Working With The InterBase API

by Paul Reeves

Writing applications in Delphi that directly use the InterBase API has never been straightforward. The API guide was written for C programmers, and all the example code fragments use the procedural programming style. Quite apart from the obvious hurdle of translating C into Pascal is the problem of pulling the disparate examples together into some kind of cohesive framework that could be called a working program.

This article has three aims. First, to give users of the new InterBase Express Components (hereafter IBX) a better appreciation of what is happening under the InterBase hood. Secondly, there are times when programming to the API directly can yield performance benefits: the simple examples supplied should give you enough to experiment with the 'hot spots' in your applications and improve on them. Thirdly, with InterBase being a multi-platform database, the examples have been designed to be portable to run under other flavours of Object Pascal. InterBase API level programming is ideal for non-interactive server-side applications and there is no reason to have to abandon Pascal just because the platform is no longer Windows.

Getting Started

There are typically four main components in what one might loosely term an InterBase application. The client application links with gds32.dll (*one*), which talks over a network connection (*two*) to the InterBase Server (*three*), which provides access to the database file (*four*).

The gds32.dll library represents the totality of that which we can achieve with InterBase. The only way we can talk to our database is through loading a copy of this DLL into our application's memory space and making calls to the functions it exports.

The first step in talking to InterBase is to get a translation of the IBASE.H file. This C header file has three roles: it defines essential data structures, it contains the function prototypes exported from gds32.dll, and it declares hundreds of constants for use with those function calls.

In general, I would recommend using the official ibase.pas interface shipped with the IBX set, as this will always be updated to reflect changes in InterBase. However, the versions that come with FreeIBComponents or IBObjects are just as good to get started with. As there are now so many translations available I am not going to dwell on the effort involved in converting ibase.h to ibase.pas.

```
function isc_attach_database(status_vector: PSTATUS_VECTOR;
  db_name_length: Short; db_name: PChar; db_handle: pisc_db_handle;
  parm_buffer_length: Short; parm_buffer: PChar): ISC_STATUS; stdcall
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
type
  TParamBlock:  array of char[0..1023];
var
  StatusVector: ISC_STATUS_VECTOR;
  DBName: String;
  DBHandle: Tisc_db_handle;
  DPBLen: SmallInt;
  DPB: TParamBlock;
procedure attachdatabase;
begin
  DBName:='c:\data\employee.gdb';
  DBHandle:=Nil;
  {Set DPBLen and DPB here }
  if isc_attach_database(@StatusVector, length(DBName), PChar(DBName), @DBHandle,
    DPBLen, @DPB )<> 0 then
    //DoErrorHandlingStuff
  else
    //continue processing
end;
```

Making A Connection

Before doing anything with our database we must connect to it. This is a relatively simple affair, requiring a call to `isc_attach_database`, see Listing 1.

We need to declare a few variables and then put some values into them.

The `DBHandle` must be assigned `NIL` before we do anything with it. If this handle is global it is good practice to check it's not already in use.

When assigning the value to the `DBName` we must include the name of the server, the connection protocol symbol, the path local to the server and the database filename itself. Typically, using TCP/IP the value in `DBName` might be:

```
DBName :=
  'myserver:c:\data\employee.gdb';
```

with the first : signifying the protocol. If we were connecting using NETBEUI we would use:

```
DBName :=
  '\\myserver\c:\data\employee.gdb';
```

Equally, if you are making a local connection then just leave out the server name and protocol symbols.

The next step is to populate the DPB. The general principles for working with parameter blocks are explained in a sidebar. Essentially the DPB contains a list of all the parameters that will define this attachment to the database. The API guide lists the most important ones. At the very least, we will need to supply a username and password. In the example code I have also shown how to reset the cache size for the database.

We are now ready to make our connection. Listing 2 takes us through the basic process.

Note that we use the @ symbol as a prefix to each variable that is to be passed as a pointer.

If this call executes successfully `DBHandle` will now represent our connection to the database. From here on, almost every API call will require this handle as a parameter.

Starting A Transaction

No data can be written to, or read from, an InterBase database outside of transaction control. Before we can start a transaction we must establish a valid transaction handle. This process is very similar to that of establishing a database handle:

```
function isc_start_multiple(
  status_vector: PSTATUS_VECTOR;
  trans_handle: pisc_tr_handle;
  db_handle_count: Short;
  teb_vector_address: PISC_TEB):
  ISC_STATUS; stdcall;
```

Where a database connection requires us to supply a DPB, a transaction requires us to supply a transaction existence block (TEB). This itself is an array of transaction parameter blocks (TPB). Why so complicated? Well, the API does have an `isc_start_transaction` call, which just takes TPBs, but it takes a variable number of parameters, which most experts I have read seem to think a bad thing for Pascal. So we have the alternative call using a TEB. The TEB is used so that a transaction can be set up spanning multiple databases. For now, though, we will just concentrate on working with one.

And working with a single database means that we can get away with an empty TEB. By default, if no TPB is set up, InterBase uses default transaction settings. These are an isolation of *repeatable read*, an access mode of *read/write*, and a lock resolution mode of *wait*. This is suitable for just about all our work and, indeed, shouldn't be changed unless you really know what you are doing.

So, the steps to get our transaction handle go roughly along the lines of Listing 3. We need to declare our variables and initialise our TEB to `nil`. If the call to `isc_start_multiple` is successful then we have our transaction handle. That's it. We have a connection to the database and we have started a transaction.

➤ *Listing 3*

```
var
  StatusVector : ISC_STATUS_VECTOR;
  TxnHandle: Tisc_tr_handle;
  TEB: ISC_TEB;
procedure StartTransaction;
begin
  //init TEB
  with TEB do begin
    db_ptr  := @DBhandle;  //previously declared
    tpb_len := 0;
    tpb_ptr := nil;
  end;
  TxnHandle:=Nil;
  isc_start_multiple(@StatusVector, @TxnHandle, 1, @TEB);
end;
```

The Meaty Stuff

Let's move on to getting some data in and out of the database. Overcoming the simple problems of acquiring database and transaction handles should not lull you into thinking IB API programming is easy, even if correctly populating parameter blocks is a bit tricky. Unfortunately, the real challenges still await us.

Doing anything at all useful with a database requires us to send Dynamic SQL (DSQL) statements to the database. We might optionally want these statements to be parameterised, and they may return data back to us (result sets). In addition, we might be using stored procedures.

All of these require slightly different combinations of calls, although we can put most of it together into a series of generic routines that can be called whatever the query type.

There are a number of steps involved in DSQL programming. Table 1 gives us an overview. There is a lot to take in, with many pitfalls along the way.

Preparation	Initialise our data structures for holding query parameters and results. Allocate a statement handle. Prepare the statement. Describe number of parameters. Allocating sufficient memory for parameters. Discover details of result set (Extract the metadata). Allocate sufficient memory to hold a single row.
Execution	Determine query type. Assign parameters for this execution. Choose execution method and do it!
Reading Results	According to query type repeatedly fetch values into the Output DataArea. Process output placing data into persistent storage for further use.

➤ *Table 1*

Preparing An SQL Statement

There are several steps to follow in 'preparing' a statement for execution. I am going to present them in a generic manner. You may be able to leave some steps out, depending on the type of statement.

Although they will be reset later, it is good practice to initialise our `InputDataArea` and `OutputDataArea`. See the sidebar for a fuller explanation of these structures. Between them, they will handle parameterised inputs and the subsequent retrieval of data for `Select` statements. To start with, we will set them to contain a single column and we will zero the memory allocated to them: see Listing 4.

There is one important point to bear in mind here. The API guide uses malloc() for memory allocation. ReallocMem is not quite a functional equivalent of this. It took me a long, painful debug session to discover that malloc() zeroes memory automatically (in fact, it was only when I carefully examined the code for FreeIBComponents that I fully realised this). You have to explicitly do this in Pascal, otherwise your queries will fail for no apparent reason.

We now need to initialise a statement handle. To do this we need to declare one:

```
var
  StmtHandle: Tisc_stmt_handle;
```

and then make this call:

```
isc_dsql_allocate_statement(
  @StatusVector,@DBHandle,
  @StmtHandle);
```

With a valid statement handle we can quickly move on to preparing the SQL statement. To do this we can take an SQL statement and cast it to a PChar within the call itself:

```
isc_dsql_prepare(@StatusVector,
  @TxnHandle,@StmtHandle,0,
  PChar(SQLString),1,
  OutputDataArea);
```

By default we will pass our OutputDataArea. If you know that your statement will not return a result you could pass NIL here.

We can ignore the constant values supplied to the prepare function: they never change under InterBase v4 or v5. However, you should always check this when a new version is released.

We now need to find out the number of parameters the SQL statement requires. In InterBase, parameters are indicated by the ? token. So a statement such as

```
Select * from employee where
  emp_no = ?;
```

will require a single parameter. Note that parameters are not named and are assigned values in their order of declaration. After calling isc_dsql_describe_bind, passing it the InputDataArea, we will find the sqld field is set with the number of parameters in the statement. We need to then re-initialise the space for the parameters and repeat the describe_bind call. Finally, we need to allocate storage for the parameters themselves. Listing 5 takes us through these steps.

AllocateSQLData is a routine I have written which handles the tricky stuff here. We need to walk through the XSQLDA checking each data type and allocating memory for it, storing the pointer for that memory in the SQLData field of each XSQLVAR.

The above sequence is all perfectly safe, even if there are no parameters. InterBase will set the InputDataArea accordingly. Unless performance really is an issue (perhaps due to a slow network, or when using a dial-up line) it is worth going through this sequence anyway, as we can test the InputDataArea later to see which method of execution is required.

Discovering The Result Set

Having worked out our input requirements we can turn to our result set. When we called isc_dsql_prepare we passed the OutputDataArea. We can now examine the sqld field to see how many columns will be returned.

We then need to resize the data area to hold these columns and call isc_dsql_describe to fill the OutputDataArea with all the information InterBase has on the result set. We finish up by allocating memory for each column. And that is it. We have successfully prepared our statement for execution. See Listing 6 for the details.

Assigning Parameters

If we have a parameterised query we now need to supply a value to each parameter. This is another hurdle for the typical Delphi programmer who, like me, missed out on the pre-Delphi era of Pascal. Parameter assignment is filled with lots of pointer fun. I have simplified it greatly with an AssignParam function in the example application.

Essentially we need to set up a FOR statement that goes through each XSQLVAR. Then, in our AssignParams routine, we test for the data type. Our parameter values will typically be strings so we need to cast them to the appropriate type and then place them in the memory area that is pointed to by the SQLData field. Simple! Listing 7 shows a snippet, for two data types.

The example code covers most of the important data types and should give you enough to manage

```
procedure InitSQLDA(var AXSQLDA: PXSQLDA; Columns: Integer);
begin
  ReallocMem(AXSQLDA, XSQLDA_LENGTH(Columns));
  FillChar(AXSQLDA^, XSQLDA_LENGTH(Columns), #0);
  AXSQLDA^.SQLn := Columns;
  AXSQLDA^.version:=SQLDA_VERSION1;
end;
...
InitSQLDA(InputDataArea,1);
InitSQLDA(OutputDataArea,1);
```

➤ *Above: Listing 4*

➤ *Below: Listing 5*

```
//call describe_bind to find out the number of params
isc_dsql_describe_bind(@StatusVector,@StmtHandle,1,InputDataArea);
//read the number of params in the statement
i:=InPutDataArea^.sqld;
//now reinitialise  space for that many params
InitSQLDA(InPutDataArea,i);
//now repeat bind call to fill the InputDataArea with meta data from InterBase
isc_dsql_describe_bind(@StatusVector,@StmtHandle,1,InputDataArea);
//allocate memory for the params
AllocateSQLData(InPutDataArea);
```

➤ *Listing 6*

```
i:=OutPutDataArea^.sqld;
InitSQLDA(OutPutDataArea,i);
isc_dsql_describe(@StatusVector,@StmtHandle,1,OutPutDataArea);
AllocateSQLData(OutPutDataArea);
```

IB API Programming Conventions: Parameter Blocks And Result Buffers

One difficulty most Delphi programmers will face is the use of arrays of chars to pass information to and from InterBase. These arrays typically hold data in the following format:

isc_constant	1 byte
Length of data to follow	2 bytes
Data to pass in or out	n bytes

It is usually easiest to hard code the memory for these arrays, otherwise memory needs to be allocated and de-allocated on the fly. The space requirements are easy enough to predict, being specific to each function call that uses them.

Let’s take the Database Parameter Block (DPB) as an example of how to work with them. First we need to declare our DPB and some other variables:

```
type
  TParamBlock = array [0..KILOBYTE-1] of Char;  //array of 1024 chars
var
  FDPB: TParamBlock;  //parameter block for database connection
  FDPBLen: Integer;   //number of bytes set in block
  UserStr: String;    //UserName
  Len: Integer        //Length of username
```

and initialise it for use:

```
//init DPB
fillchar(FDPB,sizeof(FDPB),#0);
FDPB[0] := char(isc_dpb_version1);
inc(FDPBLen);
```

Now, we need to add the parameters; the most common are a username and password, which are both strings:

```
FDPB[FDPBLen]:=isc_dpb_user_name;   //assign DPB constant
inc(FDPBLen);                       //move length on by 1 char
len:=char(Length(UserStr));         //get length of username
FDPB[FDPBLen]:=len;                 //write length of user name
inc(FDPBLen);                       //move length on again
StrPCopy(@PB[PBLen],UserStr);       //pop the user name in
inc(FDPBLen,Len);                   //and move the length on again
```

This code sequence needs to be repeated for each parameter passed. Fortunately, it can be simplified greatly. Parameters can only be of three types: `String`, `Integer` or `Boolean`. So we could write something like this to manage all string parameters:

```
//Add a string value to a parameter block
procedure Tfrs_GDS.BuildPBString( var PB: array of char; var PBLen: Integer;
  item: byte; contents: string);
var
  len: Integer;
begin
  {PBLen is the current size of the populated array,
   as well as the position indicator}
  PB[PBLen] := char(item);
  inc(PBLen);
  len:=Length(Contents);
  PB[PBLen] := char(len);
  inc(PBLen);
  StrPCopy(@PB[PBLen],Contents);
  inc(PBLen,len);
end;
```

Then all we have to do is call it like this:

```
BuildDPBString(FDPB,FDPBLen,isc_dpb_user_name,’SYSDBA’);
BuildDPBString(FDPB,FDPBLen,isc_dpb_passsword,’masterkey’);
```

The example contains similar code to manage `Integer` and `Boolean` constants. With these functions we need never worry again about the details of creating parameter blocks.

the rest. I must confess that I have only coded for the ones I have needed, and am thankful for that.

Executing A Statement

We are now ready to actually execute the statement. To do so, we must test for the type of execution required and act accordingly. For this I have created two enumerated types and two functions to help us determine our course of action:

```
function GetDsqlExecType :
  TDsqlExecType;
function GetStatementType(
  StatementHandle:
  pisc_stmt_handle):
  TStatementType;
```

With these we can put together a case statement that tests for each `ExecType` and then, depending on whether the statement type is an `stExecProcedure` or not, make the appropriate execute call.

Why so complicated? Well, actually, it is and it isn’t. InterBase has two execution calls, `isc_dsql_execute` and `isc_dsql_execute2`. The former can be passed an `Input DataArea` or `nil`, and the latter can be passed both an Input and an Output data area. So, this is why we have to test for execution type, ie, do we have parameters? and do we get a result set?

Then, stored procedures need to be treated slightly differently so we need to check the statement type. In the sample code you will see in fact that this does reduce to only four variations, but the case statement makes the code more readable by covering each possible combination. It also allows for easy expansion for other statement types. Although not covered in this article or the example application, it is possible to execute any valid DSQL statement, including `COMMIT` and `ROLLBACK` as well as DDL commands etc.

Reading Results

We are almost there. In fact, we only have one major hurdle left: reading the result set, if there is one. Again, coding can be fairly generic, as we read until we get an Error (not 0) or a No More Rows signal (100) . If there are none to start with then we will get this latter value straight away.

If we are displaying the results on a console, or writing them out to a file, the first step will be to get the column titles. This information is all stored in the OutputDataArea. In the sample code I have supplied a short routine that returns the titles as a concatenated string. It just walks through the data area grabbing each sqlvar[n].Aliasname and padding it to the size of the length of the data type or the length of the alias name (whichever is greater). A further refinement might be to check that the width is sufficient for storing large numbers as strings.

We then need to read each row. Listing 8 outlines the basic process. It looks quite simple, but in practice it isn't. It's the ProcessEachColumn bit that is easy to write in the code snippet and a lot more difficult to write in reality.

We have to ask what we want to do with the results. In the example, I am just writing them to the screen and discarding them. Perhaps they all need to be stored in a result buffer, for later manipulation on the client side.

Let's look at some of these issues. When we call isc_dsql_fetch InterBase populates the OutputDataArea with a row of data. We usually need to get it out of this temporary storage area and place it somewhere more persistent.

To do this we need to decide how and where to store the data. This is very application specific, in the example I just needed to get each row into a string and then write it to the console. Even this presents some challenges. Here is a code snippet, from immediately after a successful fetch:

```
if Fetchcode = 0 then
  with OutputDataArea^ do
    for i:=0 to sqln-1 do begin
      ReadColumn(s,i);
      if result='' then
        Result:=s
      else
       Result:=Result+' '+s;
    end;
```

```
case DataType of
  SQL_Short : begin
                len:=sqlvar[Position].sqllen;
                PSmallInt(sqlvar[Position].SQlData)^:=StrToInt(AParam);
              end;
  SQL_Long  : begin
                len:=sqlvar[Position].sqllen;
                PInteger(sqlvar[Position].SQlData)^:=StrToInt(AParam);
              end;
```

➤ *Above: Listing 7*

➤ *Below: Listing 8*

```
FetchCode=0;
repeat
  FetchCode:=isc_dsql_fetch(@StatusVector, @StmtHandle, 1, OutputDataArea);
  ProcessEachColumn;
until (FetchCode=100) or (FetchCode<>0);
```

```
with OutputDataArea^ do begin
  datatype:= sqlvar[ColNo].sqltype and (not SQL_NULL);
  case DataType of
    SQL_SHORT : if sqlvar[ColNo].sqlscale=0 then
                  AColumn:=IntToStr(PSmallInt(sqlvar[ColNo].SQLData)^)
                else
                  AColumn:=FormatFloat('#,##0.00##', AdjustScale(PSmallInt(
                    sqlvar[ColNo].SQLData)^, sqlvar[ColNo].sqlscale));
    SQL_TEXT  : begin
                  sqllen:=sqlvar[ColNo].Sqllen;
                  s:=pchar(sqlvar[ColNo].SQlData);
                  s:=Copy(S,1,Sqllen);
                  AColumn:=s;
                 end;
  end;
end;
```

➤ *Listing 9*

The result is then returned to the calling procedure, which does a writeln. The really tricky stuff is done in the ReadColumn procedure:

```
Procedure ReadColumn(
  var AColumn:
  String; ColNo: Integer);
```

This procedure is quite similar to the AssignParams procedure that we looked at earlier. In fact it is the inverse. We pass it a var string and a column number and it then checks the data type and converts the contents of the SQLData field to a string. Listing 9 shows a section of the code.

Let's go through what is happening here. We get the data type and test it against each possible type which we are interested in. Each data type requires specific processing. For instance, if the value is an SQL_SHORT it may be stored with 0 scale, in which case we can process it as is, casting the SQLData to a PSmallInt, which we then cast to a String. In another situation we may have to adjust the scale and then do the casting.

A similar process is carried out for SQL_TEXT, except that here we don't need to worry about scale, but we do need to discover the length of the text and copy that out to our local storage.

If you have managed to stay with me this far, you are just about there. The ground we have covered now takes up about 75 pages in the manual! Getting to grips with all this stuff really is a slog, but maybe now you'll never complain again about how complex the Borland Database Engine is to access directly...

Cleaning Up

Of course, having successfully connected to the database, started a transaction and queried some data, it is important to clean up our resources properly.

First of all, don't ever forget to commit your transaction, even if you haven't made any changes. This is important, as InterBase will automatically rollback any uncommitted transactions when the database is disconnected. A side effect of this is that garbage collection is delayed and back record versions are maintained, ultimately slowing the database down. In any case, committing is easy. We really should free our statement handle if we have one and clearing the data areas is not a bad idea, either. If we are packing up for the day then it is also good manners to detach from the database properly. Listing 10 has the code.

This cleanup code is especially important if you wish to reuse any resources within the same application. Stale pointers are always a problem, but never more so when programming at this level.

Putting It All Together

Just how do we use this stuff in the real world? Well, if you are creating user interfaces with lots of tables and lots of edit controls then carry on using your existing VCL components. The performance gains this type of coding will bring just aren't worth the development time. If, however, you are doing non-visual programming, then this level of control may be right for you.

The example application demonstrates a variety of techniques, from attaching to a database, to getting information from it, to querying tables. To simplify the task I have descended a simple class from `TObject` to encapsulate both the API calls and some of the basic functions of database access.

This simple class solves many of our problems. It has standard variables such as database and transaction handles that are declared automatically for us. A handle to gds32.DLL is established and the main API calls are loaded. The ones we don't need are never loaded (thanks to Jason Wharton for this tip). Data structures are initialised automatically. Memory management is simplified. Properties can be used to detect and deal with errors while handles and other objects can be initialised and cleaned up easily.

In short, all the core functionality we need can be made available to us in a single class. This monolithic class is fine for small projects, however, it does suffer from some limitations. It can only work with a single database and a single transaction. All the DSQL programming is outside of it, and should be in a class of its own.

From a purist point of view it would make sense to create several classes, for database connections, transactions and queries. Each should be derived from a base class that knows about InterBase and can test for errors. Unfortunately, this route adds a layer of complexity.

Each database object, say, needs to maintain pointers to the transactions and data sets that are using it, so that an attempt to disconnect the database will gracefully clean these objects up or raise an exception. Once our requirements get to this stage it is

➤ *Listing 10*

```
if assigned(TxnHandle) then begin
  errorcode:=isc_commit_transaction(@StatusVector, @TXnHandle);
  TxnHandle:=Nil;
end;
if assigned(StmtHandle) then begin
  isc_dsql_free_statement(@StatusVector, @StmtHandle, DSQL_Drop);
  StmtHandle:=Nil;
end;
FreeSQLData(OutputDataArea);
if assigned(DBHandle) then
  if assigned(TxnHandle) then
    raise Exception.create(
      'Transaction active. Cannot close database connection.')
else begin
  ErrorCode := isc_detach_database( @StatusVector, @DbHandle);
  DBHandle:=Nil;
end;
```

Error Detection And Handling

Throughout this article the code snippets have not been tested for an error, so as to keep the focus clearly on each call. In practice we can't get away with that. Nearly every call must be tested for failure, a non-zero value indicates an error. Additionally, the first parameter of each call is a pointer to a status vector. This is an array of 20 integers. When a call fails this status vector must be examined. The error constants within can be expanded to meaningful error messages. There are three ways to test for errors and two techniques to examine them.

The traditional C style approach is to test every call with an `if..then..else` statement.

```
if isc_attach_database(....)<>0 then
  //handle error
else
  //continue execution
```

In IBX and FIB the approach has been to call every API function as a parameter to a `CALL` method which then tests the result for an error.

```
call(isc_attach_database(.....), RaiseError);
```

`Call` essentially does something similar to the first example, raising an Exception if the `ISC_STATUS` result is not zero and the `RaiseError` variable is set to `True`. In the example code I have demonstrated using a property to achieve the same result:

```
with frs_gds do
  errorcode:=isc_attach_database();
```

The `errorcode` property has a `SetErrorCode` method which deals with all the problems.

Both the latter two examples greatly simplify the task of error detection, leaving cleaner, more readable code.

When we do get an error we need to find out what it is. For this we have an API call that, given an error code, will look it up in the interbase.msg file and return an error string. We have to repeatedly call `isc_interprete`, passing the `Statusvector` and an output buffer, until all the errors are translated. The `HandleIBErrors` method of the `Tfrs_GDS` class demonstrates how to do this.

You also have the option of using `isc_sql_interprete` to convert errors to SQL error strings where the error arises during DSQL programming. This seems like an unnecessary complication to me, my adherence to ANSI SQL 92 only goes so far and I find the actual InterBase errors more informative.

reasonable to ask whether using the lightweight objects in one of the existing IB API component sets wouldn't be a lot easier (if not a little less portable).

So, the example application keeps things simple and portable across platforms. By using the base class the programming tasks remain fairly simple. Indeed, I was surprised at how simple the coding was and how quickly the work went, once I had my framework in place. I would recommend anyone doing serious InterBase development to have a go at this level and play with the examples. It gets quite easy, once you get the hang of it.

Conclusion

In this article I have tried to summarise the basic steps involved for Delphi developers to work with InterBase at the API level. This is not a substitute for reading the API guide, however, which extensively documents almost every detail of API programming.

There are quite a few details left out. Most, thankfully, are covered in the API guide. No attention has been paid to blobs or arrays, each of which would require articles to themselves.

Take this article in the spirit it is intended: a leg-up on writing to the API in Pascal and the low-down on Pascal specific issues that are not covered in it. Hopefully, the gotchas that I encountered won't now get you.

Paul Reeves is the principal of Fleet River Software, a UK based software house dedicated to InterBase and Delphi development. He can be contacted at paul@fleetriver.demon.co.uk

EXtended SQL Descriptor Areas (XSQLDA)

The heart of Dynamic SQL (DSQL) programming is contained in XSQLDAs and their associated XSQLVARs (eXtended SQL variable). This combo might simply be described as a row of data, although they play a far more complex role than that. An XSQLDA is a contiguous area of memory with the following structure:

```
XSQLDA = record
  version : Smallint           { version of this XSQLDA }
  sqldaid : array [0..7] of Char;    { XSQLDA name field, not used in v4 or v5. }
  sqldabc : isc_long;          { length in bytes of SQLDA, not used inv4 or v5 }
  sqln : Smallint;             { number of fields allocated }
  sqld : Smallint;             { actual number of fields }
  sqlvar : array [0..0] of XSQLVAR;  { first field address }
end;
PXSQLDA = ^XSQLDA;
```

`sqlvar` is a pointer to an array of `XSLQVAR`s. The size of this array is specified by `sqln`.

```
XSQLVAR = record
  sqltype:           Smallint;     { datatype of field }
  sqlscale:          Smallint;     { scale factor }
  sqlsubtype:        Smallint;     { datatype subtype, BLObs & Text types only }
  sqllen:            Smallint;     { length of data area }
  sqldata:           Pointer;      { address of data }
  sqlind:            ^Smallint;    { address of indicator variable }
  sqlname_length:    Smallint;     { length of sqlname field }
  { name of field, name length +space for NULL }
  sqlname:           array [0..31] of Char;
  relname_length:    Smallint;     { length of relation name }
  { field's relation name + space for NULL }
  relname:           array [0..31] of Char;
  ownname_length:    Smallint;     { length of owner name }
  { relation's owner name + space for NULL }
  ownname:           array [0..31] of Char;
  aliasname_length:  Smallint;     { length of alias name }
  { relation's alias name + space for NULL }
  aliasname:         array [0..31] of Char;
end;
PXSQLVAR = ^XSQLVAR;
```

The `XSQLVAR` holds a great deal of information for each column, even a smallint requiring just two bytes of storage will initially require an `XSQLVAR` of some 150 bytes to be passed across the network, in addition to the data itself. However, the information is consistent for all datatypes and enables us to write generic routines that can manipulate it.

`XSQLDA`s and their associated array of `XSQLVAR`s are used both to assign values for parameterized queries and to read data back from a select. The former is typically called an input data area and the latter an output data area, but the structure is the same. The `sqln` property indicates either the number of parameters to be supplied or the number of columns in the result set. An output data area essentially represents a row of data in a result set and each element in the array of `XSQLVAR`s is a column. Given the declaration:

```
var
  i: integer;
  NumCols: integer;
  Value: PChar;
  OutputDataArea: PXSQLDA
```

we would read the number of columns thus:

```
NumCols:=OutputDataArea^.sqln;
```

And to get a pointer to the value stored in each particular column we could do:

```
for i:=0 to NumCols do
  Value:= OutputDataArea^.sqlvar[i].SQlData;
```

In practice things are a lot more complicated. We need to test for null values, check the data type and allocate persistent storage. Depending on the data type we then need to choose how we read the data out.

The section on working with DSQL goes into much greater detail on these issues.