DBlib - Disk I/O handling library

Paolo Marcucci
Osservatorio Astronomico di Trieste
Pubblicazione Osservatorio Astronomico di Trieste n. 1440

---

l. Introduction

DBlib is a collection of functions designed in order to simplify the creation and the maintenance of a set of disk data files.

This library is written keeping in mind the typical use of tables made by an application programmer. In fact the definition of the internal structure of tables is subject to frequent modification during the development phase of a complex project. A change in the internal structure means that the files already created are unusable with the new format, so that their data are lost and the data entry work has to be reinitiated in order to recreate the data base of the application.

While this operation is not important when we have a small data base to handle, when the amount of data increases it becomes an awkward and dangerous job to reinput all the data. Several times, this kind of work drives the programmer to a strange behaviour: either the internal format of data is full of redundancy or some less important fields in the structure are overwritten by other much mor important informations.

In both cases the internal structure of the data does not in reality reflect the data itself as defined in the specification document, and some times "programming acrobatics" are required to extract meaningful information from raw data.

DBlib solves these problems using a different approach. Instead of carefully designing the data base (which is the best thing to do), it is possible to face the problem with a "try and see" approach. This means that the programmer can define a data structure and fill a table of real data. Then, when some fields of the structure become no longer useful or others need to be added, the structure can be modified and the table, via a general conversion program (see 7.), can transfer its data to a file with the new defined structure. At this point, all the programmer has to do is to fill the table with the new information required.

2. Data base structure definition

The definition of the data base structure is left to the programmer keeping in mind the limits, defined in dblib.h, of MAXFILES tables and a maximum of MAXFIELDS fields for a single record. Within these limits the programmer is free to declare fields of five basic data types: INT, BOOLEAN, CHAR, LONG and FLOAT. Monodimensional arrays are implemented for INT, LONG, BOOLEAN, CHAR and FLOAT elements. Bidimensional arrays are implemented for INT, LONG and FLOAT elements.

2.1 Data types

As seen in the previous paragraph, 13 data types are defined. All other types (struct, union, etc..) are not currently implemented. Let's see a description of these data types:

- INT: is the classical int type defined in C. Its size depends on the machine implementation varying from two to four bytes.
- CHAR: the char type requires an additional value given in the form char [n] where n is the number of bytes reserved for this field.
- LONG: a four-byte int.
- BOOLEAN: a boolean variable that can assume only two values: True or False. It is implemented as a long integer for data portability issues.
- FLOAT: a floating point number. Internal implementation can vary depending on the machine and the operating system used.
- INTARR: an array of n INT (int [n]).
- LONGARR: an array of n LONG (1ong [n]).
- FLOATARR: an array of n FLOAT (f1oat [n]).
- BOOLARR: an array of n BOOLEAN (1ong [n]).
- CHARARR: an array of n CHAR (char [n][m])
- INTDARR: a bidimensional array of n*m INT (int [n][m]).
- LONGDARR: a bidimensional array of n*m LONG (long [n][m]).
- FLOATDARR: a bidimensional array of n*m FLOAT (float [n][m]).

2.2 The configuration file

The structure of the data base is stored in an ASCII file called the configuration file. In this file the programmer defines the structure of all the tables used by the application in the form:

```
table_1_name: {
  field_1_name: type;default;description;
  field_2_name: type;default;description;
  :
  field_n_name: type;default;description;
  }

table_2_name: {
  field_1_name: type;default;description;
  :
  field_m_name: type;default;description;
  }
```

where:

- **table_x_name** is the name of the file. However, the programmer can use a lot of files with the same table x name, so let's consider it a "class" or type of file.
- **field_x_name** is the name of each field. Different file classes can have fields with the same name without causing any interference.
- **type** is the type of field as seen in 2.1.
- **default** is the default value that will be loaded in the field at the initialization of the file.
- **description** is a brief description used for automatic documentation or editor mask self-generation.

Example:

```
# nodes- the ethernet node definition class
       nodes : {
               id: int;0;Identifier;
               name: char[41];NULL;Name of the node;
```

```
arpa: char[16];123.0.0.0;ARPA number;
type: int;1;Node type;
}
```

Note that the case is not relevant.

Comments may be added to the file by putting a '#' in the first column of the line. Blank lines are ignored.

By default the name of the configuration file is "dbms.cfg" but this can easily be changed declaring an external variable in the application program. The name of this variable (an array of 81 chars) is fdb_config.

3. Manipulation of the data base

The data base so defined has to be created and filled with information. The creation is accomplished by a single function that reads the configuration file and creates a data file with the correct structure, while the management of the table is performed by a more complex set of functions. Let's see a list of all the functions available:

int InitDB()

Initialize the data base internal structure. It has to be the first call to the DBlib library in the user program

int CreateDBFile (fn,ft,fh)

```
char fn[],ft[];
int *fh;
```

Creates a file with the name fn using the internal definiton of the ft class and returns a handle fh to this file. For example, let's see how to create a file named 'xx.xx' using the class 'nodes' seen before:

```
status = CreateDBFile ("xx.xx", "nodes", &id);
```

The function returns a status that can assume the value FDB_OK if the file has been created or an error if problems arise.

int OpenDBFile (fn,fh)

```
char fn[];
int *fh;
```

Opens an already existent file and returns a handle. There is no need to specify the class of the file, only its name is required.

int CloseDBFile (fh)

```
int fh;
```

Closes the file with the handle fh and releases the memory associated to the intermediate buffer.

int GetDBRecByNum (fh,nr)

```
int fh,nr;
```

Reads the record number nr from the file with the fh handle. For example, if we want to read the first record of the file 'xx.xx', this is the sequence of operations to execute:

```
status = OpenDBFile ("xx.xx", "nodes", &id);
status = GetDBRecByNum (id, 1);
```

in the example, no error checking was done but a real program must provide it. The record read is stored in the intermediate buffer related to the file and the fields can be retrieved using the GetDBFieldByName function.

int PutDBRecByNum (fh,nr)

```
int fh,nr;
```

Writes the record number nr on the file with the fh handle. The information written on the file are contained in the intermediate buffer that is filled using the PutDBFieldByName function.

int AddDBRec (fh)

```
int fh;
```

Adds a record to the file with the fh handle.

int DelDBRecByNum (fh,nr)

```
int fh,nr;
```

Deletes the record nr from the file, performing a shuffling of all the subsequent records in order to compress the file thereby reducing the disk space required.

int PutDBFieldByName (fh,fn,v,si,sj)

```
int fh;
char fn[];
char *v;
int si,sj;
```

Stores the data pointed to by v in the intermediate buffer of the file with the fh handle at the position defined by the field name fn. For example let's ask the user the name of the node and store it in the intermediate buffer:

```
char s[81];
status = OpenDBFile ("xx.xx", "nodes", &id);
printf ("Node name: ");
scanf ("%s",s) ;
status = PutDBFieldByName (id, "NAME", s);
```

Note again the lack of error checking...

Although this function accepts a (char*) pointer for the v parameter, in our example it was not necessary to specify it due to the fact that s was already a (char*). If we want to store a different type of data we will use the following template:

```
int i;
status = OpenDBFile ("xx.xx", "nodes", &id) ;
printf ("Node number: ");
scanf ("%d",&i);
status = PutDBFieldByName (id, "ID", (char*) &i);
```

In this way, we can pass every kind of data we want to the function. Array variables behave in a different way: the last two parameters, si and sj, are used to address the item we want to store in the intermediate buffer If the array is bidimensional,

si is the first index and sj is the second one; if the array is monodimensional only si is used. If the application program wants to store the entire array as a whole, these parameters can be set to NOP (a constant defined in dblib. h). Let's see three examples:

Example 1 - store an array as a whole:

```
int arr [5] ;
status = PutDBFieldByName (id,"ARR",(char*)&arr,NOP,NOP);
```

Example 2 - store the third element of a monodimensional array:

```
int arr [5] ;
status = PutDBFieldByName (id,"ARR",(char*)&arr,2,NOP);
```

Example 3 - store the element [2,4] of a bidimensional array:

```
int arr [ 5, 7 ] ;
status = PutDBFieldByName (id,"ARR",(char*)&arr,1,3);
```

int GetDBFieIdByName (fh,fn,v,si,sj)

```
int fh ;
char fn[];
char *v;
int si,sj;
```

Retrieves a data from the intermediate buffer of the file with the handle fh at the position indicated by the field name fn.

The syntax of this function is quite similar to PutDBFieldByName, but let's see an example:

```
int i;
char s[81];
status = OpenDBFile ("xx.xx", "nodes", &id);
status = GetDBRecByNum (id, 1);
status = GetDBFieldByName (id, "NAME", s);
status = GetDBFieldByName (id, "ID", (char*) &i);
printf ("Id=%d, Name=%s \n", i, s );
```

Array variables behave in a different way: the last two parameters, si and sj, are used to address the item we want to retrieve from the intermediate buffer. If the array is bidimensional, si is the first index and sj is the second one; if the array is monodimensional only si is used. If the application program wants to retrieve the entire array as a whole, these parameter can be set to NOP (a constant defined in dblib. h).

Let's see three examples:

Example 1 - retrieve an array as a whole:

```
int arr[5];
status = GetDBFieldByName (id,"ARR",(char*)&arr,NOP,NOP);
```

Example 2 - retrieve the third element of a monodimensional array:

```
int arr[5];
status = GetDBFieldByName (id,"ARR",(char*)&arr,2,NOP);
```

Example 3 - retrieve the element [2,4] of a bidimensional array

```
int arr [5,7] ;
status = GetDBFieldByName (id,"ARR",(char*)&arr,1,3);
```

4. Requirements

An application program that wants to make use of the disk i/o handling facilities provided by DBlib has to add only one line of code at the beginning of the source:

```
#include
```

If the program wants to use the internal fields of the data base structure, another line of code has to be added:

```
extern fdb_filedef file[MAXFILES);
```

This line is absolutely superflous if the program knows the structure of the tables that are used, but if it wants to know the number of fields or the name of these, it can have access to this information through this structure.

5. Data movements

The program variables cannot be directly written on a DBlib file. Instead, an additional function call is required to store variables in the intermediate buffer

```
Program   <---> Intermediate <---> Disk
variables <--->   buffer     <---> file
```

This may seem criptic and inefficient, but provides a layer of data abstraction by which the programmer doesn't really have to know how the data file structure is made up, but only the name of the fields that it contains. So, if the need arises to position the feld "NAME" after the field "TYPE", no changes to the

application program will be required.

6. Errors

All DBlib functions are of type int, so a status value can be returned and tested. A set of errors is defined in dblib.h :

FDB_OK
: The function performed its work properly.

FDB_WRONGREC
: A wrong record number was passed to GetDBRecByNum or PutDBRecByNum. It means that the record number is less than zero or greater than the maximum number of records in the file. If the program wants to add a record to a file, thus giving a record number that is actually greater than the maximum number of records in the file, the AddDBRec function must be used.

FDB_FILENOTOPEN
: This error is returned by all functions that perform write or read operations on a file that was not correctly opened or was already closed.

FDB_WRONGFIELD
: A wrong file name was passed to the GetDBFieldByName or PutDBFieldByName function5.

FDB_OPENERR
: A problem was reported by the OpenDBFile or CreateDBFile functions during the physical connection to the disk file.

7. Conversions

At stated before (1.), the structure of a DBlib file can easily be changed and the data contents of the file can be preserved. To achieve this, a conversion program is required. This conversion program, dbconv, reads an input file and stores the useful informations in a new file with the modified structure, leaving new fields empty and ready to be filled by users. The syntax of the program is:

```
dbconv [ inputfile output file ]
```

if the file names are not given the program will ask for them and then will start the conversion process.