Inside TDataSet ADUG 15.2.99

What is TDataset?

A virtual dataset for accessing a database.
Encapsulates the mechanisms for linking a DB to data-aware controls and sending data To/From the DB
Conceptually data is handled in table form - each row required is buffered internally. Columns are represtented using Fields

From D3 onwards all BDE functions were removed from TDataset making it independant of any DB format.

What TDataset isnt

TDataset has no:

- SQL support
- DB session control functions
- inherent links to any DB
- No index support
- No range setting
- No master-detail linking

Custom Datasets

Need to override virtual methods for minimal functionality dataset

Essentials to create a custom dataset

Fields

- Field definitions
- Binding fields
- Calculated fields

Opening/Closing

Responsible for:

- Field defs
- Record buffer specs
- Record buffer allocation and freeing

Data transfer from dataset to database

Reading/Writing

- Buffer handling
- Field data conversion/verification
- Additional data requirements

Navigation

External calls

- eg First, Next etc

Internal calls

- SetToRecord
- GoToBookmark

Other aspects to consider:

Locate
Lookup
Filtering
Bookmarking
Blobs

ADUG Demonstration design 15.2.99

Significant housework required to implement TDataset core functions
Abstracted out database access from TDataset into another class
Implemented access class using TTable

Aspects of TDataset

1. Buffer Management

TDataSet maintains an array of record buffers for the open dataset.

TDataset controls
- number of buffers
- positioning of the active buffer in that array
- buffer array initialization and destruction

Custom Datasets are responsible for
- Setting core of buffer size
- memory allocation/freeing of buffers
- initialization of buffer contents

All buffer memory is set using TDataSet.UpdateBufferCount. This is called when the dataset is opened or closed, any data sources are added/removed or datalinks altered. Each time a new buffer is required the virtual method AllocRecordBuffer is called to allocate the memory required.

Buffer size

Size of the buffer records is determined by the custom dataset, but both the ancestor and the custom dataset have contributions

TDataset	will give the space required by the calculated fields (CalcFieldsSize)
Custom dataset	needs to work out the space required for its 'own' field data, plus additional space for record number, bookmarking plus any other info req'd

What buffers are there?

function ActiveBuffer: PChar;
The current valid record buffer is given by ActiveBuffer. Although essentially used internally, ActiveBuffer is a public method of TDataSet.

For other buffering, there is a TempBuffer.
Additional buffers can be allocated and maintained within a Custom dataset

function TCustomDS.GetActiveRecBuf(var RecBuf: PChar): Boolean;

When fields are accessing a buffer, then this function dictates the buffer accessed.
This uses the dataset state to set the appropriate buffer to access.

Buffer layout and contents

Buffer contents
- Field data (?) - format not specified
- Bookmark data and flags

Record Number
Some info regarding record location/state
?blob data

Layout is not defined by TDataset - other than the fact that there must be sufficient space as above

2. Dataset States

```
TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert, dsSetKey,
   dsCalcFields, dsFilter, dsNewValue, dsOldValue, dsCurValue);
```

Most states are self-explanatory. There are issues for custom datasets in handling some of the states however. At its simplest level. Some state changes involve setting the new state, setting or reading from fields and then restoring the old state (e.g. setkey locate and filtering can do this)

NB *From Delphi help:*
OldValue *represents the original value of the field (as a Variant).*
NewValue *represents the current value of the field component including pending cached updates.*
CurValue *represents the current value of the field component including changes made by other users of the database.*

3. Fields and TDataset

Fields are intrinsically linked to TDataset. Custom datasets determine what fields exist and handle data transfer to/from the field

Field creation

The custom dataset is responsible for deciding what fields to create, their order and any other initialization required.
TDataset will 'bind' the fields - this involves some verification of the field types, checking for calculated fields (and setting the calculated buffer size required) and checking for blobs

Allocating space for fields

It is not necessary to allocate any specific space for any field type - depending on the internal data handling

Calculated fields are allocated space for the datasize +1 byte - used for setting a null flag.

Blob fields are responsible for their own data handling. This is done using a Stream - via CreateBlobStream. Blob streams are 'external' to TDataset - the CreateBlobStream returns the Stream but gives only the Field and blob stream mode (read/write)

Data conversion to/from fields

You are responsible for getting the raw data from your database and converting it into the appropriate type for you custom dataset.
Most of the field types have well known data types and the conversion is straightforward.

However….
1. Note that date/time fields use internally a TDateTimeRec (defined in DB.pas):

```
type { TDateTimeRec }
  TDateTimeRec = record
    case TFieldType of
      ftDate: (Date: Longint);
      ftTime: (Time: Longint);
      ftDateTime: (DateTime: TDateTime);
  end;
```

It is necessary to convert your data into and out of this format (via a TTimeStamp and then TimeStampToDateTime(TimeStamp) to obtain a valid TDateTime variable).
Best reference here is to see:
function TDateTimeField.GetValue(var Value: TDateTime): Boolean; in DB.pas

2. BCD fields are a special case. The internal access type of these fields is Currency type (an 8 byte scaled integer with 4 decimal places max). It is necessary to override these functions to convert your own data:

```
function BCDToCurr(BCD: Pointer; var Curr: Currency): Boolean; virtual;
function CurrToBCD(const Curr: Currency; BCD: Pointer; Precision, Decimals: Integer):
Boolean;virtual;
```

3. blobs - all blobs are descended from TBlobField and need to implement a stream of some kind to manipulate the blob data. To provide access to this data, the method CreateBlobStream needs to be overridden.

4. Lookup and lookup fields:

Lookup fields

These can operate in 2 ways - either calling lookup in the lookup dataset for each value required, or filling a lookuplist with all the table values required once and then searching in this list for each value as it is needed.

For large lookup datasets using the lookuplist will require a lot of memory. In addition, the lookuplist will not reflect any changes in the lookup dataset since the list was last filled (the list can be made current by calling RefreshlLookupList but be aware that this will cause the entire lookup dataset to be iterated from first to last as the list is filled again). Searches in the lookuplist are made iteratively until a match is found or the end of the list reached. If the lookup fields are indexed and the lookup dataset large then calling lookup for each value may be faster.

If the lookup dataset is not large and not likely to change , the lookuplist is a useful thing to use.

```
procedure TField.CalcLookupValue;
begin
  if FLookupCache then
    Value := LookupList.ValueOfKey(FDataSet.FieldValues[FKeyFields])
  else if (FLookupDataSet <> nil) and FLookupDataSet.Active then
    Value := FLookupDataSet.Lookup(FLookupKeyFields,
      FDataSet.FieldValues[FKeyFields], FLookupResultField);
end;
```

```
procedure TField.RefreshLookupList;
```

Implementing lookup fields

If you want to have lookup fields in your custom dataset, then lookup must be implemented (this is probably not strictly true as if you only ever use cached lookups via lookuplist then lookup is not called).

The BDE code for lookup is shown below:

```
function TBDEDataSet.Lookup(const KeyFields: string; const KeyValues: Variant;
  const ResultFields: string): Variant;
begin
  Result := Null;
  if LocateRecord(KeyFields, KeyValues, [], False) then
  begin
    SetTempState(dsCalcFields);
    try
      CalculateFields(TempBuffer);
      Result := FieldValues[ResultFields];
    finally
      RestoreState(dsBrowse);
    end;
  end;
end;
```

As you can see lookup is really just a wrapper for a call to locate record. Locate record is also the workhorse of the locate method.

Locate record

Locate record is NOT a TDataset method, but you will have to implement some equivalent call to get locate and lookup to work.

Your custom method will need to do the following

Decide what fields are to be searched on
Determine if a case insensitive search is required
Determine if partial matches are allowed.
Find some way to search in your underlying data table for a record that matches all the requirements. If there is a match, return the result fields in some way. In addition, if the originating call was from locate then move the table cursor to that record.

How does the BDE perform a LocateRecord?

The BDE implementation is partly hidden with the BDE itself, however the initial parts are visible.
The search fields are identified and a record buffer set with their values.
A search is made in the current indexes for the best index (or any index) that can search for these fields.
If an appropriate index is found then the search is relatively simple.
If no index is found, then some other search must be used - in the BDE a filter is constructed and used for the search.

5. Filtering

Filtering is defined (but as a virtual operation) in TDataset and the filter language is defined but no parser is supplied. There is a parser for filter text but this IS NOT BDE independent.

To implement filtering in your custom dataset requires the following:

Have a working filter parser.
Have some way to select records based on the filter expression
For records selected by the filter parser the pass to the OnFilter event (if assigned) and determine if it passes or fails. If it fails, then move on the next selected record without presenting that record to the visible parts of your dataset
(The exact implementation will depend on the underlying database, in particular whether or not it supports SQL).

Note that although the filter expression can only change by setting dataset properties (which will be notified to your dataset), the test criteria within the OnFilter event can change without any notification and so the event will need to checked before each record is presented as 'OK' to your dataset.

The FindFirst, FindNext etc functions are dependant on filtering.

As an aside note that RecordCount in TTable with filtering on