

## File Structure

Data is stored in a disk file as NULL terminated ASCII strings. The record structure of a file is defined by the default value of each field in the **Default Value Array**. In *dbopen()*, the **Variable List-Array** and **Decoded Record Buffer** are created from the **Default Value Array**, filled with the 1st record in the file and stored in the **Variable List-Header**.

The order in which fields are stored in the file, **Variable List-Array**, **Decoded Record Buffer** and **Decoded Record Structure** corresponds to the order in which they appear in the **Default Value Array**. Addresses to the **Variable List-Array** and **Decoded Record Buffer** are present in the **Variable List-Header**.

Functions that move the record pointer-*go()*, *skip()*, *loca()*, etc or write a record/field at the current pointer-*wrrec()*/*repfld()* cause the data in the **Variable List-Array** and **Decoded Record Buffer** to be updated with the current record.

Fields in the **Decoded Record Buffer** can be mapped to names in a **Decoded Record Structure** (See sample design in DlgFunc.doc) for named access of field names in programs.

### Variable List-Header

The details of a variable list are placed in a DBFL structure. The list can be the fields in a record of a database file or a list of memory variables. When dialog boxes are used each array element 'i' in the Variable List-Array corresponds to a GENERAL EDIT box of ID='i'.

```
struct DBFL{ //File & Memory Variable List-Header
    FILE *frd; //File pointer returned by fopen(). Must be NULL for memory variable lists.
    FLDS *fld; //File & Memory Variable List-Array(dynamic 1D array)
    DWORD nfld; //Number of elements in Variables list
    //-----Unused for Memory Variable Lists-----//
    DWORD rlen; //File variable(field) list(record)-byte length(incl NULL termination of field)
    DWORD rcnt; //File variable(field) list(record)-count in file
    void **buf; //File variable(field) list(record)-Decoded Record Buffer(dynamic 2D array)
};
struct FLDS{ //File & Mem Variable List Element Details
    unsigned err; //Variable-GENERAL EDIT box:Validation results
    char *lim; //Variable-GENERAL EDIT box:limits for default validation,COMBO box:List
    char *dfv; //Variable-Default Value string
    char *txt; //Variable-Actual value string
    DWORD arry; //Variable-Last (0)/not last(1) element in List
    DWORD vtyp; //Variable-Data type of value:float(1),integer(2),char(0)
    DWORD vlen; //Variable-Char length of value
    DWORD vdec; //Variable-Decimal places in value
};
```

dfv pointer to **Default Value string**. Provides the Width and type(float, integer or character) of the variable in a list. Displayed in a GENERAL/COMBO EDIT box when the 'Default' button is pressed. Also used for the default width of the box & length validation of the box entry.

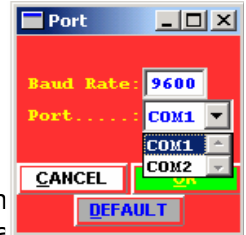
In the examples shown below spaces are part of the variable width, sign if present in numeric fields must abutt the number. "/" represents the '/' character.

Integer	- " 99 " , " +99 " , " -99 "	- variable width:4,5,5 digits.
Floating point	- " 9.9 " , " +9.9 " , " -9.9 " , " .005 " , " 500. "	- variable width:5,6,6,6,6 digits
Multi character	- " Ab.d " , " 1//2 " , " .. "	- variable width: 6,3,3 chars.
Single character-	- " " , " a " , " A " , " . " , " / "	- any non digit character

- err GENERAL/COMBO EDIT box content validation results. Assigned & used in dlgProc(), the Dialog box Window Procedure.
- txt Pointer to **Actual value string**-A field of a file record or memory variable. The length and type of the string is the same as 'dfv'. 'txt' is displayed in a GENERAL EDIT box as the initial value and user i/p to the box is written to it.
- lim Pointer to constant string.

For COMBO EDIT boxes

Each string is a compound string composed of a series of ',' delimited strings. Each ',' delimited string is placed in the LIST box appearing under the EDIT box of a COMBO EDIT box. eg lim[0]="COM1,COM2" or lim[0]="COM1, " "COM2" appears as shown.



For GENERAL EDIT boxes

Compound limit string for default validation of box contents in vfield(). If a pointer is 0(NULL) or any of the "No validation" options below, default validation performed. Structure of each string depends on the type of variable being validated.

For Numeric Variables

'lim' element Format

Spaces may be present except between an abutting sign and number.

- "Hi limit,Lo Limit" - Lo and Hi limit specified
- "Hi limit" - Lo limit=0, Hi limit specified
- "Hi limit, " - Lo limit=0, Hi limit specified
- ",Lo limit" - Hi limit=∞, Lo limit specified
- "", "", " - No validation

Validity of box contents

Lo Limit=<edit box content<=Hi Limit (Hi Limit>=Lo Limit)

Lo Limit=<edit box content (Hi Limit< Lo Limit)

Hi/Lo Limit Structure

Limits in the 'lim' string consist of Numeric literals:-

Numeric literals

Limit Format: +/-<float/integer number>. Sign, if present, must abut the number

eg Hi/Lo Limit = -123 , +456

Hi/Lo Limit = -1.23 , +45.6

For Multi character Variables

'lim' element Format

'F/E' indicates:file name/non file name. Leading/trailing spaces are removed.

- "E" -GENERAL EDIT box content is not empty
- "F" -GENERAL EDIT box content is a valid file name
- "" -No validation

Valid File Name

File names are a path & name w/o '.' and suffix, last character is not '\'.

For Single character Variables

'lim' element Format

Leading and trailing space characters are removed. If spaces must be present, they must have non space characters on both sides.

- "<List of valid characters> " - List of valid characters
- "" - No validation

Validity of box contents:

For a box to be valid, the edit box content must be a non numeric entry present in the 'lim' string. eg: lim=" abc ABC "



## Decoded Record Structure

A C++ 'Structure' used for named access of fields of the record in the **Decoded Record Buffer** array. Members are array(field) names of of **Field Arrays** of type:-

Pointer to DWORD.....: DWORD\*, for integer fields

pointer to double.....: double\* , for float fields

pointer to char pointer: char \*\* , for string fields

Thus Each member, is an array of DWORD or float data or pointers to strings. If multiple fields are enclosed in '[,]' marks in the **Default Value Array**, the array has multiple elements. Members must have the same order as the strings in the **Default Value Array** and **Decoded Record Buffer**:-

```
struct Structure_Name{
    char    **Array(0)_Name;        //Array(0)-Required for Delete Field
    type    * Array(1)_Name;        //Array(1)-type can be 'DWORD','double','char*'
    type    * Array(2)_Name;        //Array(2)-type can be 'DWORD','double','char*'
    type    * Array(n)_Name;        //Array(3)-type can be 'DWORD','double','char*'
}
```

In a variable of the **Decoded Record Structure**, each member occupies 4 bytes(bytes occupied by a pointer in C++) and are placed one after another in memory in the order of appearance in the structure declaration, thus pointer to **Field Array**, *Array(0)\_Name* appears first in memory & *Array(n)\_Name* last.

For the above example a structure FLDN would be defined as shown:-

```
struct FLDN{
    char        **Del;
    DWORD       *Integers;
    double      *floats;
    char        **strings;
};
```

## Named Access of Record Fields

Each element 'i' of the **Field Array Address**(dbf->buf[i]) are pointers to field arrays stored in the same order as members in the **Decoded Record Structure** variable. However the arrays to which they point are inaccessible since the pointer is declared as 'void\*'(pointer to an array of 'void' data) & dbf->buf as 'void\*\*' (pointer to an array of 'void\*').

If allocate memory to a variable of type FLDN and map(type cast) the contents of 'dbf->buf[][]' to it:-  
FLDN \*pst=(FLDN\*)dbf->buf;

The fields of the record in 'dbf->buf[][]' can be accessed by name as follows:-  
pst->integers[0,1,2,3],pst->floats[0,1,2],pst->strings[0,1].

## Database Functions

<b>USER Functions</b>	
<a href="#">dbopen</a>	Open database file.
<a href="#">dbclose</a>	Close database file.
<a href="#">repfld</a>	replace field with specified value, in current record
<a href="#">pack</a>	pack file
<a href="#">isdel</a>	Current record is deleted-yes(1)/no(0)
<a href="#">Loca</a>	Locate 1st key matched record
<a href="#">Cont</a>	locate next key matched record
<a href="#">LocaDel</a>	locate 1st deleted record
<a href="#">ContDel</a>	locate next deleted record
<a href="#">LocaNDel</a>	locate 1st undeleted record
<a href="#">ContNDel</a>	locate next undeleted record
<a href="#">recn</a>	current record number in file
<a href="#">skip</a>	skip 1 or multiple records
<a href="#">append</a>	Append Blank record
<a href="#">recc</a>	File undeleted record count
<a href="#">dele</a>	Delete current record
<a href="#">reca</a>	Recall current record
<a href="#">zap</a>	Delete all records and Pack file
<a href="#">gobot</a>	Go to last record in file
<a href="#">gotop</a>	Go to First record in file
<a href="#">go</a>	Go to specified record
<a href="#">dbeof</a>	Record pointer is at end of file?
<b>INTERNAL Functions</b>	
<a href="#">openf</a>	Open/create binary file for r/w
<a href="#">fsize</a>	file size in bytes
<a href="#">rent</a>	file total record count calc
<a href="#">ndel</a>	file deleted record count
<a href="#">recl</a>	file record length calculation
<a href="#">rdrec</a>	Read current record into <b>Variable List-Array</b>
<a href="#">wrrec</a>	Write current record from <b>Variable List-Array</b>
<a href="#">blankbuf</a>	Create new record buffer filled w/ spaces
<a href="#">DelBuf</a>	Delete <b>Decoded Record Buffer</b>
<a href="#">MakBuf</a>	Create empty <b>Decoded Record Buffer</b>
<a href="#">CopBuf</a>	Fill <b>Decoded Record Buffer</b> form <b>Variable List-Array</b>
<a href="#">FieldID</a>	Get Field index in <b>Variable List-Array</b>
<a href="#">ArrayWid</a>	Get number of elements in specified <b>Field Array</b>

### Common Function Inputs

- dbf*    Pointer to **Variable List-Header** variable of type DBFL, declared in the calling program and filled by dbopen().
- kfn*    Field Array Pointer in Decoded Record Buffer of Key Field. If 0, delete field is selected
- kfi*    Index in Field Array of Key Field.

## USER Functions

### dbopen

**DBFL dbopen(DBFL \*dbf, CHAR \*\*dfv, char \*fnam);**

#### Parameters

*fnam* File name & path string pointer. If not found, a new file is created at the name & path specified  
*dfv* **Default Value Array**. Must be initialized in the function calling dbopen() & remain unchanged while file is open.

#### Descripton

Open a database file. A file containing unpacked deleted records will result in erroneous operation.

#### Returns

Variable of type DBFL with the first file record read into it. If a new file was created by the function or the opened file was empty, all fields contain NULL terminated spaces.

#### Processing

- 1>Each substring 'i', corresponding to field 'i' in the current file record and box ID 'i' in CreateWindowEx(), is extracted from *dfv*.
  - 1a>For each string, an element is appended to the **Variable List-Array** 'dbf->fld'(dynamic array).
  - 1b>the '[' and ']' array delimiters are removed from the extracted string by FrmStr() and its address is placed in 'dbf->fld[i].dfv'(dynamic array).
  - 1c>Each string within the array delimiters represents a record field that is part of a multi element array. This is indicated by '1' in 'dbf->fld[i].array' for all except the last substring before ']'.  - 1d>An empty dynamic string w/ length of the extracted string is created and its address placed in 'dbf->fld[i].txt'.
- 2>The number of substrings(fields) extracted is placed in 'dbf->nfld'.
- 3>An empty **Decoded Record Buffer** is created by MakBuf() and its address assigned to 'dbf->buf'.
- 4>The total length of all the default value strings(incl NULL) is assigned to 'dbf->rln'(record length).
- 5>In openf(), file *fnam* is searched & opened, if not found, a new file is created. 'dbf->frd' assigned the file handle.
- 6>If the file is not empty, the first record is read into the 'dbf->fld[i].txt' strings created.
- 7>The number of undeleted records is counted and assigned to 'dbf->rcnt'.

### dbclose

**void dbclose(DBFL \*dbf);**

#### Descripton

Close database file

#### Processing

Pack the file for error free execution of subsequent calls to dbopen() and delete the dynamic arrays created in dbopen().

### isdel

**DWORD isdel(DBFL \*dbf);**

#### Descripton

returns 1-Current record is deleted, 0-Current record is not deleted.

# repfld

```
void repfld(DBFL *dbf, void *kfn, DWORD kfi, char *val, DWORD p);  
void repfld(DBFL *dbf, void *kfn, DWORD kfi, double val);  
void repfld(DBFL *dbf, void *kfn, DWORD kfi, DWORD val);
```

## Parameters

*val* Data(string,float,unsigned integer) to replace field with

*p* 1-If *val* is a dynamic string(created with the 'new' operator) and is to be deleted.

## Descriptor

Replace field at **Field Array** element *kfn*[*kfi*] with value *val* in the current record.

# pack

```
void pack(DBFL *dbf);
```

## Descriptor

Remove deleted records from the file(Pack the file). Deleted records are not physically removed instead undeleted records are moved to the top of the file and deleted records to the bottom. When records are appended, deleted records are overwritten. The contents of the deleted records is junk and must not be used.

# Loca

```
DWORD Loca(DBFL *dbf, void *kfn, DWORD kfi, char *key, DWORD exact, DWORD top);
```

## Parameters

*key* String to match Key Field with

*top* 1-Start from top of file/0-Start from next record.

*exact* If the following conditions on string lengths are violated return FALSE(0)

*exact=0*:w/o trailing spaces, LH len=<RH len(extra RH len ignored)

*exact=1*:w/o trailing spaces, LH len==RH len

*exact=2*:w/ trailing spaces, LH len==RH len

*exact=3*:w/ trailing spaces, LH len=<RH len(extra RH len ignored)

LH string: Key Field

RH string: *key*

## Descriptor

If the conditions imposed on string lengths by the *exact* i/p are violated, return FALSE(0). Ignoring the extra (trailing)length on the RH string over the LH string, compare them for each record in the file, from the 1<sup>st</sup> record(top=1) or from the next record(top=0). If a record is found in which the compare yields a case insensitive match, position the record pointer at this record and return 1. Otherwise position the record pointer at the end of file and return 0.

## Returns

1-Record found. Record pointer is at the 1<sup>st</sup> record at which result of compare is TRUE(1)

0-Record not found. Record pointer is at last+1 record. dbeof() returns TRUE.

# Cont

**DWORD Cont(DBFL \*dbf, void \*kfn, DWORD kfi, char \*key, DWORD exact);**

## Parameters

See **Loca()**. All inputs must be same as used in the matching call to **Loca()**

## Description

**Cont()** is a database function that searches from the current record position for the next record meeting the most recent **Loca()** condition executed. It terminates when a match is found or end-of-file is encountered. If **Cont()** is successful, the matching record becomes the current record.

## Returns

1-Record found. Record pointer is at the 1<sup>st</sup> record at which result of compare is TRUE(1)

0-Record not found. Record pointer is at last+1 record. **dbeof()** returns TRUE.

## Examples

- Normal Use:-

```
Loca()
while(!dbeof()){
    Cont()
}
```

- Equivalent Use:-

```
Loca()
while(!dbeof()){
    skip()
    Loca()
}
```

# LocaDel

**DWORD LocaDel(DBFL \*dbf);**

## Description

Search each record in the file, from the 1<sup>st</sup> record onwards for a deleted record. If a record is found, position the record pointer at this record and return 1. Otherwise position the record pointer at the end of file and return 0.

## Returns

1-Deleted record found. Record pointer is at the 1<sup>st</sup> deleted record

0-Deleted record not found. Record pointer is at last+1 record. **dbeof()** returns TRUE.

# ContDel

**DWORD ContDel(DBFL \*dbf);**

## Description

**ContDel()** is a database function that searches from the current record position for the next deleted record. It terminates when a record is found or end-of-file is encountered. If **ContDel()** is successful, the found record becomes the current record.

## Returns

1-Deleted record found. Record pointer is at the 1<sup>st</sup> deleted record

0-Deleted record not found. Record pointer is at last+1 record. **dbeof()** returns TRUE.

### **Examples**

- Normal Use:-

```
LocaDel()  
while(!dbeof()){  
    .  
    ContDel()  
}
```

- Equivalent Use:-

```
LocaDel()  
while(!dbeof()){  
    .  
    skip()  
    LocaDel()  
}
```

## **LocaNDeI**

**DWORD LocaNDeI(DBFL \*dbf);**

### **Description**

Search each record in the file, from the 1<sup>st</sup> record onwards for a non deleted record. If a record is found, position the record pointer at this record and return 1. Otherwise position the record pointer at the end of file and return 0.

### **Returns**

1-Non Deleted record found. Record pointer is at the 1<sup>st</sup> non deleted record  
0-Non Deleted record not found. Record pointer is at last+1 record. dbeof() returns TRUE.

## **ContNDeI**

**DWORD ContNDeI(DBFL \*dbf);**

### **Description**

**ContNDeI()** is a database function that searches from the current record position for the next non deleted record. It terminates when a record is found or end-of-file is encountered. If **ContNDeI()** is successful, the found record becomes the current record.

### **Returns**

1-Non Deleted record found. Record pointer is at the 1<sup>st</sup> non deleted record  
0-Non Deleted record not found. Record pointer is at last+1 record. dbeof() returns TRUE.

### **Examples**

- Normal Use:-

```
LocaNDeI()  
while(!dbeof()){  
    .  
    ContNDeI()  
}
```

- Equivalent Use:-

```
LocaNDeI()  
while(!dbeof()){  
    .  
    skip()  
    LocaNDeI()  
}
```

## recn

DWORD recn(DBFL \*dbf);

### Returns

Returns the record number of the current record as an integer numeric value. If the file contains zero records. **recn()** returns one, **dbeof()** returns TRUE and **recc()** returns zero.

## skip

DWORD skip(DBFL \*dbf, int gap);

### Parameters

*gap* A numeric value specifying the number of records to move the record pointer from the current position. A positive value moves the record pointer forward and a negative value moves the record pointer backward.

### Descripton

Moves the record pointer to a new position relative to the current position in the file. SKIPPING forward beyond the last record positions the record pointer at **recc()+1** and **dbeof()** and **skip()** return TRUE. SKIPPING backward beyond the first record or beyond the end-of-file causes **skip()** to return zero and the record pointer to not change.

### Returns

1-If the skip was performed  
0-If the skip was not performed(no change in record pointer).

## skip

DWORD skip(DBFL \*dbf);

### Descripton

Moves the record pointer one record forward from the current position in the file. SKIPPING beyond the last record positions the record pointer at **recc()+1** and **dbeof()** and **skip()** return TRUE. SKIPPING beyond the end-of-file causes **skip()** to return zero and the record pointer to not change.

### Returns

1-If the skip was performed  
0-If the skip was not performed(no change in record pointer).

## append

void append(DBFL \*dbf);

### Descripton

Adds a new record to the end of the current database file and then makes it the current record. The new field values are initialized to empty values(spaces) for each data type

## recc

DWORD recc(DBFL \*dbf);

### Returns

Returns the number of physical records in the current database file as an integer numeric value. If the file contains zero records. **recc()** returns zero.

# dele

```
void dele(DBFL *dbf);
```

## Description

The first field in every record is the Delete field. A '\*' here indicates that the record is deleted and ' ' that it is not deleted.

**dele()** is a database command that tags records so that they can be queried with **isdel()** or logically removed from the database with **pack()**. A Deleted records is tagged by placing a '\*' in its Delete field. Once a record is deleted, it can be Recalled by **reca()**.

# reca

```
void reca(DBFL *dbf);
```

## Description

The first field in every record is the Delete field. A '\*' here indicates that the record is deleted and ' ' that it is not deleted.

**reca()** is a database command that restores the current record in the current file, if it was marked for deletion by **dele()**. Note that once a file has been Packed by **pack()**, deleted records are logically removed from the file and can't be recovered.

# zap

```
void zap(DBFL *dbf);
```

## Description

Delete all records in a file and pack the file.

# gobot

```
DWORD gobot(DBFL *dbf);
```

## Description

**gobot()** is a database command that positions the record pointer in the current file to the bottom(last record) of the file. record pointer is set to record:**recc()+1** ie **recn()**. If the file is empty, **dbeof()** returns TRUE(1)

## Returns

If the file is empty, returns FALSE(0) else TRUE(1)

# gotop

```
DWORD gotop(DBFL *dbf);
```

## Description

**gotop()** is a database command that positions the record pointer in the current file to the top(1st record) of the file. Record pointer is set to record:**recc()+1** ie **recn()**. If file is empty, **dbeof()** returns TRUE.

## Returns

If the file is empty, returns FALSE(0) else TRUE(1)

# go

DWORD goto(**DBFL** \**dbf*, **DWORD** *rec*);

## Parameters

*rec* Target record

## Description

**go()** is a database command that positions the record pointer in the current file to a record in the file specified by a number *rec*. Record pointer is set to record:**recc()**+1 ie **recn()**. If target record is out of range of the database file, **dbeof()** returns TRUE

## Returns

If the target record is out of range of the database file, returns FALSE(0) else TRUE(1)

# dbeof

DWORD **dbeof**(**DBFL** \**dbf*);

## Returns

If the database file is empty, no file is opened or an attempt is made to move the record pointer beyond the last record in a database file, **dbeof()** returns TRUE(1), otherwise it returns FALSE(0).

## Description

A database function to test for an end-of-file boundary condition when the record pointer is moving forward through a database file. Any function that can move the record pointer can set **dbeof()**.

The most typical application is a part of the *<condition>* argument of a **while(<condition>){}** construct that sequentially processes records in a database file. Here *<condition>* would include a test for **!dbeof()**, forcing the loop to terminate when **dbeof()** returns TRUE.

**dbeof()** is often used to test whether a **Loca()**, **LocaDel()**, **LocaNDel()**, **Cont()**, **ContDel()**, **ContNDel()** failed

When **dbeof()** returns TRUE, the record pointer is at **recc()**+1 and further attempts to move the record pointer forward return the same result without error. Once **dbeof()** is set to TRUE, it retains its value until there is another attempt to move the record pointer.

## **INTERNAL Functions**

# openf

FILE\* **openf**(char \**name*);

## Parameters

*name* Database file name and path

## Description

Uses Dos style fopen(). If file doesn't exist, creates a new file.

## Returns

File handle

# rcnt

DWORD **rcnt**(**DBFL** \**dbf*);

## Returns

Number of records(deleted+undeleted) in file.

## fsize

DWORD fsize(FILE \*hand);

### Parameters

*hand* Database File handle-returned by openf()

### Returns

File size(deleted+undeleted recs, including EOF mark) in bytes

## ndel

DWORD ndel(DBFL \*dbf);

### Returns

Number of deleted records in file.

## recl

DWORD recl(DBFL \*dbf);

### Returns

Byte length of record in file. Includes NULL termination of all fields

## rdrec

DWORD rdrec(DBFL \*dbf);

### Description

Read current record into **Variable List-Array & Decoded Record Buffer**.

### Returns

If file is empty, the **Variable List-Array & Decoded Record Buffer** are filled with spaces & return value is 0 otherwise they are filled with field values of the current record and return value is 1.

## wrrec

DWORD wrrec(DBFL \*dbf);

### Description

Write **Variable List-Array** to file at current record.

### Returns

1-Write was successful,0-Write was unsuccessful.

## blankbuf

char\*\* blankbuf(DBFL \*dbf);

### Description

Create a record buffer filled w/ spaces. The record buffer is a dynamic array of dynamic strings. The number of strings is equal to the number of fields and the length of each string is the length of the corresponding field.

### Returns

Pointer to buffer

## DelBuf

```
void DelBuf(DBFL *dbf);
```

### Description

Delete Decoded Record Buffer.

## MakBuf

```
void MakBuf(DBFL *dbf);
```

### Description

Create empty Decoded Record Buffer. String elements are filled with spaces & numerics with zero.

## CopBuf

```
void CopBuf(DBFL *dbf);
```

### Description

Fill Decoded Record Buffer form Variable List-Array.

## FieldID

```
DWORD FieldID(DBFL *dbf,void *start,DWORD ofst);
```

### Parameters

*start* Start address of Field Address Array of the Field whose index in the Variable List-Array is required.

*ofst* Index in Field Address Array of the Field whose index in the Variable List-Array is required.

### Description

Determines the index in the Variable List-Array of a field

### Returns

Returns the index in the Variable List-Array of a field

## ArrayWid

```
DWORD FieldID(DBFL *dbf,void *start);
```

### Parameters

*start* Start address of Field Address Array whose number of elements is required.

### Description

Determine the number of elements in a Field Array

### Returns

Returns the number of elements in a Field Array