TDataSource;
  function CanOpenLockList: Boolean;
  procedure DoActiveChanged(Sender: TObject);
protected
  procedure Notification(
    AComponent: TComponent; Operation: TOperation); override;
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  function CreateHandle: HDbiCur; override;
published
  property DataSource: TDataSource read GetDataSource write SetDataSource;
  property AllUsers: boolean read FAllUsers write FAllUsers default True;
  property AllLockTypes: boolean
    read FAllLockTypes write FAllLockTypes default True;
end;
```

descendant and the AllUsers and AllLockTypes properties are simply used as parameters to DbiOpenLockList. The LOCKLIST.DPR demo application on the disk also makes use of a TTableLocks component. We've already encountered the Notification method in the previously presented DBRecLabel component.

There is one drawback to using TTableLocks. Have you played with the LOCKLIST.DPR demo and spotted it? The record number field is missing; in fact it's been ignored by our new component. To explain why we'll need to delve a little into the topic of data translation between IDAPI logical field types and the physical field types supported by the different table drivers.

Each table type stores field types in different formats. A Paradox table stores a floating point value differently to a dBase or Interbase table. Without data translation an application would have to know the native internal storage format of each field type for the table in question. Field translation gets around this problem. When a field is retrieved from the table it gets converted from the native format to an IDAPI logical format; when a field is written back to the table it is converted from the logical format back to the native format. Table 3 lists the IDAPI field types.

A cursor or table's exltMode property specifies the translation mode for that table. xltFIELD specifies that field values are translated between the table's native or physical field types and IDAPI or logical field types. xltNONE specifies no translation and so the application is responsible for handling the table's physical field types. The translate mode of the LOCKS in-memory table is xltRECORD which is in effect similar to xltNONE in that no data translation occurs. Our problem occurs because Delphi's TDataSet doesn't expect to have to deal with the physical field types presented to it in its InternalOpen method, which in turn calls TFieldDefs.AddFieldDesc where LOCKS' record number field is treated as having an unknown type and isn't added to TDataset.FieldDefs. There is no simple workaround for this problem caused by fields of type unsigned 32-bit integer being unrecognised.

But that shouldn't cause us too much of a problem, thanks to the TLocksList object (not a component) which I've implemented in the LOCKINFO unit which you'll find on the disk. Listing 4 shows the object definition. This useful utility can be used to check for the presence of a user-specified matching table or record lock defined by the values in each field of the LOCKS in-memory table.

To use a TLocksList object you first need to create an instance of it. Each instance can be associated with a particular TTable by assignment to the instance's Table property. Here are the steps:

➢ Assign the active TTable to the Table property.

➤ *Table 3: IDAPI logical types*

IDAPI type	Description
fldZSTRING	Zero terminated array of chars
fldDATE	Date
fldBLOB	Binary Large Object
fldBOOL	16-bit signed integer
fldINT16	16-bit signed integer
fldINT32	32-bit signed integer
fldFLOAT	64-bit floating point
fldBCD	Binary coded decimal
fldBYTES	Fixed-length byte array
fldTIME	Time
fldTIMESTAMP	Timestamp (contains time and date)
fldUINT16	16-bit unsigned integer
fldUINT32	32-bit unsigned integer
fldFLOATIEEE	80-bit floating point
fldVARBYTES	Length prefixed byte array

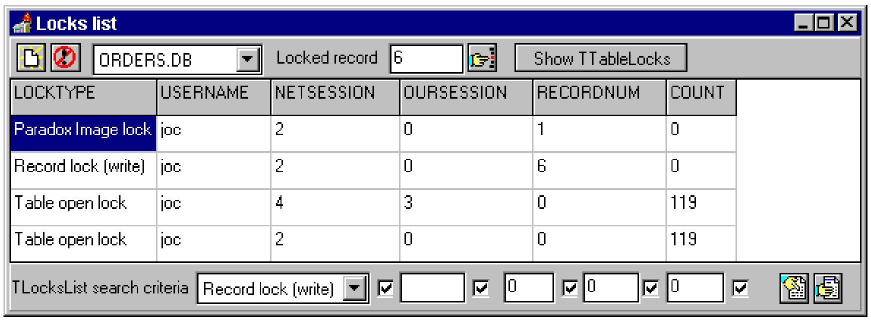
➤ *Listing 4: TLocksList types, properties and methods*

```
TLockInfoType = (liLockType, liUsername, liNetSess, liLocalSess, liRecNo);
TLookFor = set of TLockInfoType;
TLockType = (lkRecordWrite, lkRecordRead, lkPdoxGroup, lkPdoxImage, lkOpen,
  lkRead, lkWrite, lkExcl, lkError, lkUnknown, lkIgnore);
TUserName = string[DBIMAXUSERNAMELEN];
property Table: TTable;
property LockType: TLockType;
property UserName: TUserName;
property NetSession: Word;
property LocalSession: Word;
property RecNo: LongInt;
procedure SetParams(const LType: TLockType; const LUser: TUserName;
  const LNetSess, LOurSess: Word; const LRecNo: LongInt);
function FindFirst(var LockInfo: LOCKDesc): Boolean;
function FindNext(var LockInfo: LOCKDesc): Boolean;
```

➤ *Listing 5*

```
function DbiAnsiToNative(
 LdObj         : Pointer;     { Language driver }
 pNativeStr    : PChar;       { Destination buffer (opt) }
 pAnsiStr      : PChar;       { Source buffer }
 iLen          : Word;        { Length of buffer (opt) }
 var bDataLoss : Bool         { Returns TRUE if conversion will lose data (opt) }
 ): DBIResult;

function DbiNativeToAnsi(
  LdObj         : Pointer;     { Language driver }
  pAnsiStr      : PChar;       { Destination buffer (opt) }
  pNativeStr    : PChar;       { Source buffer }
  iLen          : Word;        { Length of buffer (opt) }
  var bDataLoss : Bool         { Returns TRUE if conversion will lose data (opt) }
  ): DBIResult;
```


Locks list

ORDERS.DB   Locked record 6   Show TTableLocks

LOCKTYPE	USERNAME	NETSESSION	CURSESSION	RECORDNUM	COUNT
Paradox Image lock	joc	2	0	1	0
Record lock (write)	joc	2	0	6	0
Table open lock	joc	4	3	0	119
Table open lock	joc	2	0	0	119

TLocksList search criteria   Record lock (write) ☑   ☑ 0 ☑ 0 ☑ 0 ☑

➤ *Figure 1: The LOCKLIST demo application*

➢ Set the search options to identify which fields in the LOCKS table will be used to locate a lock: these are set by assignment to the LookFor property.

➢ Set the locks search parameters by assignment to the LockType, Username, NetSession, LocalSession and RecNo properties. Alternatively use the SetParams method passing dummy values for any parameter excluded from the search parameters.

You're now ready to start locating matching locks. The first method you must call after setting search parameters is FindFirst which searches for a matching lock starting from the top of the LOCKS table. If a lock is found then FindFirst returns True and assigns the lock descriptor to the LockInfo parameter. You can then repeatedly call FindNext until it returns False in which case LockInfo contains an invalid lock descriptor. Don't try to call FindNext before calling FindFirst otherwise you'll raise an *Invalid method call: FindNext* exception. And don't forget to free the instance when you've finished with it.

The LOCKINFO unit also implements a function, GetLockUser, to return the username owning a lock on a particular record (useful before attempting to edit a record which may be locked):

```
function GetLockUser(
  ATable: TTable;
  RecNum: LongInt): TUserName;
```

If the record isn't locked then a blank string is returned. The demo application LOCKLIST.DPR on the disk (Figure 1) demonstrates the use of TLocksList and GetLockUser.

Mind Your Language

Whenever we call an IDAPI function that takes string arguments we must ensure that the string's character set (usually ANSI) matches that of the language driver for the IDAPI object involved in the function call. The two functions DbiAnsiToNative and DbiNativeToAnsi translate strings from ANSI to the language driver's native character set and vice-versa. The function prototypes are shown in Listing 5.

Fortunately Borland have provided the following function and procedure (which use the aforementioned functions) to simplify translation between ANSI and native language driver character sets:

```
function AnsiToNative(
  Locale: TLocale;
  const AnsiStr: string;
  NativeStr: PChar;
  MaxLen: Word): PChar;
procedure NativeToAnsi(
  Locale: TLocale;
  NativeStr: PChar;
  var AnsiStr: string);
```

Both are defined in the DB unit and provide the convenience of using Pascal strings to specify the ANSI string translated to or from the native character set. If you don't use ANSI to native translation you'll encounter problems with string arguments which contain lower case or non-alphabetic characters passed to an IDAPI function call. Likewise, un-translated native strings returned by an IDAPI function call may contain garbage characters. You've been warned!

The TLocale parameter used in AnsiToNative and NativeToAnsi is a pointer to the IDAPI language driver object associated with a BDE session, database and cursor/table. This language driver object is used in the translation between ANSI and native character sets and is destroyed when the associated IDAPI object is destroyed. This is important because you'll need to decide which TLocale to use when calling any of the above translation functions. If the IDAPI method called relates to an open TTable then use TTable.Locale, but if the table is closed then use TTable.DBLocale which corresponds to the database's language driver. If no database or table is open then use Session.Locale. Passing nil to a function requiring a TLocale causes the default System language driver (settable from the System page of the IDAPI Configuration Utility) to be used.

The DbiOpen... Functions

If you have been perusing the IDAPI User's Guide or on-line function reference you may have noticed a number of DbiOpen... functions such as DbiOpenUserList or DbiOpenIndexList (to name a few) which create in-memory tables containing information about some IDAPI object. For instance, DbiOpenTableList creates an in-memory table with information about all tables associated with a particular database. The cursor to the table is returned in the hCur parameter for these functions whose names, record descriptor and purpose are listed in Table 4. I've encapsulated most of these functions as the components TIdapiCfg, TTabInfo, TTableLocks and TDbUserList as listed in the *Component* column of Table 4.

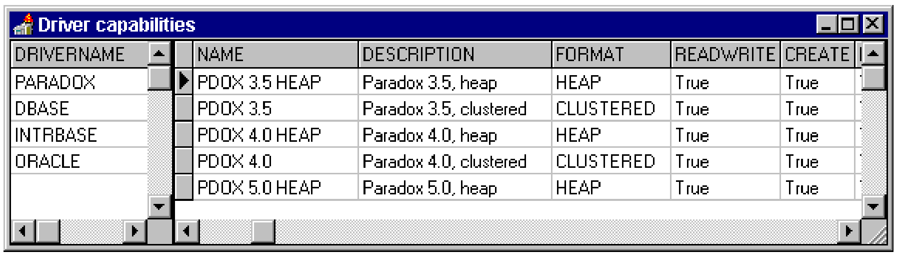
The information contained in the in-memory tables created by some of these functions is very useful indeed. We've already covered DbiOpenLocksList but take DbiOpenTableTypesList which returns detailed table type capability information, such as maximum record size, number of field types and the maximum number of fields in a table, for each table type supported by a particular table driver; the TABLECAP.DPR application (Figure 2) uses this function. The functions DbiOpenFieldTypesList

IDAPI Function	Descriptor	Purpose	Component
function DbiOpenCfgInfoList(hCfg : hDBICfg; eOpenMode : DBIOpenMode; eConfigMode : CFGMode; pszCfgPath : PChar; var hCur : hDBICur): DBIResult;	CFGDesc	Creates a table listing all the nodes in the IDAPI Configuration file accessed by pszCfgPath	
function DbiOpenDatabaseList(var hCur : hDBICur): DBIResult;	DBDesc	Creates a table listing all accessible databases found in IDAPI.CFG	TIdapiCfg
function DbiOpenDriverList(var hCur : hDBICur): DBIResult;	No descriptor: just DRIVERNAME field	Creates a table listing drivers available to the IDAPI client	TIdapiCfg
function DbiOpenFamilyList(hDb : hDBIDb; pszTableName : PChar; pszDriverType : PChar; var hFmlCur : hDBICur): DBIResult;	FMLDesc	Creates a table listing family members (.MB .PX .VAL files etc) for a specified table	TTableInfo
function DbiOpenFieldList(hDb : hDBIDb; pszTableName : PChar; pszDriverType : PChar; bPhyTypes : Bool; var hCur : hDBICur): DBIResult;	FLDDesc	Creates a table listing detailed field info for a specified table; bPhyTypes specifies whether logical or physical field types are returned	TTableInfo
function DbiOpenFieldTypesList(pszDriverType : PChar; pszTblType : PChar; var hCur : hDBICur): DBIResult;	FLDDesc	Creates a table listing field types supported by a specified table driver	TIdapiCfg
function DbiOpenFileList(hDb : hDBIDb; pszWild : PChar; var hCur : hDBICur;): DBIResult;	FILEDesc	Creates a table listing detailed info about all files in hDb matching pszWild	
function DbiOpenIndexList(hDb : hDBIDb; pszTableName : PChar; pszDriverType : PChar; var hCur : hDBICur): DBIResult;	IDXDesc	Creates a table listing detailed index info for a specified table	TTableInfo
function DbiOpenIndexTypesList(pszDriverType : PChar; var hCur : hDBICur): DBIResult;	IDXType	Creates a table listing index types supported by a specified table driver	TIdapiCfg
function DbiOpenLdList(var hCur : hDBICur): DBIResult;	LDDesc	Creates a table listing available language drivers	TIdapiCfg
function DbiOpenLockList(hCursor : hDBICur; bAllUsers : Bool; bAllLockTypes : Bool; var hLocks : hDBICur): DBIResult;	LOCKDesc	Creates a table listing locks on the table specified by hCursor; this table can include just record locks or all locks for all users or just the current user	TTableLocks
function DbiOpenRintList(hDb : hDBIDb; pszTableName : PChar; pszDriverType : PChar; var hChkCur : hDBICur): DBIResult;	RINTDesc	Creates a table listing referential integrity links for a specified Paradox table	TTableInfo
function DbiOpenSecurityList(hDb : hDBIDb; pszTableName : PChar; pszDriverType : PChar; var hSecCur : hDBICur): DBIResult;	SECDesc	Creates a table listing record-level security details for a specified table	
function DbiOpenSPList(hDb : hDBIDb; bExtended : Bool; bSystem : Bool; pszQual : PChar; var hCur : hDBICur): DBIResult;	SPDesc	Creates a table listing information about the stored procedures (may include system procedures) associated with database hDb	
function DbiOpenSPParamList(hDb : hDBIDb; pszSPName: PChar; bPhyTypes: Bool; uOverload: Word; var hCur : hDBICur): DBIResult;	SPParamDesc	Creates a table listing the parameters associated with a specified stored procedure, pszSPName	
function DbiOpenTableList(hDb : hDBIDb; bExtended : Bool; bSystem : Bool; pszWild : PChar; var hCur : hDBICur): DBIResult;	TBLFullDesc which is the combination of TBLBaseDesc and TBLExtDesc if bExtended = True	Creates a table listing detailed info about all tables in hDb matching pszWild: bSystem includes SQL system tables; bExtended returns extended table info	TTableInfo
function DbiOpenTableTypesList(pszDriverType : PChar; var hCur : hDBICur): DBIResult;	TBLType	Creates a table listing table type names for the specified table driver	TIdapiCfg
function DbiOpenUserList(var hUsers : hDBICur): DBIResult;	UserDesc	Creates a table listing all current IDAPI users	TDBUserList
function DbiOpenVchkList(hDb : hDBIDb; pszTableName : PChar; pszDriverType : PChar; var hChkCur : hDBICur): DBIResult;	VCHKDesc	Creates a table listing Paradox table validity checks or a table listing required fields for an SQL table	TTableInfo
Note: Where a function uses parameters szTableName and szDriverType, the latter can be omitted if szTableName includes a file extension			

➤ *Table 4: Useful DbiOpen... functions* and DbiOpenIndexTypes list are also very useful for determining field type and index type capabilities for a particular table driver. Whilst these IDAPI functions are without doubt very useful for telling you what the BDE can do or what a particular table's properties are, most of the information contained in these information tables is available via Delphi's data access components. For instance, the information obtained at run time using DbiOpenFieldList is more easily obtained from the TTable FieldDefs property; it's easier to call Session.GetAliasNames rather than call DbiOpenDatabaseList and Session.GetTableNames instead of DbiOpenTableList.

My point is, don't get carried away using calls to IDAPI where a VCL property or method will achieve the same end.

I won't go into the details of how the TIdapiInfo, TTableInfo and TDBUserList components work: they're pretty similar to the TTableLocks component in that the CreateHandle method is overridden to initialise the dataset's Handle property. See for yourself in the component's source and play around with them in the IDE. The components just discussed are useful encapsulations of most of the DbiOpen... functions listed in Table 4, but what about those functions not encapsulated as a component? That's where TGenTable comes in handy. It is a TDataSet descendant whose CreateHandle method and FHandle property are overridden so that it's Handle property is declared public and read/write. This means that a valid IDAPI cursor handle can be assigned to Handle before opening or activating the dataset. It also means you can assign any old pointer to Handle and potentially wreak havoc! Unlike TIdapiInfo etc, you can't use TGenTable interactively within the IDE because you can only assign to the Handle property at run-time, but using TGenTable instead of one of the other in-memory table components does give more flexibility. Listing 6 shows how to use TGenTable with a call to DbiOpenSecurityList.


Driver capabilities

DRIVERNAME
PARADOX
DBASE
INTRBASE
ORACLE

NAME	DESCRIPTION	FORMAT	READWRITE	CREATE
PDOX 3.5 HEAP	Paradox 3.5, heap	HEAP	True	True
PDOX 3.5	Paradox 3.5, clustered	CLUSTERED	True	True
PDOX 4.0 HEAP	Paradox 4.0, heap	HEAP	True	True
PDOX 4.0	Paradox 4.0, clustered	CLUSTERED	True	True
PDOX 5.0 HEAP	Paradox 5.0, heap	HEAP	True	True

➤ *Figure 2: The TABLECAP application*

```
procedure TForm1.DoItClick(Sender: TObject);
var Cur: hDBICur;
begin
  Check(DbiOpenSecurityList(Database1.Handle, 'CUSTOMER.DB', nil, Cur));
  MemTable1.Handle := Cur;
  MemTable1.Open;
end;
```

➤ *Listing 6: Using TGenTable*

```
function DbiCreateInMemTable(
  hDb             : hDBIDb;          { Database handle }
  pszName         : PChar;           { Logical Name }
  iFields         : Word;            { No of fields }
  pfldDesc        : pFLDDesc;        { Array of field descriptors }
  var hCursor     : hDBICur          { Returned cursor handle }
  ): DBIResult;
```

➤ *Listing 7*

Be aware that you must always obtain a valid cursor handle (ie from an active dataset) to assign to the Handle property before opening the TGenTable. If the table is closed the Handle property becomes invalid because the associated cursor will have been closed and the cursor freed.

We know we can use TGenTable with any IDAPI function which returns a valid cursor, such as DbiCreateInMemTable, which creates a temporary in-memory table (Listing 7).

The INMEMTAB.DPR demo application on the disk demonstrates the use of this function with a TGenTable component. Unlike the in-memory tables previously encountered, the eOpenMode property is dbiREADWRITE. Strange, then, that records in a table created using DbiCreateInMemTable can only be added or edited but not deleted. This limits the usefulness of in-memory tables somewhat and really makes them just a glorified expandable array of records. Let's build an in-memory table.

Like most tables, the main requirements of an in-memory table are a database, a table name and a bunch of fields. The fields are simply an array of IDAPI field descriptors, as defined in Listing 8.

Despite the number of fields in this record we need only specify the first six (after initialising the record to nulls) in order to create a valid field descriptor. The short procedure in Listing 9 creates an in-memory table with two fields and assigns the cursor to the table to an instance of TGenTable.

Be aware that in-memory tables support only IDAPI logical field types. Instead of going through the tedium of defining the field descriptors for the in-memory table we can use the field descriptors of another table, borrowing its structure. DbiGetFieldDescs achieves this:

```
function DbiGetFieldDescs(
  { Cursor handle }
  hCursor      : hDBICur;
  { Array of field descriptors}
  pfldDesc     : pFLDDesc
  ): DBIResult;
```

The TInMemTable component does just this via its BorrowFrom property which references an open TTable or TQuery instance. Listing 10 shows TInMemTable's CreateHandle method which illustrates how to use DbiGetFieldDescs.

The reason for the call to CheckIsBorrowFromActive is to avoid passing a dud cursor handle to DbiGetCursorProps or DbiGetFieldDescs. Once the TInMemoryTable instance is opened you can add as many records as you like, but you can't delete any.

Temporary Tables

Continuing on from in-memory tables, IDAPI provides temporary tables which differ in that they can be committed to disk and made permanent. They also support indexes and can be fully modified, which is much more useful to us. The function DbiCreateTempTable creates such a table:

```
function DbiCreateTempTable(
  { Database handle }
  hDb            : hDBIDb;
  { Table description }
  var crTblDsc  : CRTblDesc;
  { Returned cursor on table }
  var hCursor   : hDBICur
 ): DBIResult;
```

In common with DbiDoRestructure, this function takes a table descriptor (CRTblDesc) which we briefly examined last month. Let's look at it in more detail (see Listing 11).

The table descriptor record is little more than a collection of various descriptors and their counts. Note that temporary tables don't support referential integrity (for obvious reasons!) but passwords (for Paradox-type temporary tables) are supported. The driver type can be any of the standard table types.

As with in-memory tables I've created the TTempTable (a subclassed TTable) component to encapsulate the cursor returned by DbiCreateTempTable. Like TInMemTab it borrows an active TTable or TQuery's structure via the BorrowFrom property. A TTable's indexes can be borrowed provided that TTempTable.BorrowIndexes is set. In contrast to in-memory tables, the field descriptors are passed within the table descriptor which also includes index descriptors, but out of all the fields in CRTblDesc only a handful are required for a call to DbiCreateTempTable, as shown in the code snippet from TTempTable.CreateHandle in Listing 12.

Notice the last line of code to change the translate mode to xltFIELD. This is necessary because temporary tables use physical field types by default whereas the database VCL supports only logical field types.

There are a few gotchas when using TTempTable. I mentioned that temporary tables support indexes; to make use of this in TTempTable its TableName property (which is otherwise irrelevent for opening a TTempTable) must be set to that of the TTable referenced by the BorrowFrom property. The reason for this is in the way in which a TTable's IndexDefs are built, relying on a call to DbiOpenIndexList (which uses a table name) rather than DbiGetIndexDescs (which uses a cursor handle) which makes life easier for the VCL development team. However, because the temporary table's structure is borrowed from the table named by TableName, the indexes will be exactly the same and so TTempTable.IndexName can be used to switch indexes. Great! However, there's another, more serious, problem which severely limits the

➤ Listing 8

```
FLDDesc = record
  iFldNum         : Word;         { Field number (1..n) }
  szName          : DBINAME;      { Field name }
  iFldType        : Word;         { Field type }
  iSubType        : Word;         { Field subtype (if applicable) }
  iUnits1         : Integer;      { Number of Chars, digits etc }
  iUnits2         : Integer;      { Decimal places etc. }
  iOffset         : Word;         { Offset in the record (computed) }
  iLen            : Word;         { Length in bytes (computed) }
  iNullOffset     : Word;         { For Null bits (computed) }
  efldvVchk       : FLDVchk;      { Field Has vcheck (computed) }
  efldrRights     : FLDRights;    { Field Rights (computed) }
end;
```

➤ Listing 9

```
procedure CreateInMemTable(Value: TDatabase);
var fd: array [0..1] of FLDDesc;
    c:  HDbiCur;
begin
  FillChar(fd, 2*sizeof(FLDDesc), 0);
  fd[0].iFldNum  := 1;
  StrPCopy(fd[0].szName, 'FieldOne');
  fd[0].iFldType:= fldZSTRING;
  fd[0].iSubType:= 0;
  fd[0].iUnits1  := 5;
  fd[0].iUnits2  := 0;
  fd[1].iFldNum  := 2;
  StrPCopy(fd[1].szName, 'FieldTwo');
  fd[1].iFldType:= fldZSTRING;
  fd[1].iSubType:= 0;
  fd[1].iUnits1  := 5;
  fd[1].iUnits2  := 0;
  Check(DbiCreateInMemTable(Value.Handle, 'ATABLE', 2, @fd, c));
  MyGenTable.Handle := c;
  MyGenTable.Open;  {opens an instance of TGenTable}
end;
```

➤ Listing 10

```
function TInMemTable.CreateHandle: HDbiCur;
var PFieldDescs: Pointer;
    Props: CURProps;
    szTableName: DBITBLNAME;
begin
  Result := nil;
  PFieldDescs := nil;
  CheckIsBorrowFromActive;
  StrPCopy(szTableName, 'INMEMORYTABLE');
  Check(DbiGetCursorProps(FBorrowFrom.Handle, Props));
  try
    PFieldDescs := AllocMem(Props.iFields * sizeof(FLDDesc));
    Check(DbiGetFieldDescs(FBorrowFrom.Handle, PFieldDescs));
    Check(DbiCreateInMemTable(Database.Handle, szTableName,
      Props.iFields, PFieldDescs, Result));
  finally
    if Assigned(PFieldDescs) then
      FreeMem(PFieldDescs, Props.iFields * sizeof(FLDDesc));
  end;
end;
```

usefulness of TTempTable: you can't modify its records. The culprit is this code from TDataset.InternalOpen:

```
DbiGetCursorProps(
  FHandle, CursorProps);
FRecordSize :=
  CursorProps.iRecBufSize;
FBookmarkSize :=
  CursorProps.iBookmarkSize;
FCanModify :=
  (CursorProps.eOpenMode =
   dbiReadWrite) and not
  CursorProps.bTempTable;
```

The reason for this lies in the design of the database VCL. Non-live query results are returned in temporary tables and as we well know such query results are read-only, hence the above assignment to FCanModify. Of course there's nothing to prevent you from altering the offending code in DB.PAS to make TTempTable read/write via the VCL (except that's it's a bit naughty and breaks the rules of OOP) and perhaps in future versions of the VCL, Borland will do just that (please?).

If you're an OOP purist and don't wish to alter DB.PAS you can always modify the table using IDAPI functions. DbiDeleteRecord does what it says and is easy to use, but adding records and modifying fields is a little trickier.

Another difference between in-memory and temporary tables is that the latter can be made permanent instead of being destroyed when closed. The DbiMakePermanent function achieves this:

```
function DbiMakePermanent(
  { Cursor handle }
  hCursor      : hDBICur;
  { Rename temporary table }
  pszName      : PChar;
  { Overwrite existing file }
  bOverWrite   : Bool
  ): DBIResult;
```

The first argument, hCursor, must be a handle to a temporary table, it won't work with an in-memory table. pszName is the name of the newly committed table and bOverWrite specifies whether an existing table of the same name is overwritten. After calling this function the table is made permanent but isn't immediately committed to disk, but a call to DbiSaveChanges solves that. In fact a temporary table can be made permanent and committed to disk with just a call to DbiSaveChanges but you cannot specify the permanent table's name using just this function.

Local query results can be made permanent with the function calls:

```
Check(DbiMakePermanent(
  Query1.Handle, 'QUERY1.DB',
  True));
Check(DbiSaveChanges(
  Query1.Handle));
```

If you examine a TQuery's cursor properties you'll find that the cursor is in fact a temporary table; it may interest you to examine a query result cursor's properties before and after a call to DbiMakePermanent or DbiSaveChanges. All in all, temporary tables are very useful, particularly for processing or modifying query results (batch moved from a TQuery) or as intermediate tables used as part of some larger processing task.

➤ *Listing 11*

```
CRTblDesc = record
  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      : Bool;          { Master password supplied in szPassword }
  bPack           : Bool;          { 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;
```

➤ *Listing 12*

```
FillChar(TblDesc, sizeof(CRTblDesc), 0);
with TblDesc do begin
  StrCopy(szTblName, szTableName);
  StrCopy(szTblType, Props.szTableType);
  iFldCount := Props.iFields;
  pfldDesc  := PFieldDescs;
  iIdxCount := Props.iIndexes;
  pidxDesc  := PIndexDescs;
end;
Check(DbiCreateTempTable(Database.Handle, TblDesc, Result));
Check(DbiSetProp(HDBIObj(Result), curXLTMODE, LongInt(xltFIELD)));
```

Conclusion

In this series of articles we've covered much about the BDE that is useful to the Delphi developer needing to extend the VCL's database access capabilities. As you can see, the BDE provides much more functionality than is used by Delphi's data-access components and hopefully I've whetted your appetite to dig deeper into IDAPI, particularly in future versions of the product which will only get better and better. Note that an updated IDAPI help file for Delphi 1.02 (file BDEHELP.ZIP) can be downloaded from Compuserve and Borland's web site at www.borland.com.

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