The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project?
Just email Brian Long, our Delphi Clinic Editor, on 76004.3437@compuserve.com or write/fax us at The Delphi Magazine

Paradox Tables On A CD

Q I have an application with associated Paradox tables that I have burnt onto a CD. If I put the CD into a network CD drive, the program doesn't run properly: IDAPI tries to write lock files onto the (obviously) read-only device. Can I stop this?

A Borland also fell foul of this problem. The old BDE standalone package came on a CD and also exhibited these symptoms. What you need to do is apply a directory lock to the directory that houses the files and *then* copy all the files, including locks, onto the CD. The question then is how to apply a directory lock. If you look in the directory \RUNIMAGE\DELPHI\DEMOS\DATA on your Delphi CD you'll notice the two lock files: PARADOX.LCK and PDOXUSRS.LCK.

Record locks are applied when calling `TTable.Edit` and table locks are applied by `TTable.LockTable`, but Delphi offers no help for directory locks. You need to ask IDAPI to put a persistent lock on a non-existent table called PARADOX.DRO. This can only be done if local share has been turned on in the BDE configuration program. The code to do this is only one line, if you have access to the database handle (as opposed to a table handle). To demonstrate its use, there is an example program on the disk called DIRLOCK.DPR, which takes rather more than one line. It caters for a number of things: real database aliases, temporary database aliases and Delphi 2 multiple sessions (temporary aliases can be made in any of a number of sessions). See Figure 1.

At start-up, the program ensures that local share is on with a call to `DbiGetSysConfig` and warns if it isn't. Also, to ensure successful execution under Delphi 2 it calls `Session.Open` first. In Delphi 1, the single `TSession` component `Session` was automatically opened at start-up in a database program. In Delphi 2, where there can be many sessions, they are opened when required by the database components. If you wish to use IDAPI manually before this time, you need to open a relevant session. See Listing 1.

Multi-Record Object Bias

Q Why can't I place `TDBImage` or `TDBMemo` controls on a Delphi 2 `TDBCtrlGrid`? I notice that the front cover of Issue 5 had a picture showing that it can be done.

A Apparently that screenshot came from Borland during the Delphi 2 beta program and the VCL has been through a few changes since then. For an object to work in conjunction with the database control grid, otherwise known as the multi-record object (MRO), it must support replication: the MRO duplicates the object in each of its panes. The two data-aware controls you mention do not have this support. I don't know why this is, as it seems easy to get them working. The two components `TDBRepMemo` and `TDBRepImage` in Listing 2 (REPCTRLS.PAS) show how to achieve this.

A Delphi 2 project on this month's disk, MROPROB.DPR, proves the point (provided you install the components on the component palette) and looks like Figure 1: pretty much the same as the picture on Issue 5's cover.

Incidentally, you do encounter a small problem with the data aware

➤ *Listing 1*

```
{ Initialise BDE in Delphi 2, before using IDAPI. Delphi 1 does this for you }
{$ifdef Win32}
  Session.Open;
{$endif}
  { Check for local share. If not on, raise exception }
  Check(DbiGetSysConfig(Cfg));
  if not Cfg.bLocalShare then
    raise EDatabaseError.Create(
      'Local share must be on for successful directory locking');
...
{ Local share must be on in BDE Config for this to work. To be compatible
with Delphi 2 which can have databases open and defined in multiple
sessions, the session must be passed along. In Delphi 1, this is simply the
Session variable. In Delphi 2 it is the dataset's DBSession property }
procedure DirectoryLock(const DatabaseName: String;
  Session: TSession; LockDir: Boolean);
const
  DirectoryReadOnly = 'Paradox.DRO';
  LockOrRel: array[Boolean] of function(hDb: hDBIDb; pszTblNam,
    pszDrvType: PChar): DBIResult {$ifdef Win32}stdcall{$endif}
    = (DbiRelPersistTableLock, DbiAcqPersistTableLock);
begin
  with Session, OpenDatabase(DatabaseName) do
    try
      Check(LockOrRel[LockDir](Handle, DirectoryReadOnly, szParadox));
    finally
      CloseDatabase(FindDatabase(DatabaseName));
    end;
end;
```

memo component on a data control grid. Normally, `Enter` and `Ctrl+Enter` are taken by the memo as a new line character. When placed on an MRO, the grid intercepts the `Enter` key and takes it as a toggle for editing the dataset. To stop it doing this for the memo control requires making a new component based on the `TDBCtrlGrid` and writing a `cm_ChildKey` message handler. Code for this is also in REPCTRLS.PAS, and is shown in Listing 3.

Copying Tables

Q When copying a table using the `TBatchMove` component, or a table's `BatchMove` method, no indexes are made on the destination table, even when some are present on the source table. How do I create matching indexes when batch moving?

A You can make use of methods and properties of the table to find what indexes are on the source and create them on the target. Listing 4 shows three subroutines that can be used to copy a table, a table's structure (fields and indexes) or just a record. These are used in the COPYING.DPR project on this month's disk.

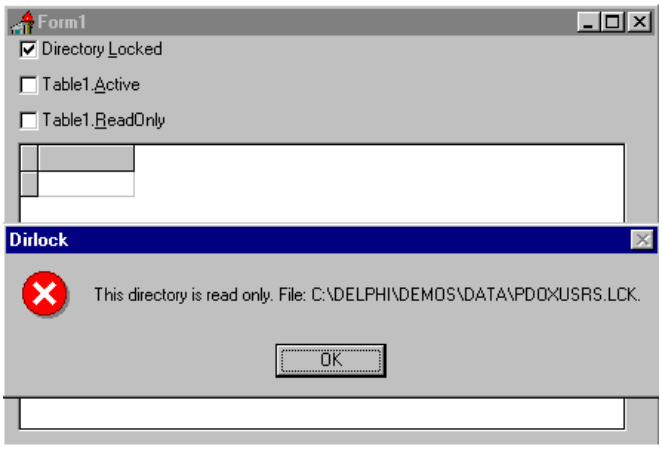
Delphi 2 Is Slooowww

Q How can the same things take much longer time to compile in Delphi 2 than in Delphi 1? Borland say the new 32-bit Delphi 2 runs much faster than the 16-bit Delphi 1. Also, why does the MASTAPP.DPR demo project take 5 seconds to start from Delphi 1 and 10 seconds from Delphi 2? I have plenty of memory to spare, I have 32Mb on a 133 MHz Pentium.

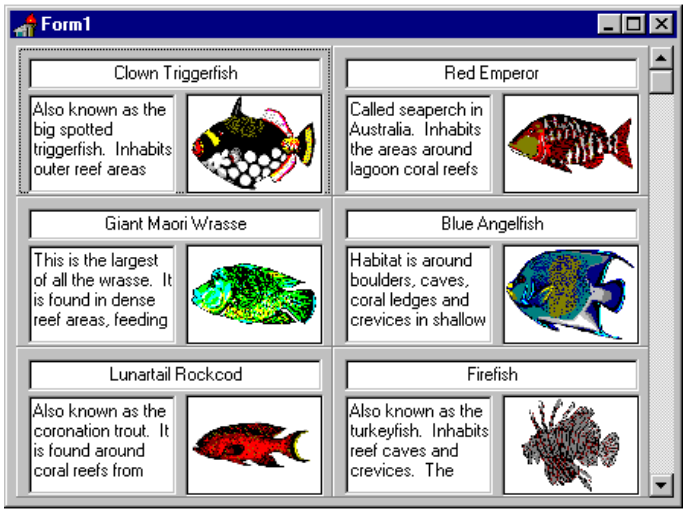
A *We thought this was one for Borland, so the response below is based on information from Danny Thorpe of Borland USA:*

There are a number of questions here, and a number of reasons to go through. Lets start with the compilation speed first of all. The 32 bit DCU format is more version-resilient than the 16 bit format.

➤ *Figure 1: Trying to open a table in a read-only directory*



➤ *Figure 2*




```
type
  TDBRepMemo = class(TDBMemo)
  public
    constructor Create(AOwner: TComponent); override;
  end;
  TDBRepImage = class(TDBImage)
  public
    constructor Create(AOwner: TComponent); override;
  end;
constructor TDBRepMemo.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  ControlStyle := ControlStyle + [csReplicatable];
end;
constructor TDBRepImage.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  ControlStyle := ControlStyle + [csReplicatable];
end;
```

➤ *Listing 2*

```
type
  TDBRepCtrlGrid = class(TDBCtrlGrid)
  protected
    procedure CMChildKey(var Msg: TWMKeyDown); message cm_ChildKey;
  end;
procedure TDBRepCtrlGrid.CMChildKey(var Msg: TWMKeyDown);
begin
  if not ((Msg.CharCode = vk_Return) and
    (Screen.ActiveForm.ActiveControl is TDBMemo)) then
    inherited;
end;
```

➤ *Listing 3*

This means that 32 bit DCUs distributed without source code are much less likely to require recompilation to be used with subsequent versions of the Delphi compiler. Basically, 32 bit DCUs are version checked on a symbol by symbol basis. If Unit A uses function X of unit B, and you modify the declaration of function X, ie the parameter list, Unit A will need to be re-compiled. However, unlike 16 bit Delphi, modifying interfaced symbols of Unit B which Unit A does not use (directly or indirectly) will not require Unit A to be re-compiled.

This will come as a major relief to users of previous Borland Pascal products. As these developers will know from bitter experience, if they bought a third-party package that came without unit source code and Borland then released a new version of the product, they had to purchase a new version of the package.

DCU Format

To accomplish this improved situation, the 32 bit compiler has to unpack the DCUs to find the symbol information stored within. In 16 bits, the DCU was pretty much a snapshot of the compiler's internal memory structures for a particular unit. The 16 bit compiler basically just loaded the DCU into memory, added it to a linked list, and moved on to the next unit. The 32 bit compiler, however, has to unpack the DCU, which is a slight performance hit compared to 16 bit, but which insulates the DCU file format from internal compiler structure changes.

However, this performance hit is felt only the first time the DCU is loaded into memory. The 32 bit IDE caches DCUs in memory to avoid incurring the DCU load cost. This means the second compile of a project in the 32 bit IDE runs almost entirely in memory – and it is noticeably faster than 16 bit Delphi. The first compile of a project in Delphi 2.0 will be slightly slower than Delphi 1.0, but the second and subsequent compiles will be faster than Delphi 1.0. This trade-off of performance, memory, and DCU flexibility fits well with the Delphi RAD model of iterative or spiral design: design a little piece, implement the little piece, compile, test, and repeat for next piece. As opposed to the waterfall model, which says the entire product is designed to completion, then the entire product is implemented to completion, then the entire product is tested to completion, which is non-iterative.

Testing this on my PC, I find this to be the case. Building the MastApp demo project takes approximately 4 seconds in Delphi 1. Subsequent builds take 3 seconds, which can be accounted for by my disk cache. Delphi 2 takes approximately 5 seconds for the first build, but 2 seconds for subsequent builds.

Memory Use

The Delphi 2.0 compiler is rather more aggressive in its use of memory than the 16 bit compiler. The 16 bit compiler was originally designed to work on machines with 64Kb of RAM. Even though that 16 bit architecture grew to support 640Kb RAM, then multimegabytes of RAM, it retained many of the feature limitations of the original 64Kb memory architecture: first and foremost, it lacked full code optimisations.

The 32 bit compiler trades memory thriftiness for outright performance, and that translates into more features for the same compile speed. The 32 bit compiler's internal structures are laid out to favour speed over data density. That speed gain is large enough to allow the 32 bit compiler to compile the source code into intermediate code, optimise the intermediate code, and emit the optimised machine code in less time than the 16 bit compiler compiles source into machine code with fewer optimisations.

16 bit compiler support for code optimisations comparable to the 32 bit compiler would make the 16 bit compiler two to three times slower. Basically, the move to 32 bit instructions and the generous Win32 memory model gave Borland full code optimisations for free, with room to spare. I wonder what they'll do with that in the fullness of time *[Would you like to give us some ideas, Danny?! Editor]*.

When compiling large projects with Delphi 2.0, the amount of free, available physical RAM you should have available for the compiler is about 4 times the size of the final EXE image. To compile a 1Mb EXE at top speed (that is, without hitting the swap file), you should have 4Mb of RAM free and clear (in addition to what is required by the operating system, other apps and the IDE). To compile a 10Mb EXE at 350,000 lines per minute (and it has been done – those figures aren't dreamt up by a Borland marketing person), you'll need 40Mb of physical RAM. Another way to calculate this is to add up the file sizes of all the DCU files that make up the project, since that's what you want to keep in memory.

Windows 95 or NT?

When running under Windows 95, the intrusive swap file management definitely has a negative impact on Delphi 2's performance. Windows NT's swap file management is much smoother. You never see NT grinding the hard disk after the machine has been idle for 5 seconds. Windows 95 is also prone to false out of memory errors if its swap file grows to fill the hard disk completely before the machine has been idle long enough for the Windows 95 garbage collector to kick in. Additionally, Windows 95's virtual memory performance is noticeably slower if you configure it to have a fixed size swap file (instead of allowing Windows 95 to dynamically grow the swap file as needed), so *don't* use a fixed-size swap file.

In general, Windows 95 feels snappier than Windows NT 3.51, primarily because of its lower system overhead (the lack of security code and the thin process isolation) and faster graphics. Windows NT, on the other hand, makes better use of large memory machines (32Mb plus), multiple processors, and large hard disks (NTFS rules the land of 1Gb plus disks). And Windows NT is about as close to

unflappable as one can get in Windows – very important for a development machine. Apparently, MicroSoft is re-configuring the graphics system for Windows NT 4.0 to address the graphics performance bottleneck.

Even though Windows 95 tends to run generally quicker than Windows NT, there are aspects of the system architecture that play an important role in slowing down a 32 bit program.

Despite the quicker graphics system, when manipulating Windows interface elements such as listboxes and memos, your code will suddenly get much slower than in 16 bit Windows. This is because all the controls that are common to 16 bit and 32 bit are implemented in 16 bit code. 32 bit applications that manipulate these controls call Windows code that thunks down to the 16 bit stuff in order to run.

Consider a list box with a lot of text in it. If you set the `Sorted` property to `True`, it will take longer for a 32 bit app running on Windows 95 to do it than a 16 bit app running on Windows 95. The figures on my machine were 0.49 second for the 32 bit app and 0.38 second for the 16 bit application. Running the application on Windows NT would give a better time than either of these.

If you are writing code that does a lot of control manipulation, consider doing it yourself, rather than letting Windows do it. For example, if you use a `TStringList` object and sort that, it took 0.11 seconds for my PC to sort it in 32 bit, and 0.2 second in 16 bit.

Borland Database Engine

The Borland Database Engine (BDE) does do a lot of set-up work when the first application loads BDE: loading support DLLs, initialising cache memory, detecting and initialising network support, etc. And the new 32-bit BDE does rather more than the 16-bit BDE.

Nevertheless, BDE 3 is noticeably slower at loading than BDE 2.5. That said, there are several things which help speed up database applications. The ODBC socket in BDE 3 is much better, helped by 32 bit ODBC being rather faster. Cached updates also help out.

Incidentally, the BDE 3 implements some new caching techniques similar to the IDE's unit caching which make subsequent queries against Paradox and dBASE tables many times faster than the first query. The biggest difference will be seen with large tables and complex queries.

Acknowledgements

Thanks to Roy Nelson at Borland for the basis of the Paradox directory lock technique. Also thanks to Borland's Danny Thorpe for the low down on the 32-bit compiler's operation.