Inside The BDE: Building A Data Dictionary

by Brandon Smith

The Borland Database Engine (BDE) is a mysterious beast, even after you shell out the extra time and money to obtain the manual. Information hiding is one of those concepts whose time has come, and the Delphi approach to database programming, just like Wirth's original definition of the Pascal language, is forcing good programming habits on us.

The purpose of this article and the sample programs is to show you a little of what can be achieved using the BDE directly with Delphi to create database applications. The example I've chosen is a data dictionary application, which I hope you will find useful as well as informative. Note that whilst much of what I discuss *may* apply to Delphi 2, I've written it from the perspective of a Delphi 1 user.

Records And Fields

While one can, with some effort, justify a need to work with a record number in Delphi, I've found that it really isn't that much of a loss. Speaking as an ex-dBase programmer, I like to know which record number I'm seeing, but my customers generally don't care. Their attention is on the data, not its physical location in the table.

By focusing on the tools that Delphi does provide, my attention is also on the data – the table, the query and (most importantly) the *field definitions*. These funny animals, TIntegerField, TBlobField and so forth, are the key to database programming in Delphi.

But if TField is the key, why do you have to go through hoops to get at the field definitions? Why aren't they simply built into visual components like TDBedit? Perhaps the answer partly lies in another question. How would you like a huge component palette that listed TDBIntegerEdit, TDBSmallIntEdit, TDBWordEdit and so on? A separate icon for each kind of visual representation for each kind of field? This is a valid approach, but Delphi is already a monster package! Even Borland cautions that data-aware components use considerably more resources than the other kinds.

Another part of the reason why the TFields are not present on the Component Palette is that they serve as the buffer between an application or form and the BDE. On the BDE side, the TField is initialized when you open a table or query with an "accurate" definition of what the physical characteristics of the data is. On the application side, the TField provides the programmer with validation properties such as minimum and maximum values.

Why the scare quotes around "accurate"? Because the BDE has its own notions about what the data is. For example, suppose you had an old dBase file lying around with a field called SIZE and that when you look at it with the dBase LIST STRUCTURE command you see that this field is a number, length 10, decimals 0. Now move over to Delphi, drop a table on a form, link the table to that old dBase file, double click on the table, add all the fields, then go to the Object Inspector and you find Delphi thinks SIZE is a TfloatField!

At first glance, this is not good at all. However, what did length 10 mean in dBase? It meant that the valid range was 0..999999999. In the old dBase, if you had actually wanted the valid range to be 0..1111111111, then you would have to add in additional code to validate user input. In Delphi, you simply fill in the minimum and maximum properties of this TfloatField. Of course, you'll have to go through some effort to make the error message presented to the user say something different from the Delphi default. However, I've found most of Delphi's default error messages perfectly clear.

Finding Field Definitions

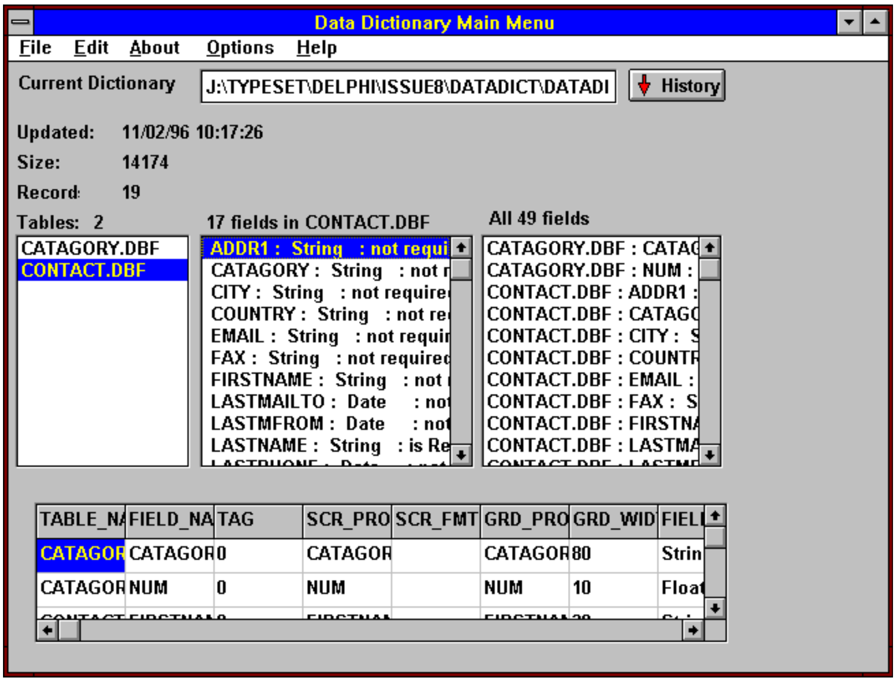
What's not clear, though, is how to easily get at the various definitions stored in TFields. You can't get to them with a simple click on a visual component. If, part way through development, you have to change some TField properties, you have to go looking for them. Also, although Delphi will let you just define some of the TFields, the manuals caution against it. Either instantiate all the TFields 'for a particular table or let Delphi create them at runtime and do your own validation in code. You can also create TField definitions on the fly, but I've found this doesn't seem to work right unless you create them all. However, what you can do, after a table is opened, is set TField properties on the fly. For example:

```
MyTable.FindField(
  'LASTNAME').required := true;
```

This will set the required property to true and generate a Delphi warning if a user tries to post a record in which this field is blank. Since FindField returns a TField and required is a property of TField, no problem. However, when you get to MinValue and MaxValue, properties of each of the numeric TFields, you will have to typecast to the appropriate kind of TField:

```
TFloatField(MyTable.FindField(
  'SIZE')).MinValue := 4;
```

This statement will work, but is also very dangerous to use. If the BDE decided SIZE was a TInteger when it opened the table, then when Delphi gets to this statement,



Current Dictionary J:\TYPESET\DELPHI\ISSUE8\DATADICT\DATADI

Updated: 11/02/96 10:17:26
Size: 14174
Record: 19

Tables: 2

CATAGORY.DBF
CONTACT.DBF

17 fields in CONTACT.DBF

ADDR1 : String : not requi
CATAGORY : String : not r
CITY : String : not require
COUNTRY : String : not re
EMAIL : String : not requir
FAX : String : not required
FIRSTNAME : String : not
LASTMAILTO : Date : not
LASTMFROM : Date : not
LASTNAME : String : is Re

All 49 fields

CATAGORY.DBF : CATAG
CATAGORY.DBF : NUM :
CONTACT.DBF : ADDR1 :
CONTACT.DBF : CATAGO
CONTACT.DBF : CITY : S
CONTACT.DBF : COUNTR
CONTACT.DBF : EMAIL :
CONTACT.DBF : FAX : S
CONTACT.DBF : FIRSTNA
CONTACT.DBF : LASTMA

TABLE_NA	FIELD_NA	TAG	SCR_PRO	SCR_FMT	GRD_PRO	GRD_WID	FIELI
CATAGOR	CATAGOR	0	CATAGOR		CATAGOR	80	Strin
CATAGOR	NUM	0	NUM		NUM	10	Float

➤ *Figure 1: The DDICT data dictionary in action*

it invalidates the whole table. Even when it does work, you have to be sure to set both minimum and maximum, otherwise your users will get an error message.

Rather than go into how we could use is, as and the case structure to get around this, let's back up and take a wider view of the problem. What are we using these TFields for? To define the characteristics of the data elements in our database tables. In other words, a *data dictionary*.

Data Dictionaries

Delphi Developer 2 has a data dictionary which allows you to specify things like minimum and maximum values at design time (at the time of writing Delphi 2 has not been released so I don't know what's in it yet). Great, but what about all of us who are still happily using Delphi 1? I decided to implement what I'd call a "real" data dictionary for use with Delphi 1.

Well, what is a "real" data dictionary? When I hear the term, I think back to my mainframe days and the thick stacks of paper that were usually called "The Data Element Dictionary." Each of the hundreds of pages contained a single data element definition where the data element name, its type, its size and various other pieces of information were presented. "*What a waste of paper!*" I hear you mumble, especially when you find out that many of the pages were more than half blank! After all, what more can you say about SIZE other than it is a number limited to the range 4..20?

Actually, quite a bit, when you stop to think about it. In particular, you can describe just exactly what the number stored in this field represents in the real world, why you want it stored in the computer and what you plan to do with it, what name you want the user to see, what hint string you want to display for that field, how it relates to other fields, and so on.

"Oh, that's just documentation. Boring, really, let someone else do it. I've got code to write. Just hand me the dictionary and I'll prop it up next to my screen and make sure the validation code works."

But what if the dictionary comes to you as a database table, that you can simply call on to set up the TFields for you? Now we have a bit more than boring old documentation. Now we have documentation that feeds directly into the business of making code. This is what I think of as a *real data dictionary*.

The remainder of this article describes two applications. The first, DDICT.DPR, is the data dictionary itself, the second, CONTACTS.DPR, is a sample application which uses a table generated by the data dictionary to modify its TFields on the fly. The unit DBUTILS.PAS is common to both programs and is set up as a component, though not used that way in this example. The DictCtrl class contains the pieces that build the dictionary as well as the pieces that enable modifying TFields. In order to keep down the BDE overhead, when I grab the data dictionary information for the target application I pull it into a TStringGrid and work on it from there.

The DDICT Project

DDICT.DPR, the data dictionary itself (see Figure 1), generates a dBase table which contains the data element definitions. Generating an empty table is not difficult, though working with the BDE alias scheme is a pain – more about that later. The code fragment in Listing 1 (over the page) illustrates how to generate a dBase table on the fly (the full source code is included on this month's disk of course).

Once the empty data dictionary table is created, we could use the dictionary input form to create each of the fields. However, I find it easier to use Delphi 1's Database Desktop to do the initial build of my actual application. Another reason to use the Database Desktop to do the initial build is that when I pull in the basic data element information into the data dictionary, I ensure that my dictionary knows what kind of data type the BDE has assigned each field. This way I avoid trying to figure out whether a numeric field with a length of 6 is a TSmallInt, TInteger or TFloat. Pulling this basic information is accomplished by the code illustrated in Listing 2 (more detail on the disk again).

Once we have the basic information about what the BDE thinks is in our table, we use the DDDICT program to add details to the CONTACT data dictionary. For example, we'll add a hint to the field FIRSTNAME (see Figure 2), we'll check off Required for field LASTNAME

```
try
  main.sourceDatabase.close;
  main.SourceDatabase.Params.clear;
  main.SourceDatabase.Params.Add(’PATH=’+fPathName);
  main.SourceDatabase.open;
  with main.DictTable do begin
    active := false;
    databasename := main.SourceDatabase.databasename;
    tablename := fTableName;
    tabletype := ttdBase;
    with FieldDefs do begin
      clear;
      Add(’TABLE_NAME’, ftString, 20, false);
      Add(’FIELD_NAME’, ftstring, 10, false);
      Add(’FIELD_TYPE’, ftstring, 1, false);
      Add(’REQUIRED’,   ftBoolean,  0, false);
      { ... more fields as needed }
    end;
    createTable;
    Result := true;
  end;
except
  on EdatabaseError do begin
    MessageDlg(’Error attempting to create DD file.’,
      mtInformation, [mbOK], 0);
    Result := false;
  end;
end;
```

➤ *Listing 1*

```
var
  tmpint, thisfield : integer;
  tmpstr : string;
  FromField : tField;
begin
  try
    if  {both the dictionary table and the table we are importing from are
    opened successfully} then begin
      with DictTable do begin
        for thisfield := 1 to FromTable.fieldCount -1 do begin
          append;
          findfield(’Table_name’).text := FromTablename;
          findField(’Field_name’).text :=
            FromTable.fields[thisfield].fieldname;
          findField(’Field_type’).text :=
            FieldTypeStr[FromTable.fields[thisfield].datatype];
          FromField := FromTable.fields[thisfield];
          tStringField(findField(’Scr_prompt’)).value :=
            FromField.DisplayName;
          tbooleanField(FindField(’Required’)).value := Fromfield.Required;
          { the rest of the fields ....}
          post;
        end;
      end;
      result := true;
    end else begin
      result := false;
      exit;
    end;
  except
    { some error occurred while trying to import the field information}
    on EdataBaseError do begin
      screen.cursor := crDefault;
      MessageDlg(’DB error while reading field info...’,
        mtInformation, [mbOK], 0);
      result := false;
    end;
 end;
end;
```

➤ *Listing 2*

(Figure 3), and we’ll set a minimum and maximum for NumMailTo. Now, after saving the dictionary database, all we have to do in the Contact program is call on DictCtrl to set the appropriate properties at run time, as shown in Listing 3.

The choice of when and where to open the data dictionary depends on your specific application. In this case, doing all the work in the Activate phase makes sense if you need to re-open the table which the dictionary applies to each time the form is activated. And, of course, a real application would have try..except and try..finally blocks in addition to the basic error handling shown here.

In addition, the code illustrated includes a couple of typecasts that are not good programming practice. In the code on the disk, in DBUTILS.PAS, you’ll find an initial draft of a procedure called SetUpTable that uses a number of case statements to sort out the typecasting in a safe manner.

However, even if we keep ourselves straight on typecasting, there is the rather sticky mess represented by the data type mapping tables shown on pages 90 and 91 of the Delphi 1 *Database Application Developer’s Guide* which is expanded to six pages (99 to 104) in the Delphi 1 *BDE User’s Guide* where it is called *Data type translations and IDAPI logical types and driver-specific physical types*. Where this mess reaches out and grabs you is when you build an empty table based on the definitions you’ve got in your dictionary. Suppose we have a field called NumHits that we want to be a large number, so we assigned it a length of 11 when we initially built the dBase table using the Database Desktop. When we pull the structure into our data dictionary and started editing, we notice that the BDE has turned this into TFloatType. But we know that NumHits will always be a whole number, so we change the type by clicking on Word in the data types list.

When we try to build an empty table, however, we get the BDE error *“Capability not supported”*. Unfortunately, we don’t get this when the program executes in the place where the FieldDef states that NumHits will be ftWord. Instead, the error occurs when BDE attempts to execute TTable.Create. Looking through these tables, I failed to find Word anywhere. There’s Short, Long, Integer, Float and Number among other

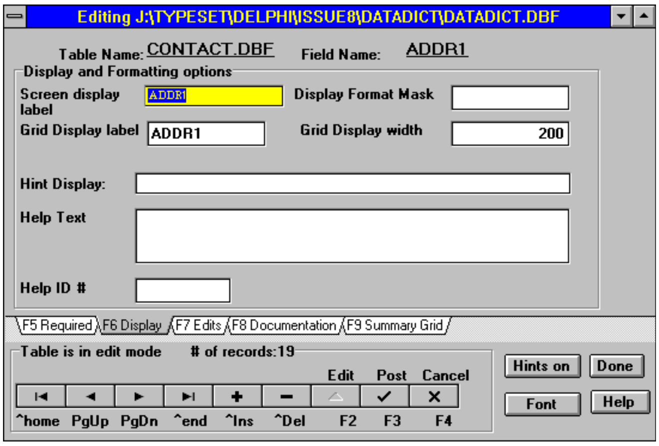

Editing J:\TYPESET\DELPHI\ISSUE8\DATADICT\DATADICT.DBF

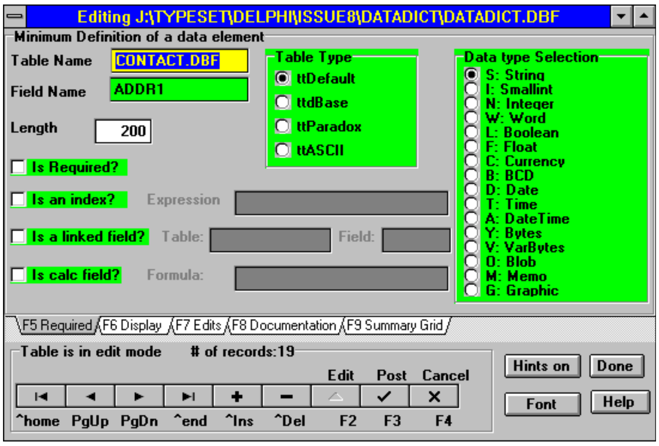
Table Name: CONTACT.DBF   Field Name: ADDR1

Display and Formatting options

Screen display label: ADDR1
Display Format Mask:
Grid Display label: ADDR1
Grid Display width: 200
Hint Display:
Help Text:
Help ID #:

Table is in edit mode   # of records:19

➤ *Figure 2: Setting display, hint and help options*


Editing J:\TYPESET\DELPHI\ISSUE8\DATADICT\DATADICT.DBF

Minimum Definition of a data element

Table Name: CONTACT.DBF
Field Name: ADDR1
Length: 200

□ Is Required?
□ Is an index? Expression:
□ Is a linked field? Table: Field:
□ Is calc field? Formula:

Table Type
● ttDefault
○ ttdBase
○ ttParadox
○ ttASCII

Data type Selection
● S: String
○ I: SmallInt
○ N: Integer
○ W: Word
○ L: Boolean
○ F: Float
○ C: Currency
○ B: BCD
○ D: Date
○ T: Time
○ A: DateTime
○ Y: Bytes
○ V: VarBytes
○ O: Blob
○ M: Memo
○ G: Graphic

Table is in edit mode   # of records:19

➤ *Figure 3: Defining data elements*

```
procedure TInputDBForm.FormActivate(Sender: TObject);
begin
  FdataDictName := AddBackSlash(extractfilePath(application.exename))+
     'datadict.dbf';
  DictCtrl.OpenDictionary(FdataDictName, MyDataBase, DictTable, MyQuery,
     MyDataSource);
  if openDB(MyDataBase, myTable, myQuery, myDataSource,
    ExtractFilePath(application.exename), 'contact.dbf') then begin
    MyDataSource.enabled := true;
    MyTable.open;
    DictCtrl.SetCurrentFieldTo('Contact.dbf','FirstName');
    EditFirstName.hint := DictCtrl.hint;
    ShowHint := true;
    DictCtrl.SetCurrentFieldTo('Contact.dbf','LastName');
    MyTable.findfield('LASTNAME').required := DictCtrl.required;
    DictCtrl.setCurrentFieldTo('contact.dbf','NumMailTo');
    TSmallIntField(MyTable.findfield('NumMailTo')).minvalue :=
      DictCtrl.minValue;
    TSmallIntField(MyTable.findfield('NumMailTo')).maxvalue :=
      DictCtrl.maxValue;
    MyTable.edit;
  end else begin
    messagedlg('Problem opening database', mtinformation, [mbOK],0);
  end;
end;
```

➤ *Listing 3*

variations of expressing a number. Apparently, TWordField is strictly to be used on the application side of the BDE connection, in other words, only to be used when you create TFields at design time. You get the same effect by using ftFloatType and setting min and max values to 0 and 65,535, a method that would be clearer to the client than trying to explain that Word usually refers to the values expressed by two bytes when the significant bit is not used as a sign bit!

I tend to think the best place to take care of ensuring data types are correct for table types is right in the dictionary editing module. The example of building an empty table in the code on the disk (in the file DBUTILS.PAS) does not sort this out, though it does illustrate how to get the informative *"Capability not supported"* message. Also not implemented in the code on disk is a method for generating a Delphi unit to build the table, since that was included on a disk with this magazine a few issues back *[Issue 5: MAKEDBF, courtesy of Charlotte Gamsu. Editor]*.

Nightmare On Alias Street...

Finally, before I close, I'd like to explain how I avoided dealing with the infamous BDE alias. Perhaps you have no problem with the way Borland set up the alias business and made TDatabase something we're not supposed to have to use. This is all fine and dandy if you are building for yourself on your own machine or you don't mind writing special code to do the BDE configuration at installation, or perhaps giving the user special instructions on how to do that. However, in this data dictionary application, I needed to be able to hop around and grab dictionary data by opening a table in any directory as well as creating dictionaries in any directory and, further, creating tables based on the dictionary in any directory. Being one who digs around, when I found TDatabase I figured I'd found the cure. And it is the cure – just give the database instance a name and ignore the alias field.

The problem with using the alias field of TDatabase is that when you give the database instance an alias name you run the risk of generating spurious errors such as *"Duplicate Alias"*. Fine, I *knew* it was a duplicate. I had just closed the database over in directory X and now I want to open one here in directory Y! *"No, no"*, says the BDE, *"Not allowed"*. You can't have one alias pointing to two different directories in the same program, even if you close everything down using the TSession routines. Once the BDE link is started during a run, or set up via the BDE configuration tool, an alias is locked to whatever it is first set to. OK, so lets try changing the alias name. I don't recall what happened then, but that didn't work either. In the end I used the Object Inspector at design time to give one database the name XXX and the other the name YYY and left the alias name blank. The function in Listing 4 illustrates a generic way of opening a database without involving alias considerations. Note that you do have to have each of these components dropped somewhere in your application and that the alias property of TDatabase must be blank and the database name property of TDatabase must be non-blank. Also, tables created this way will not be listed when you go into the Database Desktop: you'll have to navigate to the directory yourself.

```
function openDB(var whichdb : tdatabase; var whichtable : ttable;
  var whichQuery : tquery; var whichsource : tDataSource;
  const pathname, tablename : string): boolean;
begin
  try
    WhichDB.close;
    WhichDB.Params.clear;
    WhichDB.Params.Add(’PATH=’+PathName);
    WhichDB.open;
    WhichTable.DatabaseName:= WhichDB.databasename;
    WhichTable.tablename := TableName;
    WhichTable.Active:= True;
    WhichSource.DataSet:= WhichTable;
    WhichQuery.databaseName := WhichDB.databasename;
    WhichQuery.dataSource := WhichSource;
    WhichQuery.close;
    WhichQuery.sql.clear;
    WhichQuery.params.clear;
    result := true;
  except
    on EdataBaseError do begin
      screen.cursor := crDefault;
      MessageDlg(’Could not open ’+
        pathname + ’ ’+tablename, mtInformation, [mbOK], 0);
      result := false;
    end;
  end;
end;
```

➤ *Listing 4*

Wrapping Up

In conclusion, I'd like to point out that you could use this approach to set up separate data dictionaries so that, for example, the German edition not only has German prompts, but also generates a table with German table and field names. I also included a memo field for inserting help text with the idea that help files could also be generated on the fly. By centralizing all the data management information in a data dictionary, future code maintenance is made much easier.

Delphi provides the tools to create a very useful and powerful linkage between a formal data dictionary database and the tables needed by the application it applies to. While resource files could probably be used to accomplish nearly the same thing, having the data dictionary be a database itself means that it will be easy to produce readable reports and it will be easy to update both the data dictionary and the supported application. The code presented here is a long way from production, but I hope it will help others who may be looking for these solutions.

Brandon Smith lives and works in Mansfield, MO, USA and can be emailed at Synature@aol.com