Digging Into IDAPI Part 2

by John O'Connell

Last month I introduced you to the concept of cursor and record properties. Let's take a closer look at cursor properties and assess the usefulness of each individual property. Table 1 lists them all.

We used the `iSeqNums` property last month to retrieve a record's number for local tables. Many of the property names are self explanatory but let's discuss a few of the more useful properties.

Bookmarks

The `bBookMarkStable` property indicates whether or not bookmarks can be relied upon to identify the correct position of a bookmarked record despite any changes to the table. Bookmarks are stable with dBase tables and tables with a primary key/index; bookmarks are not stable with Paradox heap tables, that is, tables without a primary key/index. You can experiment with this yourself by writing a simple database application containing a `TTable`, `TDatasource`, `TDBGrid` and `TDBNavigator`, then add two `TButton` controls whose `OnClick` handlers allow you to set a bookmark (using the method `TTable.GetBookmark`) and move to that previously set bookmark (using `TTable.GotoBookmark`). Now run the application, set a bookmark anywhere in the table, delete the bookmarked record and then move to any other record. If your table is a keyed Paradox table or a dBase table, moving to the bookmark will raise a *Record/key deleted* exception; if your table is a Paradox heap table then moving to the bookmark will simply move to the record previously positioned after the deleted record. In general, unstable bookmarks point to an absolute record position in a table rather than to a particular record, therefore new and deleted records may affect the bookmark. To illustrate this, try setting a bookmark on the last record of a Paradox heap table, delete the record and then move to the bookmark – see what happens? Stable bookmarks are guaranteed to position the cursor at the correct record unless it has been deleted.

When an index is activated on the table, any previously set bookmarks become invalid: moving to such a bookmark will at best raise an exception, at worst the view of the table will be corrupted. This behaviour occurs because activating an index changes the bookmark's size as retrieved from the `iBookMarkSize` cursor property. If the bookmark size has changed, any previously set bookmarks become invalid and cannot be used.

But what exactly is a bookmark? The Delphi type `TBookmark` is actually a pointer to a string of bytes in which bookmark data is stored: this data contains internal IDAPI information about the current position of the cursor. To get a bookmark for the current record the Delphi programmer simply calls `TTable.GetBookmark` which returns a `TBookmark`, but the BDE programmer must first call `DbiGetCursorProps` to retrieve the size of the desired bookmark, then allocate a bookmark buffer of `iBookMarkSize` bytes and finally call `DbiGetBookMark`, passing the cursor handle and a pointer to the bookmark buffer, which gets filled with bookmark data by the function call. This bookmark buffer is the same as a `TBookmark`.

We can compare bookmarks (set within the same active index) with a call to `DbiCompareBookmarks`:

```
function DbiCompareBookMarks(
  hCur: hDBICur; pBookMark1:
  Pointer; pBookMark2:
  Pointer; var CmpBkmkResult :
  Word): DBIResult;
```

by passing a `TTable Handle` and the `TBookmarks` to be compared. For example:

```
var BkMk1, BkMk2: TBookmark;
    CmpResult: Word;
begin
  ...
  BkMk1 := Table1.GetBookmark;
  Table1.MoveBy(20);
  BkMk2 := Table1.GetBookmark;
  Check(DbiCompareBookmarks(
    Table1.Handle, BkMk1,
    BkMk2, CmpResult));
  ...
end;
```

The return value in `CmpResult` will be `CMPLess` if `BkMk1` is before `BkMk2`, `CMPEql` if `BkMk1` is the same as `BkMk2`, `CMPGtr` if `BkMk1` is after `BkMk2`, or `CMPKeyEql` when `BkMk1` and `BkMk2` have the same key value. A call to this function can be useful when you're programmatically navigating a dataset which has no concept of record numbers, such as a query result or an SQL table. For instance, if your application requires the user to specify a range of records to be processed in some way, the user could set two bookmarks between which records will be processed: your application would then use `DbiCompareBookMarks` to determine which bookmark to start processing from and advance the record pointer until the other bookmark is reached.

Table Rights And Passwords

The `eOpenMode` property indicates whether the table was opened as read/write or read only: setting a `TTable`'s `ReadOnly` property to `True` before opening the table results in `eOpenMode` being set to `dbiREADONLY`. Similarly `eShareMode` indicates whether the table was opened for shared or exclusive use: setting `TTable.Exclusive` to `True` before opening the table results in `eShareMode` being set to `dbiOPENSHARED`.

Paradox tables can be assigned passwords with associated table privileges. There are two types of password: the master password and auxiliary passwords. The master password gives a user full

Property Name	Type	Description
szName	DBITBLNAME	Table name
iFNameSize	Word	Size of buffer to hold expanded file name
szTableType	DBINAME	Table driver type (PARADOX, DBASE, etc)
iFields	Word	Number of fields
iRecSize	Word	Record size (logical record if exltMode is xltFIELD, else physical record)
iRecBufSize	Word	Record size (physical record)
iKeySize	Word	Key size
iIndexes	Word	Number of indexes
iValChecks	Word	Number of Paradox table validity checks
iRefIntChecks	Word	Number of Paradox referential integrity constraints
iBookMarkSize	Word	Bytes needed to allocate a Bookmark
bBookMarkStable	Bool	Are bookmarks stable?
eOpenMode	DBIOpenMode	Table open mode: ReadOnly or ReadWrite
eShareMode	DBIShareMode	Table opened Exclusive or Shared
bIndexed	Bool	Is an index active?
iSeqNums	Integer	1 indicates uses logical record sequence numbers, 0 indicates uses physical record numbers (dBase)
bSoftDeletes	Bool	Supports soft deletes? (dBase)
bDeletedOn	Bool	Can soft deleted records be seen?
iRefRange	Word	Not used
exltMode	XLTMode	Translate Mode: XLTNONE or XLTField
iRestrVersion	Word	Restructure version number
bUniDirectional	Bool	Is the cursor uni-directional (SQL only)?
eprvRights	Word	Table rights (depends on current password)
Dummy4	Word	Used in BDE 3.0 / Delphi 2.0 only
iFmlRights	Word	Not used
iPasswords	Word	Number of Auxiliary passwords
iCodePage	Word	Codepage (0 if unknown, used by dBase)
bProtected	Bool	Is the table is protected by password?
iTblLevel	Word	Driver dependent table level
szLangDriver	DBINAME	Language driver name
bFieldMap	Bool	Is a field map active?
iBlockSize	Word	Physical file block size in Kb
bStrictRefInt	Bool	Is strict referential integrity used?
iFilters	Word	Number of filters current on the cursor
bTempTable	Bool	Is the table a temporary table?
iUnUsed	array [0..15] of Word;	Not used

Notes

```
DBITBLNAME   = array [0..DBIMAXTBLNAMELEN] of Char;
DBINAME      = array [0..DBIMAXNAMELEN] of Char;
DBIOpenMode  = (dbiREADWRITE, dbiREADONLY);
DBIShareMode = (dbiOPENSHARED, dbiOPENEXCL);
XLTMode      = (xltNONE,  xltRECORD, xltFIELD);
```

Notes for Delphi 2.0 / BDE 3.0

Type WordBool is used instead of Bool
Type SmallInt is used instead of Integer
Type packed array[0..15] of Word is used instead of array[0..15] of Word

➤ *Table 1: IDAPI cursor properties*

rights on the table whereas auxiliary passwords give a user restricted rights on the table. Table 2 lists the table privileges supported. The eprvRights property specifies the table level rights for the user who opened the table. If the table is not password protected (check the bProtected cursor property) the user has full table rights. The number of auxiliary passwords can be determined using the iPasswords property. Do bear in mind that a cursor's properties can only be retrieved when the cursor/table is open, therefore to examine a protected table's properties the user must first supply a password to open the table: obviously you can't examine the table's protection properties unless you have opened the table with a password which bestows privileges other than prvNone!

Un-Deleting Records

The concept of "soft deletes" is supported by dBase tables, meaning that deleted records aren't physically removed from the table but are flagged as deleted and are not displayed in the table. Deleted records are removed from the table only when the table is packed. The bSoftDeletes cursor property indicates whether the table supports soft deletes, but what use are soft deleted records if they can't be seen? Well they can, by setting the bDeletedOn cursor property. So far we've just looked at *getting* cursor properties but what about *setting* them? An IDAPI object's properties can be set by calling the DbiSetProp function:

```
function DbiSetProp(
  { Object handle: }
  hObj       : hDBIObj;
  { Property to set: }
  iProp      : Longint;
  { Property value: }
  iPropValue : Longint
  ): DBIResult;
```

hObj is a generic object handle to which a handle for any IDAPI object can be passed just as long as the handle is cast as type hDBIObj. The property we wish to set is curSOFTDELETETON with a property

value of 1. The following function call sets the bDeletedOn property for Table1:

```
DbiSetProp(HDBIObj(
  Table1.Handle),
  curSOFTDELETEON, 1);
```

After setting this property we must refresh the view of the table by calling TTable.Refresh and voila: we can now see any soft deleted records; but how can we distinguish between deleted and non-deleted records? If you cast your keen mind back to the discussion of record properties in last month's article you may remember the bDeleteFlag property which, oddly enough, identifies a deleted dBase record. Deleted records can be un-deleted by calling DbiUndeleteRecord which takes a cursor handle as its sole parameter. Attempting to un-delete a non-deleted record raises an exception when called within the Check procedure.

An Enhanced TTable

It's time to put all we've learned so far about IDAPI into practice: using Dbi functions and IDAPI properties we can set to work developing a sub-classed TTable which implements all these extras. There appears to be a naming convention for components which involves prefixing the component name with the vendor's lower case initials, for example, Woll2Woll's InfoPower components are prefixed with ww. In a similar spirit we'll give our enhanced TTable the name TjocTable – if you'll please pardon my ego!

TjocTable implements as methods those Dbi function calls covered in last month's article: record and cursor properties have been surfaced as component properties. See this month's disk for the source of TjocTable to which we'll add functionality as we cover IDAPI properties and functions in greater depth. Support for soft deletes has been implemented in the Deleted read-only property and ShowDeleted property which calls the SetShowDeleted property write access method. The functionality

Privilege	Description
prvNONE	No privileges
prvREADONLY	Can read the table or field
prvMODIFY	Can read and modify fields
prvINSERT	Can insert records and all of the above
prvINSDEL	Can delete records and all of the above
prvFULL	Full rights
prvUNKNOWN	Unknown rights

➤ *Table 2: Paradox table/field privileges*

balance between TTable in Delphi 1.0 and 2.0 has been somewhat redressed by implementing the RecNo property. The subtle bug in Delphi 1.0's TTable.MoveBy is worked around by providing a MoveRelative method which achieves the same thing but without the limitations of a signed integer parameter. The new GotoRecord method enables movement to any record number or sequence number in the table: just remember what I said about sequence numbers in last month's article.

I've also implemented the properties IsProtected, PasswordCount and TableRights, which indicate whether the table is password protected, the number of auxiliary passwords and the table access rights for the user. I've been a little naughty and published these properties which should be read-only but have been made read-write (so that they can be published) with empty write-access methods, thus effectively making them read-only properties. My reason for publishing these properties is pure convenience: you don't have to write code to examine the properties at run-time.

There are other useful cursor properties which relate to table type and capability. The iTblLevel property specifies the table type version: Paradox 7.0 format tables are level 7, Paradox 5 tables are level 5 and so on. This property is useful for diagnostic purposes. The iBlockSize property specifies the size of a table allocation block in Kb. What effect does block size have on a table? It is only relevant for Paradox tables and determines the maximum size of the table. Here's how: a Paradox table contains a maximum of 64K blocks which means that a table with 2Kb block size can grow to a maximum size of 128Mb or 256Mb with a 4Kb block size. Paradox 5.0 tables can have a maximum block size of 32Kb. Each block contains table records. Choosing a block size must take into account storage efficiency and performance considerations: depending on the record size, smaller blocks can be more efficient but larger blocks will improve performance (at the cost of memory usage), especially where a large number of records are concerned. Furthermore, table record size cannot exceed block size because records are not stored across block boundaries, therefore you may have no choice but to use a larger block size which will be chosen for you by the BDE when the table is created.

A table's block size can be set using the IDAPI Configuration Utility. On the Drivers page select PARADOX and set the blocksize parameter as a multiple of 1024 and save the changes to IDAPI.CFG, any Paradox tables created in future will use the new block size. To change an existing table's block size you'll need to create a copy of the table (with a different name) using a TBatchMove component or by using Database Desktop to query out all the records in the table to an answer table which will have the new block size. Just copying the table using Database Desktop won't do the trick.

The iRestrVersion cursor property is a count of the number of times a Paradox table has been restructured and can be useful for making sure that a large table is packed at regular intervals to

```
function DbiRenameTable(    { Rename table & family }
  hDb           : hDBIDb;   { Database handle }
  pszOldName    : PChar;    { Old name }
  pszDriverType : PChar;    { Driver type /NULL }
  pszNewName    : PChar     { New name }
  ): DBIResult;
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;
```

➤ *Listing 1*

defragment its contents. You can also use it to check if a table has been restructured with excessive frequency.

I've implemented the published read-only properties `RestructVersion`, `BlockSize` and `TableLevel` to encapsulate the `iRestrVersion`, `iBlockSize` and `iTblLevel` cursor properties.

Copying And Renaming Tables

It's possible to copy and rename tables within a database. This saves us having to rely on a dataset's `BatchMove` method to achieve the same result. `DbiRenameTable` and `DbiCopyTable` can be used to rename and copy tables (see Listing 1).

Both functions require a valid database handle with source and destination table names, but the driver type can be passed as `nil` provided the table names have extensions, otherwise the driver type must be either `szPARADOX` or `szDBASE` which are string constants defined in the `DBITYPES` unit. The driver type is ignored if `hDb` is an SQL database and can thus be passed as `nil`. A table to be copied must be at least read-locked beforehand, a table to be renamed must be exclusively locked. You can lock a table using the `TTable.LockTable` and `UnlockTable` methods:

```
TLockType =
  (ltReadLock, ltWriteLock);
procedure TTable.LockTable(
  LockType: TLockType);
procedure TTable.UnlockTable(
  LockType: TLockType);
```

To obtain an exclusive lock on a `TTable` you must first set its `Exclusive` property to `True` and then open the table; the table must then be closed just before the call to `DbiRenameTable` otherwise a *Table is busy* exception will be raised. Neither of these two Dbi functions can be used to change a table's type: a Paradox table (.DB) cannot be renamed to a dBase table (.DBF) or vice versa.

I've implemented the `CopyTable` and `RenameTable` methods for Delphi 1.0. Delphi 2.0 provides the `TTable.RenameTable` method so only `CopyTable` is implemented in Delphi 2.0, by use of the `Win32` conditional compilation flag.

Packing Tables

Packing a table generally means that the space used by deleted records gets reclaimed and the table gets defragmented. Specifically, packing a dBase table removes any soft deleted records and is achieved with a call to the function `DbiPackTable`.

To pack a Paradox table you must call `DbiDoRestructure` with a `pack` flag set to `True`. Let's examine these in more detail, starting with the function prototype for `DbiPackTable`:

```
function DbiPackTable(
  { Database handle: }
  hDb           : hDBIDb;
  { Cursor: }
  hCursor       : hDBICur;
  { (OR) Table name: }
  pszTableName  : PChar;
  { Driver type /NULL: }
  pszDriverType : PChar;
  { Regenerate indexes: }
  bRegenIdxs    : Bool
  ): DBIResult;
```

Here we have some of the usual parameters required when dealing with a cursor/table (the database handle and cursor handle) as well as null-terminated string parameters which specify the table name and table type.

If `pszTableName` includes a file extension then `pszDriverType` can be passed as `nil`, otherwise it must be passed the predefined string constant `szDBASE`. The good news is that if `hDb` and `hCursor` are supplied then both table name and table type can be passed as `nil` which makes life a little easier. The `bRegenIdxs` parameter specifies whether all indexes are to be rebuilt in addition to packing the table.

Packing Paradox tables is a little more involved because calling `DbiDoRestructure` is slightly less straightforward, as it involves setting up a table descriptor record which is then passed as a parameter to the function.

Rather than listing all the fields in the table descriptor, we'll just examine those which are relevant to our purpose of packing Paradox tables:

```
CRTblDesc = record
  { TableName with optional
    path & extension: }
  szTblName : DBITBLNAME;
  { Driver type (optional): }
  szTblType : DBINAME;
  ...
  { Pack table
    (restructure only): }
  bPack     : Bool;
  ...
end;
```

and here's the function prototype for `DbiDoRestructure`:

```
function DbiDoRestructure(
  { Database handle: }
  hDb           : hDBIDb;
  { Number of table
    descriptors (1): }
  iTblDescCount : Word;
  { Array of table descs: }
  pTblDesc      : pCRTblDesc;
  { Save to this table
    (optional): }

  { continued next page... }
```

```
  pszSaveAs      : PChar;
  { Keyviol table name
    (optional): }
  pszKeyviolName : PChar;
  { Problems table name
    (optional): }
  pszProblemsName : PChar;
  { Analyze restructure: }
  bAnalyzeOnly   : Bool
  ): DBIResult;
```

The only parameters we need to specify are `hDB`, `iTblDescCount` which is always 1 and the pointer to the table descriptor, `pTblDesc`; the others can be passed as `nil` or `False` as appropriate. If you wish to save the restructured table to a different name you can specify the table name in `pszSaveAs`.

Before calling this function we must first initialise the table descriptor with the table name, table type and `bPack` set as `True`. Unlike `DbiPackTable`, the table to be restructured must be closed before calling `DbiDoRestructure`, otherwise the function will fail and return a *table busy* result code.

Before calling `DbiPackTable` or `DbiDoRestructure` the table must be opened exclusively. See the `Pack` and `PackPdoxTable` methods to see how these functions are used. Conveniently, `DbiDoRectructure` also rebuilds the table's indexes which leads us on to...

Rebuilding Table Indexes

It's not necessary to call on `DbiDoRestructure` or `DbiPackTable` just to rebuild a table's indexes, IDAPI provides function calls specially for the purpose. First let's discuss table indexes and why/when they need to be rebuilt.

Table indexes are just like book indexes – they help to locate a particular field value much faster then using a sequential table scan. An index is basically a type of lookup table where each record contains a particular field value (or field values for a composite or multi-field index) and its associated record positions within the indexed table.

So if you're searching for the first record in a customer address table where `PostCodeArea = 'W1'`, rather than scanning each record, the index is used to locate the value `'W1'` and then the first associated record position is retrieved from the index record. There are two types of index: maintained and non-maintained. Strangely enough, maintained indexes are updated as and when the table is updated whereas non-maintained indexes remain static so there's obviously some (usually small) performance overhead when using maintained indexes.

Delphi's `TTable` maintains a list of indexes in its `IndexDefs` property of type `TIndexDefs`: an array of `TIndexDef` objects, each of which encapsulates a single table index. However, only maintained indexes are stored in `IndexDefs`, non-maintained indexes are ignored, as you can readily see from the source for `TTable`'s `UpdateIndexDefs` method and also from the list of available indexes for `TTable`'s `IndexName` property in the Object Inspector.

Fortunately for us this does not mean that non-maintained indexes cannot be used in Delphi applications: simply assigning the name of the non-maintained index to the `IndexName` property of a `TTable` will make that index active, but beware that non-maintained active indexes render the table view read-only. Why is this the case? Well it actually makes perfect sense: if the active index is non-maintained then it'll become out of date as soon as the table is modified and obviously that can't be allowed to happen.

What if a non-maintained index is created and then the table is modified when that index in not active – what happens then? The non-maintained index will become out of date and any attempt to make it the active index will fail with an *Index is out of date* exception which necessitates rebuilding the index.

IDAPI provides two functions for rebuilding indexes: `DbiRegenIndex` and `DbiRegenIndexes` which regenerate a single specified index and all indexes respectively. The `DbiRegenIndexes` function takes just a cursor handle as a parameter whereas `DbiRegenIndex` is a little more involved:

```
function DbiRegenIndex(
  { Database handle: }
  hDb           : hDBIDb;
  { Cursor (OR): }
  hCursor       : hDBICur;
  { Table name: }
  pszTableName  : PChar;
  { Driver type: }
  pszDriverType : PChar;
  { Index name: }
  pszIndexName  : PChar;
  { Index tagname: }
  pszIndexTagName : PChar;
  { Index number: }
  iIndexId      : Word
  ): DBIResult;
```

We've come across `hDb` and `hCursor` before, as well as `pszTableName` and `pszDriverType`, which can both be passed as `nil` if `hCursor` is passed. This makes sense as `hCursor` obviously identifies the table in question. The index name, index tag name and index ID parameters identify the index to be rebuilt. The index name or index ID identifies a Paradox index, the index tag name identifies a dBase index.

So how do we determine the parameters needed to call this function? Well the first two are easy enough, we can pass both the table name and driver type as `nil` and the index name (or tag name for dBase tables) we can retrieve from the relevant `TIndexDef` of the `TTable`'s `IndexDefs` property. But what about the index ID? We can't use the array position of the specified `TIndexDef` in the `IndexDefs` property because that's the index sequence number, not the index ID. But surely if we've identified the index using the index name or tag name we don't care what we pass as `iIndexID`? That's true, but we can simplify matters by calling the `DbiGetIndexDesc` function to retrieve the index properties of an index specified by a particular sequence number:

```
function DbiGetIndexDesc(
  { Cursor handle: }
  hCursor     : hDBICur;
  { Index number: }
  iIndexSeqNo : Word;
  {Returned index description:}
  var idxDesc : IDXDesc
  ): DBIResult;
```

The index properties are returned in the index descriptor idxDesc of type IDXDesc, shown in Listing 2, from which we retrieve the index name, index tag name and index ID to use with DbiRegenIndex without worrying whether we should pass the index name, tag name or ID based on the table type.

Both DbiRegenIndexes and DbiRegenIndex require that the table is opened exclusively. SQL table indexes cannot be rebuilt using these functions. I've implemented three more TjocTable methods for rebuilding indexes: RebuildIndexes, RebuildIndex and RebuildNamedIndex. It's not possible to use RebuildIndex or RebuildNamedIndex to rebuild non-maintained indexes, you must use RebuildIndexes instead.

```
IDXDesc = record
  szName          : DBITBLNAME;    { Index name }
  iIndexId        : Word;          { Index number }
  szTagName       : DBINAME;       { Tag name (for dBASE) }
  szFormat        : DBINAME;       { Optional format (BTREE, HASH etc) }
  bPrimary        : Bool;          { True, if primary index }
  bUnique         : Bool;          { True, if unique keys }
  bDescending     : Bool;          { True, for descending index }
  bMaintained     : Bool;          { True, if maintained index }
  bSubset         : Bool;          { True, if subset index }
  bExpIdx         : Bool;          { True, if expression index }
  iCost           : Word;          { Not used }
  iFldsInKey      : Word;          { Fields in the key (1 for Exp) }
  iKeyLen         : Word;          { Phy Key length in bytes (Key only) }
  bOutofDate      : Bool;          { True, if index out of date }
  iKeyExpType     : Word;          { Key type of Expression }
  aiKeyFld        : DBIKEY;        { Array of field numbers in key }
  szKeyExp        : DBIKEYEXP;     { Key expression }
  szKeyCond       : DBIKEYEXP;     { Subset condition }
  bCaseInsensitive : Bool;         { True, if case insensitive index }
  iBlockSize      : Word;          { Block size in bytes }
  iRestrNum       : Word;          { Restructure number }
  iUnUsed         : array [0..15] of Word;
end;
```

➤ *Listing 2: The index descriptor type*

Other Useful IDAPI Functions

Let's check out some further useful Dbi functions. DbiSaveChanges commits all table changes to disk thus improving the safety of a system prone to power failures etc. But that's not the only good reason to call this function.

Emptying a table using the method TTable.EmptyTable, which calls DbiEmptyTable, actually deletes the table from disk after borrowing the table structure with which a new empty copy of the table is created. The problem is that the new empty table exists only in IDAPI's buffers until they're flushed to disk: that's where DbiSaveChanges comes in. I strongly recommend that this function be called immediately after emptying a table before a system failure occurs and the table is lost. DbiSaveChanges takes a cursor handle as its sole parameter and has been implemented in TjocTable as the Flush method.

Whilst we're on the topic of saving table changes, the function DbiUseIdleTime allows the BDE to carry out background tasks such as flushing its buffers to disk. One dirty buffer (ie the contents of the buffer have changed) is flushed to disk for each call of this function. If there are no dirty buffers to be flushed then the function returns immediately.

DbiUseIdleTime can be called on a TTimer event though the easiest way is to call the function in an Application.OnIdle event handler, although I recommend that you use either DbiSaveChanges or DbiUseIdleTime in your application (not both, as this can lead to excessive disk writes which will slow the system).

We can check to see whether a table is shared by using a call to DbiIsTableShared:

```
function DbiIsTableShared(
  hCursor: hDBICur;
  var bShared: Bool):
  DBIResult;
```

Obviously, if a TTable is opened with Exclusive set to True then the table is not shared.

We can also check the number of cursors open on a particular table using a call to the function DbiGetTableOpenCount:

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

This can be useful for checking if a table is busy before attempting to rename it or restructure it.

Both DbiIsTableShared and DbiGetTableOpenCount have been implemented in TjocTable as the IsShared and OpenCount properties.

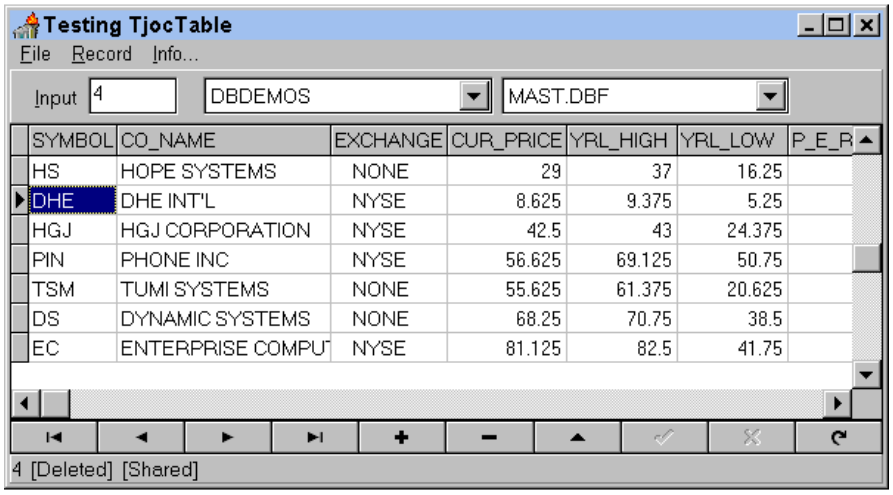
Finally, for all you client/server developers out there who might be feeling a bit left out because this article has so far only discussed local table issues, the function DbiGetTranInfo can tell us if a database transaction is currently active:

```
function DbiGetTranInfo(
  { Database handle: }
  hDb    : hDBIDb;
  { Transaction handle: }
  hXact  : hDBIXact;
  { Transaction info: }
  pxInfo : pXInfo
  ): DBIResult;
```

The transaction info record is of type XInfo:

```
XInfo = record
 exState : eXState; {xsActive, xsInactive}
 eXIL    : eXILType; {Xact isolation level}
 uNests  : Word;    { Xact children }
end;
```

Because IDAPI doesn't support nested transactions we're not required to pass a transaction handle, just the database handle, in a call to DbiGetTranInfo. The



➤ *Figure 1: The TESTJTAB application*

`TransActive` function implemented in TJTABLE.PAS (on the disk) determines whether a `TDatabase` has an active transaction.

The TESTJTAB demo application (Figure 1 and excerpts in Listing 3) shows off `TjocTable`'s capabilities. Any table can be opened by selecting from a combination of the database and table names drop-down lists. You can enter a record number to move to, or the number of records to move by into the `Input` edit box. The status bar displays the record number and various other information about the current table/record such as whether the table is shared or the record deleted. For SQL tables a `Transaction` menu appears, from which transactions can be started, committed or rolled back; the current transaction state is displayed in the status bar. The `TjocTable` source is on the disk too of course.

➤ *Facing Page*
Listing 3: Testjtab source code

In the next article we'll look at the relatively undocumented topic of record and table locking, as well as in-memory tables and their uses.

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

```
unit Jtabtst;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages,
  Classes, Graphics, Controls, Forms,
  Dialogs, ExtCtrls, DBCtrls, Grids, DBGrids,
  DB, DBTables, Menus, StdCtrls,
  TJTable, DbiTypes, DbiProcs;
type
  TMainForm = class(TForm)
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    DBNavigator1: TDBNavigator;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Showdeleted1: TMenuItem;
    Moveby1: TMenuItem;
    Gotorecord1: TMenuItem;
    Undelete1: TMenuItem;
    Toolbar: TPanel;
    Edit1: TEdit;
    Label1: TLabel;
    Tables: TComboBox;
    Databases: TComboBox;
    Msg: TPanel;
    Table1: TjocTable;

    ...
      SEE THIS MONTH'S DISK FOR LINES OMMITTED
      HERE DUE TO LACK OF SPACE
    ...

  private
    { private declarations }
  public
    { public declarations }
  end;
var
  MainForm: TMainForm;
implementation
{$R *.DFM}
procedure TMainForm.Undelete1Click(Sender: TObject);
begin
  Table1.UndeleteRecord;
end;
procedure TMainForm.Moveby1Click(Sender: TObject);
begin
  Table1.MoveRelative(StrToInt(Edit1.Text));
end;
procedure TMainForm.Gotorecord1Click(Sender: TObject);
begin
  Table1.GotoRecord(StrToInt(Edit1.Text));
end;
procedure TMainForm.Showdeleted1Click(Sender: TObject);
begin
  ShowDeleted1.Checked := not ShowDeleted1.Checked;
  Table1.ShowDeleted := ShowDeleted1.Checked;
end;
procedure TMainForm.Exit1Click(Sender: TObject);
begin
  Close;
end;
procedure TMainForm.DataSource1DataChange(
  Sender: TObject; Field: TField);
begin
  with Table1 do begin
    Msg.Caption := IntToStr(RecNo) + ' ';
    if Deleted then
      Msg.Caption := Msg.Caption + ' [Deleted] ';
    if IsShared then
      Msg.Caption := Msg.Caption + ' [Shared] ';
    if TransActive(Table1.Database) then
      Msg.Caption := Msg.Caption + '[Transaction
active]';
  end;
end;
procedure TMainForm.FormCreate(Sender: TObject);
begin
  Session.GetAliasNames(Databases.Items);
  Table1.Open;
  Application.OnIdle := DoIdle;
  Transaction1.Visible := Table1.Database.IsSQLBased;
end;
procedure TMainForm.TablesChange(Sender: TObject);
begin
  Table1.Close;
  Table1.DatabaseName := Databases.Text;
  Table1.TableName    := Tables.Text;
  Table1.Open;
  Transaction1.Visible := Table1.Database.IsSQLBased;
end;
procedure TMainForm.DatabasesChange(Sender: TObject);
begin
  Session.GetTableNames(
    Databases.Text, '', True, False, Tables.Items);
end;
procedure TMainForm.Packtable1Click(Sender: TObject);
begin
  Table1.Close;
  Table1.Exclusive := True;
  Table1.Open;
  Table1.Pack;
end;
procedure TMainForm.Copytable1Click(Sender: TObject);
begin
  Table1.Close;
  Table1.Exclusive := True;
  Table1.Open;
  Table1.CopyTable('DAVEY');
end;
procedure TMainForm.Renametable1Click(Sender: TObject);
begin
  Table1.Close;
  Table1.Exclusive := True;
  Table1.Open;
  Table1.RenameTable('COPIED');
end;
procedure TMainForm.Savechanges1Click(Sender: TObject);
begin
  Table1.Flush;
end;
procedure TMainForm.Info1Click(Sender: TObject);
var
  MsgText: string;
begin
  with Table1 do begin
    MsgText :=
      'Open cursors = ' + IntToStr(OpenCount) + #10 +
      'Table level  = ' + IntToStr(TableLevel)  + #10 +
      'Block size   = ' + IntToStr(BlockSize)+'K'#10#10+
      'Table has been restructured ' +
      IntToStr(RestructVersion) + ' times';
    if IsProtected then
      MsgText := MsgText +
      #10#10'Table is protected and has ' +
      IntToStr(PasswordCount) + ' auxiliary passwords';
  end;
  MessageDlg(MsgText, mtInformation, [mbOK], 0);
end;
procedure TMainForm.DoIdle(
  Sender: TObject; var Done: Boolean);
begin
  DbiUseIdleTime;
end;
procedure TMainForm.Begin1Click(Sender: TObject);
begin
  Table1.Database.StartTransaction;
  if TransActive(Table1.Database) then
    Msg.Caption := Msg.Caption + '[Transaction active]';
end;
procedure TMainForm.Commit1Click(Sender: TObject);
begin
  Table1.Database.Commit;
  Table1.Refresh;
end;
procedure TMainForm.Rollback1Click(Sender: TObject);
begin
  Table1.Database.Rollback;
  Table1.Refresh;
end;
end.
```