INDEX

Overviews

Specs

BULLET functions

System-level Functions:

InitXB
ExitXB
AtExitXB
MemoryXB
BreakXB
BackupFileXB
StatHandleXB
GetExtErrorXB

Debug Functions:

DVmonCXB

Mid-level Functions:

CreateDXB
OpenDXB
CloseDXB
StatDXB
ReadDHXB
FlushDHXB
CopyDHXB
ZapDHXB
CreateKXB
OpenKXB
CloseKXB
StatKXB
ReadKHXB
FlushKHXB
CopyKHXB
ZapKHXB
GetDescriptorXB
GetRecordXB
AddRecordXB
UpdateRecordXB
DeleteRecordXB
UndeleteRecordXB
PackRecordsXB
FirstKeyXB
EqualKeyXB
NextKeyXB
PrevKeyXB
LastKeyXB
StoreKeyXB
DeleteKeyXB
BuildKeyXB
CurrentKeyXB

High-level Functions:

GetFirstXB
GetEqualXB

GetNextXB
GetPrevXB
GetLastXB
InsertXB
UpdateXB
ReindexXB

Network Functions:

LockXB
UnlockXB
LockKeyXB
UnlockKeyXB
LockDataXB
UnlockDataXB
DriveRemoteXB
FileRemoteXB
SetRetriesXB

DOS Functions:

DeleteFileDOS
RenameFileDOS
CreateFileDOS
AccessFileDOS
OpenFileDOS
SeekFileDOS
ReadFileDOS
ExpandFileDOS
WriteFileDOS
CloseFileDOS
MakeDirDOS

Data packs

AccessPack
BreakPack
CopyPack
CreateDataPack
CreateKeyPack
DescriptorPack
DOSFilePack
DVmonPack
ExitPack
FieldDescTYPE
HandlePack
InitPack
MemoryPack
OpenPack
RemotePack
SetRetriesPack
StatDataPack
StatKeyPack
StatHandlePack
XErrorPack

Sample source

InitXBsrc
ExitXBsrc
AtExitXBsrc
MemoryXBsrc
BreakXBsrc
BackupFileXBsrc
StatHandleXBsrc
GetExtErrorXBsrc
DVmonCXBsrc
CreateDXBsrc
OpenDXBsrc
CloseDXBsrc
StatDXBsrc
ReadDHXBsrc
FlushDHXBsrc
CopyDHXBsrc
ZapDHXBsrc
CreateKXBsrc
OpenKXBsrc
CloseKXBsrc
StatKXBsrc
ReadKHXBsrc
FlushKHXBsrc
CopyKHXBsrc
ZapKHXBsrc
GetDescriptorXBsrc
AddRecordXBsrc
UpdateRecordXBsrc
DeleteRecordXBsrc
UndeleteRecordsrc
PackRecordsXBsrc
FirstKeyXBsrc
EqualKeyXBsrc
NextKeyXBsrc
PrevKeyXBsrc
LastKeyXBsrc
StoreKeyXBsrc
DeleteKeyXBsrc
BuildKeyXBsrc
CurrentKeyXBsrc
GetFirstXBsrc
GetEqualXBsrc
GetNextXBsrc
GetPrevXBsrc
GetLastXBsrc
InsertXBsrc
UpdateXBsrc
ReindexXBsrc
LockXBsrc
UnlockXBsrc
LockKeyXBsrc
UnlockKeyXBsrc
LockDataXBsrc
UnlockDataXBsrc

DriveRemoteXBsrc
FileRemoteXBsrc
SetRetriesXBsrc
DeleteFileDOSsrc
RenameFileDOSsrc
CreateFileDOSsrc
AccessFileDOSsrc
OpenFileDOSsrc
SeekFileDOSsrc
ReadFileDOSsrc
ExpandFileDOSsrc
WriteFileDOSsrc
CloseFileDOSsrc
MakeDirDOSsrc

The BULLET.H file

The BULLET.H file contains the exact definitions of all the BULLET data packs and functions. It is highly recomended that you refer to this file for information on the structures and (for C/C++ programmers) the naming conventions which should be used. For ease of reference, the entire contents of this file are given below. Pay particular attention to the byte-alignment requirement, because your program will not run correctly unless this is so. To differentiate Bullet for Windows from other versions, the LIB file is named WBULLET.LIB.

```
/*      bullet.h

        Defines the BULLET library's structs, consts, and function declaration


    ********************************************************************
    * NOTE: BULLET is for medium, large, or huge models (do not use tiny, *
    * ---- small, or compact since these allow for only 1 code segment).  *
    ********************************************************************

     Struct types must be standard byte packed; do not special align elements.
YOUR PROGRAM WON'T RUN CORRECTLY UNLESS ALL BULLET-USED STRUCTURES ARE BYTE-ALIGNED!
*/

#pragma pack(1)

#define __BULLET_H

#ifdef __cplusplus
extern "C" {
#endif
int     far pascal BULLET(void far *datapack);
#ifdef __cplusplus
}
#endif

#define INITXB          0           /* system */
#define EXITXB          1
#define ATEXITXB        2
#define MEMORYXB        3
#define BREAKXB         4
#define BACKUPFILEXB    5
#define STATHANDLEXB    6
#define GETEXTERRORXB   7
#define DVMONCXB        9

#define CREATEDXB       10          /* data control mid-level */
#define OPENDXB         11
#define CLOSEDXB        12
#define STATDXB         13
#define READDHXB        14
#define FLUSHDHXB       15
#define COPYDHXB        16
#define ZAPDHXB         17

#define CREATEKXB       20          /* key control mid-level */
#define OPENKXB         21
#define CLOSEKXB        22
#define STATKXB         23
#define READKHXB        24
#define FLUSHKHXB       25
#define COPYKHXB        26
#define ZAPKHXB         27
```

```
#define GETDESCRIPTORXB 30          /* data access mid-level */
#define GETRECORDXB     31
#define ADDRECORDXB     32
#define UPDATERECORDXB  33
#define DELETERECORDXB  34
#define UNDELETERECORDXB 35
#define PACKRECORDSXB   36

#define FIRSTKEYXB      40          /* key access mid-level */
#define EQUALKEYXB      41
#define NEXTKEYXB       42
#define PREVKEYXB       43
#define LASTKEYXB       44
#define STOREKEYXB      45
#define DELETEKEYXB     46
#define BUILDKEYXB      47
#define CURRENTKEYXB    48

#define GETFIRSTXB      60          /* key & data access high-level */
#define GETEQUALXB      61
#define GETNEXTXB       62
#define GETPREVXB       63
#define GETLASTXB       64
#define INSERTXB        65
#define UPDATEXB        66
#define REINDEXXB       67

#define LOCKXB          80          /* network control */
#define UNLOCKXB        81
#define LOCKKEYXB       82
#define UNLOCKKEYXB     83
#define LOCKDATAXB      84
#define UNLOCKDATAXB    85
#define DRIVEREMOTEXB   86
#define FILEREMOTEXB    87
#define SETRETRIESXB    88

#define DELETEFILEDOS   100         /* DOS file I/O low-level */
#define RENAMEFILEDOS   101
#define CREATEFILEDOS   102
#define OPENFILEDOS     103
#define SEEKFILEDOS     104
#define READFILEDOS     105
#define WRITEFILEDOS    106
#define CLOSEFILEDOS    107
#define ACCESSFILEDOS   108
#define EXPANDFILEDOS   109
#define MAKEDIRDOS      110

#define cUNIQUE         1           /* key type flags */
#define cCHAR           2
#define cINTEGER        16
#define cLONG           32
#define cNLS            0x4000      /* note: cNLS is set by BULLET */
#define cSIGNED         0x8000

#define READONLY        0           /* do NOT use O_RDONLY,O_WRONLY,O_RDWR */
#define WRITEONLY       1
#define READWRITE       2

#define COMPAT          0X0000      /* okay to use SH_DENYRW, etc. */
#define DENYREADWRITE   0x0010      /* or O_DENYREADWRITE, etc.    */
```

```
#define DENYWRITE       0x0020
#define DENYREAD        0x0030
#define DENYNONE        0x0040
#define NOINHERIT       0x0080


struct accesspack {
        unsigned        func;
        unsigned        stat;
        unsigned        handle;
        long            recno;       /* signed */
        void       far *recptr;
        void       far *keyptr;
        void       far *nextptr;
}; /* 22 */

struct breakpack {
        unsigned        func;
        unsigned        stat;
        unsigned        mode;
}; /* 6 */

struct copypack {
        unsigned        func;
        unsigned        stat;
        unsigned        handle;
        char       far *filenameptr;
}; /* 10 */

struct createdatapack {
        unsigned        func;
        unsigned        stat;
        char       far *filenameptr;
        unsigned        nofields;
        void       far *fieldlistptr;
        unsigned        fileid;
}; /* 16 */

struct createkeypack {
        unsigned        func;
        unsigned        stat;
        char       far *filenameptr;
        char       far *keyexpptr;
        unsigned        xblink;
        unsigned        keyflags;
        int             codepageid;
        int             countrycode;
        char       far *collateptr;
}; /* 24 */

struct fielddesctype {
        char            fieldname[11];
        char            fieldtype;
        unsigned long   fieldda;
        unsigned char   fieldlen;
        unsigned char   fielddc;
        long            fieldrez;
        char            filler[10];
}; /* 32 */

struct descriptorpack {
        unsigned        func;
        unsigned        stat;
```

```
        unsigned        handle;
        unsigned        fieldnumber;
        struct fielddesctype   fd;
}; /* 40 */

struct dosfilepack {
        unsigned        func;
        unsigned        stat;
        char       far *filenameptr;
        unsigned        handle;
        unsigned        asmode;
        unsigned        bytes;
        long            seekoffset;
        unsigned        method;
        void       far *bufferptr;
        unsigned        attr;
        char       far *newnameptr;
}; /* 30 */

struct dvmonpack {
        unsigned        func;
        unsigned        stat;
        unsigned        mode;
        unsigned        handle;
        unsigned        vs;
}; /* 10 */

struct exitpack {
        unsigned        func;
        unsigned        stat;
}; /* 4 */

struct handlepack {
        unsigned        func;
        unsigned        stat;
        unsigned        handle;
}; /* 6 */

struct initpack {
        unsigned        func;
        unsigned        stat;
        unsigned        jftmode;
        unsigned        dosver;
        unsigned        version;
        unsigned        osversion;
        unsigned long   exitptr;
}; /* 14 */

struct memorypack {
        unsigned        func;
        unsigned        stat;
        unsigned long   memory;
};

struct openpack {
        unsigned        func;
        unsigned        stat;
        unsigned        handle;
        char       far *filenameptr;
        unsigned        asmode;
        unsigned        xblink;
}; /* 14 */
```

```
struct remotepack {
        unsigned        func;
        unsigned        stat;
        unsigned        handle;
        unsigned        isremote;
        unsigned        flags;
        unsigned        isshare;
}; /* 12 */

struct setretriespack {
        unsigned        func;
        unsigned        stat;
        unsigned        mode;
        unsigned        pause;
        unsigned        retries;
}; /* 10 */

struct statdatapack {
        unsigned        func;
        unsigned        stat;
        unsigned        handle;
        unsigned char   filetype;
        unsigned char   dirty;
        unsigned long   recs;
        unsigned        reclen;
        unsigned        fields;
        char            f1;
        unsigned char   luyear;
        unsigned char   lumonth;
        unsigned char   luday;
        unsigned        hereseg;
        char            filler[10];
}; /* 32 */

struct statkeypack {
        unsigned        func;
        unsigned        stat;
        unsigned        handle;
        unsigned char   filetype;
        unsigned char   dirty;
        unsigned long   keys;
        unsigned        keylen;
        unsigned        xblink;
        unsigned long   xbrecno;
        unsigned        hereseg;
        unsigned        codepageid;
        unsigned        countrycode;
        unsigned        collatetablesize;
        unsigned        keyflags;
        char            filler[2];
}; /* 32 */

struct stathandlepack {
        unsigned        func;
        unsigned        stat;
        unsigned        handle;
        unsigned        id;
        char       far *filenameptr;
}; /* 12 */

struct xerrorpack {
        unsigned        func;
        unsigned        stat;
```

```
        unsigned        errclass;
        unsigned        action;
        unsigned        location;
}; /* 10 */

#pragma pack()

/* end of BULLET.H */
```

License agreement

Before using this software you must agree to the following:

1. You are not allowed to operate more than one (1) copy of this software package at one time per license. This means that if you have 10 programmers that COULD possibly use the BULLET library at the same time, you must also have ten (10) BULLET licenses.

2. You are not allowed to distribute non-executable code containing BULLET code. This means that you are not allowed to redistribute BULLET code as another .LIB, for example. Also, if BULLET code is to be contained in a Dynamic Link Library (DLL) then it must be part of a stand-alone product. This means that you cannot provide a .DLL containing BULLET code if that .DLL is to be used as a programming library for other programmers. If you wish to distribute non-executable code containing BULLET code you must obtain written permission from the author.

3. This license grants you the right to use the BULLET library code on a royalty-free basis.

4. BULLET is owned by the author, Cornel Huth, and is protected by United States copyright laws and international treaty provisions. You are not allowed to make copies of this software except for archival purposes.

5. You may not rent or lease BULLET. You may not transfer this license without the written permission of the author. If this software is an update or upgrade, you may not sell or give away previous versions.

6. You may not reverse engineer, decompile, or disassemble this software.

7. There are no expressed or implied warranties with this software.

8. All liabilities in the use of this software rest with the user.

9. U.S. Government Restricted Rights. This software is provided with restricted rights. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at 52.227-7013. Manufacturer is Cornel Huth/6402 Ingram Rd/San Antonio, TX 78238.

This agreement is governed by the laws of the state of Texas.

See: Product_Support

Product support

Support is available at my BBS, The 40th Floor, 7 days a week. Hours are **5pm to 9am**, Central Time (USA, -500/600 GMT). Weekend BBS hours are 24 hours (5pm Friday to 9am Monday). Hours other than during above periods are voice.

BBS tele#: **1-210-684-8065** (times listed above)
N81, 300 to 14.4k bps (at least)

Latest releases of BULLET are available for free download by registered users. Also available are other shareware products by me such as the Ruckus soundcard toolkit, and the LP, the linear programming optimizer. Bullet OS/2 support is coming!

My E-mail address:

Internet: **cornel@crl.com**
FTP: **ftp.crl.com /users/co/cornel**
WWW: **ftp://ftp.crl.com/users/co/cornel**

To order Bullet, use the supplied !ORDER.FRM file (!ORDER.CC for credit cards), or send check/money order, in US Dollars/US bank to:

Cornel Huth
6402 Ingram Rd
San Antonio, Texas 78238-3915
USA

Funds must be in US Dollars and must be drawn on a US bank (complete with bank routing numbers). If sending currency, use REGISTERED AirMail.

For credit card orders (Visa, MC, Amex, Discover), call **(800)242-4775** (outside US, call **(713)524-6394**), or fax to **(713)524-6398**, or via CompuServe e-mail to **71355,470**, requesting Bullet for Windows (item# **11185**). This is for ORDERING Bullet. Do NOT use these numbers for any other purpose! Be sure to specify "Bullet for Windows".

Price is $**89** plus $1 AirMail shipping to all destinations (anywhere in the world). For fastest delivery, *add* $**15** for USA/Canada-only Next-Day Express Mail, or *add* $**20** for International Express Mail. Standard AirMail is 3-5 days for US destinations, and 5-14 days for international destinations. Disk media is 3.5-inch, 1.44M unless requested otherwise.

See: Troubleshooting

Troubleshooting

Common problems encountered by new users of Bullet are:

P: Data fields are skewed, with some fields containing parts of other fields' data.

S: Most of these problems relate to the way C strings are 0-terminated, while dBASE DBF fields are fixed-length, and so need not have the \0. If you are going to use strcpy, or any other C run-time library that requires null-terminated strings, then be sure to provide for this when you define your DBF record. For example, if you want two fields, LASTN and FIRSTN, to each have 15 characters, but will be using C strings, define the fields as 16. This will allow room for the \0 on your strings, and give 15 true character places.

Often, the case is such that all appears normal when the DBF file is accessed (i.e., the data fields don't appear to be skewed), but indexed access is not correct. This is because your program code has templated the data record consistent with what was written (and is read) at the DBF level. However, it is not consistent with the DBF header specifications (written in CreateDXB) and so any external access to the DBF (including Bullet keyed access) will not be correct.

Skewed data also results when you change either your program's data structure template of the DBF record, or your DBF field list definitions, but don't do the same to both. The physical record buffer in your program must exactly match that of the DBF record. Off the subject a bit, but possibly on your mind: yes, you can create all this on the fly (i.e., not hard-coded fieldnames, etc.), but it does require that you manage all offsets yourself, rather than have the compiler do it for you. A job for a C++ wrapper, I suppose. Certainly do'able as I''ve done it using a BASIC compiler (see IDEMO.*)

Another reason for skewed fields is that the compiler is modifying the alignment of your program's DBF record structure. It is imperative that you instruct your compiler to not modify any Bullet-used structure: #pragma pack() is used by many compilers. Consult your compiler documentation. For non-Bullet structures, alignment doesn't matter to Bullet. Bullet structures are any of the Bullet packs, and any DBF record template/buffer/structure that your program uses to communicate with Bullet.

The main point is to ensure that your physical data record (structure) you use in your C/C++ program exactly matches, field for field, that of the DBF record on disk.

P: Error 240 occurs when attempting to index or reindex when all else seems to be right.

S: dBASE specifications require that fieldnames of the DBF be 0-filled. Some programs create DBF files that do adhear to this specification, but instead just null-terminate the fieldname. These files must be fixed before they can be used by Bullet. When creating Bullet DBF files, you must 0-fill the fieldnames (11 bytes, 10 bytes data -- byte 11 always=0). Memset() can be used to quickly 0-fill the entire fieldlist.

P: Find an exact match on a GetEqualXB.

S: The problem usually stems from index files created to allow non-unique keys. Personally, I don't create index files where the primary key is not unique, but the demand was there and so Bullet provides for this. However, since there's no real way to offer indexing on physically indentical keys, Bullet adds an enumerator word at the end of each key (only if the index file is not using unique keys). This two enumerator is in high byte/low byte order, and is incremented for each duplicate key in the index file. For example, if KINGM is added as a key, and is found to be the first such key by Bullet, the actual physical key is KINGM\0\0, where the enumerator is the last two bytes (in Motorola-word order for proper collating). Upon the next insert of a "KINGM" key, Bullet uses KINGM\0\1. And so on.

To search for this key, which by the way is 7 bytes long, and so reported by StatKXB, though logically it is only the 5 bytes (as in strcpy(keyExp,"SUBSTR(LASTN,1,5)") with non-unique keys permitted), you must account for this in your keybuffer. For example, in your global scratch keybuffer (or you can use a separate buffer for each index file, up to you) you must specify all 7 bytes, not just the first 5. This because the data at bytes 6/7 may not equal \0\0 or \0\1, and so these keys will not be found (error 201). In this particular case, where duplicate keys exist, and you want to locate a KINGM record, you really have no choice but to start with strcpy(keybuff,"KINGM\0\0", be aware of the "extra" \0 put by the strcpy function) and GetEqualXB, check the data record returned, and if not the one wanted, do GetNextXB until the desired record is located (or not). Alternatively, you could specify "KINGM\255\255" and use GetEqualXB followed immediately by GetPrevXB (enumerator \255\255 is the very last possible, and so will never be found, but will setup the internal gears so that the GetPrevXB will locate the very last existing "KINGM??").

All this is really only a concern if you permit duplicate keys. The above could be made unique by adding expanding the key expression. For example, a key based on SUBSTR(LNAME,1,5)+SUBSTR(SSN,6,4) would be pretty unique (first 5 characters of last name followed by last 4 SSN numbers (used as characters)).

Q: How do I specify the key I want to find?

A: The keybuffer (AP.keybuffptr locating it) is filled with whatever data type you want to locate (hence, keybuffer is of void type). If you want to locate a character key, simply specify the key, such as strcpy(keybuffer,"whatever") (see "Find an exact match on GetEqualXB", above). If you are locating a binary key value, place the binary value, with the appropriate cast, into the keybuffer: e.g., *((long *)keybuffer) = 5L;.

These problems are the most frequently expressed to me. If you are having other problems related to getting Bullet going, feel free to contact me at the locations listed under Product Support. My ability to quickly respond to your requests is directly related to how thorough you are in expressing the exact cicumstances of any problem.

Complete sample programs are in short supply. Experienced C or C++ programmers that would like their "Bullet wrappers" possibly included in future releases are requested to contact me. Additional samples will be made available on the BBS, available to all. Of course, registered Bullet users will have access to the best of the best, and always have free access to the current versions of Bullet.

When you've done everything and it's still not working, it's time to read the documentation!

What is BULLET?

BULLET is a program module that handles the details of putting information to and getting information from your hard disk using a standard data file format called the Xbase DBF format with very fast and efficient index file routines. It can be used as-is by most DOS compilers.

See: What is a database?

What is a database?

A database is a collection of data arranged so that the it can be accessed as useful information. For example, let's say we have two files. Each consists of two fields. The first file has codenumber and score. The second file has a codenumber and name. Separately, the files are merely a collection of data. Together, however, they tie the name to the score:

score	**codenumber**	**codenumber**	**name**
99	100	100	John
87	155	105	Paul
66	125	110	George
:	:	:	:

Codenumber 100 is John, who scored 99. The other members scores are not in the abbreviated data file listing.

A database can be a single data file but more often it is a group of related data files and usually these data files are indexed by keys (in the index file, also called key file) so that very fast, direct access is possible.

Ringo.

See: What is DBF?

What is DBF?

DBF is the file extension of dBASE III-compatible data files (filename.DBF). The file format is used by dBASE IV, FoxPro and many other database programs. Many programs can also use the file format to import/export data using it. The DBF format is the most common data file format used on PCs.

A DBF-compatible data file consists 3 distinct areas. First is the data header. This contains information such as the number of records in the file. Second is the field descriptors. These descriptors define the makeup of each field in the record. The third is the record area. Each record is a logical unit of data.

For example, a record, all of which are made up of the same fields but with different data, could (conceptually) look like this:

	field 1	**field 2**	**field 3**	**field n**
record 1	Johnson	Larry	465310555	...
record 2	Aberdeen	Zara	465230555	...
record n	...	...	...	...

See: What is a Btree?, DBF Specs

What is a Btree?

A b-tree is a sorting method ideally suited to data structures maintained on a hard disk. It is very fast on retrieval and is inherently self-balancing during inserts and deletes. Self-balancing ensures performance remains consistent.

The reason the b-tree is ideally suited to hard disks is that, when looking for a particular key, most of the time involved in accessing the key is spent by the hard drive moving to various locations on the disk. The task of a good access method is to reduce the number of seeks that the disk must perform. The b-tree accomplishes this by maintaining several keys (perhaps 50) on each node, with the necessary pointers to previous and following nodes. A b-tree of order 20 (19 keys per node) can find a key in a file of 1,000,000 keys in a MAXIMUM of 5 disk accesses, where each disk access visits a node.

```
'BASIC program to find *max* seeks needed/avg time
Keys& = 1000000: KeysPerNode = 19: AvgSR = 25
Order = KeysPerNode + 1
max = (LOG((Keys& + 1) / 2) / LOG(Order / 2))
PRINT "Max nodes accessed for"; Keys; "keys & b-tree of order"; Order;
PRINT "is"; max; "nodes"
PRINT "Max disk time based on avg seek+read of"; AvgSR;
PRINT "ms is"; AvgSR / 1000 * max; "seconds"
```

See: What is a network?, Index specs

What is a network?

A network is a group of computers able to communicate with one another. Often called a LAN (local area network), a network allows resources to be shared. Sharing resources can lead to problems if steps are not taken to ensure that two computers don't try to share the same resource at the same time. For example, say two computers try to change the same record in a file on a network drive. Let's say both users are accessing the number of widgets in inventory. The first user gets there a micro-second before the second and allocates the last widget in stock. The second user comes in right after and, since the first user has not yet updated the inventory, allocates the very same widget. One widget, two users. When the first user updates the inventory, widgets in inventory is changed to 0 (previous - 1). The second updates the inventory in the same manner and sets widgets to 1 less what it was when it started, or 0 also. You see the problem.

In order to successfully share a file on a network, the file must first be locked to a single user. Once that user has locked the file, he has sole access to the data within it and he will not experience the scenario above. When the user has completed the changes, he unlocks the file so that others may use it.

See: What is file locking?

What is file locking?

File locking is a means to obtain exclusive access to a file. This is needed in cases of multiple programs or users accessing a shared file at the same time.

There are several methods to ensure only one process or user has access to a file. The first method is to open the file so that while the file is open only your program can access any part of it. This is simple to implement and the operating system handles the details of this. However, it requires your program to open/close files all the time since no other process may access the file while it is open.

Another method is to use byte-level locks. Also managed by the OS, this method allows for restricting access to any particular region within the file. Which regions are to be locked is to be determined by your program, however, and it can be complex to perform multiple locks at the byte, field, or record level.

Another use of the byte-level lock is to specify that all bytes within the file are to be locked. This greatly simplifies the process of obtaining a lock and has the advantage over a file access lock of not needing to open/close the file for each lock. It is very fast and easy to implement. BULLET offers all three lock types.

What is NLS?

NLS stands for National Language Support. This feature is available in DOS 3.3 and later. BULLET makes use of NLS by getting from DOS the current DOS country collate-sequence table. The collate table is used to properly sort mixed-case character strings and also foreign (or non-USA) language character strings according to that country's alphabet. This is an option but is recommended.

In addition, BULLET provides for a programmer-supplied collate-sequence table.

How to design a DB

To design a database, above all else, know what information you require from it. Having established what you need to know, collect the data that lets you formulate this into useful information.

For example, you want to track a class of students and determine how well they achieve on tests. The criterion you use is the test score. You determine that your data is 1) students, 2) tests, and 3) test scores. Too simplify, you use a single 20-character field for student, a numeric field for test number (1 to the n tests), and a numeric field for test scores (0 to 100).

Since the objective is to track students' scores, arrange the data so that output consists of each student's score in test order. Do this by specifying an index file containing an index based on the student's name and test number:

```
char keyexpression[136];
strcpy (keyexpression,"STUDENT+TEST");       /* use two-field key */
```

By using the routines of the database langauge, you can easily create the data and index files, add data, list student's scores, or make changes to the database. Note: these *How to* examples are meant only to show the basis behind an operation.

See: How to create a DB, CreateKXB

How to create a DB

Having defined the database, create it. First, create the datafile based on the 3 fields you defined in your design. To do this, allocate an array for the field descriptors for the number of fields (see also CreateDataPack):

```
struct fielddesctype fieldlist[3];
memset(fieldlist,0,sizeof(fieldlist); /* must be 0-filled */
strcpy(fieldlist[0].fieldname, "STUDENT");
fieldlist[0].fieldtype='C';
fieldlist[0].fieldlen = 20;
fieldlist[0].fielddc = 0;
strcpy(fieldlist[1].fieldname, "TEST");
fieldlist[1].fieldtype='N';
fieldlist[1].fieldlen = 1;
fieldlist[1].fielddc = 0;
strcpy(fieldlist[2].fieldname, "SCORE");
fieldlist[2].fieldtype='N';
fieldlist[2].fieldlen = 3;
fieldlist[2].fielddc = 0;
```

The fieldlist is a struct member in CDP (CreateDataPack) as in

```
CDP.func = CREATEDXB;
CDP.fieldlistptr=fieldlist;
```

Call CreateDXB to create the data file. To create the index file, first open the data file just created, then call CreateKXB to create the index file. Open the index file so we can use it (data file is already open).

See: How to add to the DB

How to add to the DB

Once you have the database designed and the data and key files created and open you can start putting the student's test data into it. Note that the DBF-format requires that all data in a data file be in ASCII format. This means that we must convert the numeric test score into its ASCII form. C has the itoa() function to do this. In addition, numbers generally should be right-justified in their field.

Note that BULLET does not require that you use only ASCII field data. If you want to forgo dBASE compatibility, binary data can be used directly in the fields.

See: How to query the DB, InsertXB

How to query the DB

Now that you have data in the database you want to see what's in there. Since the index file is in "STUDENT+TEST" order, the information we'll be getting out of the database is in Student name order, with each student's scores in test number order.

If we want to look at all the students, we can use GetFirstXB to retrieve the first student's score for the first test. GetNextXB retrieves the next record (the first student's score for the second test), and so on. When all records have been retrieve GetNextXB returns an End Of File error code.

If we want to look at a particular student's score only, we can use GetEqualXB to go directly to a student's first test score. GetNextXB get his next and so on until GetNextXB retrieves the next student's first test score. You can stop at this point (student names no longer match).

We might also want to find all students who scored less than 65 on any test. To do this we can GetFirstXB, check SR.score for < 65 and if so print that record. Continue by using GetNextXB, printing each record that has a score < 65.

See: How to update the DB

How to update the DB

To update a particular record in the database we must first locate and identify it using one of the get routines such as GetEqualXB. The Get() routine return the record data, and also the physical record number of the record accessed, into the AccessPack RecNo. Having used one of the Get() routines to read the data record from disk to memory, you can make any changes to the data record in memory. E.g., if a student's score needs to be changed from a 69 to a 96, first find the record (and its RecNo), then update the score field.

Note that any change to a key field will initiate a key file update automatically.

See: How to delete a record

How to delete a record

To delete a particular record in the database we must first locate it using one of the get routines such as GetEqualXB. These Get() routines return the actual record number of the data record accessed by Get() into the AccessPack RecNo. Having used one of the Get() routines to find the data record, make a call to the delete function.

The DeleteRecordXB routine does not physically remove the record from the data file but instead tags it as being "deleted".

How to call BULLET

BULLET is called through a single entry point. The only argument passed to it is a far pointer to the control pack using the Pascal calling convention. The first two entries in this pack are the function to be performed and the function return status. BULLET is a function call returning an integer status value. See BULLET.H for more.

Each function (or routine) uses a prescribed pack format. For example, some routines need only know the handle of the file, along with the function number itself. So, to flush a data file, for example, you would do the following:

```
struct handlepack HP;
HP.func = FLUSHDHXB;              /* defined as a CONST in BULLET.H*/
HP.handle = file2flushhandle;     /* handle returned from Open() routine */
rstat = BULLET(&HP);              /* do the actual call to BULLET */
```

The value of rstat is set to the completion code as returned by the FlushDHXB routine. It is the same as the value returned in HP.stat **IN ALL BUT A FEW** cases: InsertXB, UpdateXB, ReindexXB, and LockXB. These routines return not the actual error code, but rather a transaction index number of the access that failed. See those routines for more information.

Note: This help file (and the original CZ help file) use upper and lower case in order to make the routine names and packs more readable. However, the BULLET.H file defines all the routines (constants) in upper case, and the pack names in lower case.

Overall specs

BULLET is dBASE III/III+/IV .DBF-compatible. This format is compatible with a large base of software programs including the latest database packages such as dBASE IV and FoxPro. Spreadsheet packages such as Excel and 1-2-3 can directly import BULLET DBF data files, too. And because of BULLET's versatility, it can also create very non-standard data files. This may be a useful feature if data secrecy is of concern.

BULLET requires MS-DOS 3.30 or above. It uses 19K of code/static data space and requires at least 40K of workspace. 140K of workspace is ideal.

Overall Specifications:

DBF		**INDEX**	
Max records:	16,777,215	Max nodes:	65,535
Record length:	2-4000 (8192)	Max keys:	4M
Max fields:	128 (255)	Key length:	1-64
Field length:	1-254 (255)	Max key fields:	16

Total open index plus data files can be up to 255. Numbers in () indicate extended specifications.

See: DBF specs

DBF specs

To remain compatible with other dBASE III .DBF platforms you should restrict your data files to the following specifications:

File ID byte: 3 (83hex if .DBF has memo field, not currently supported)
Max record size: 4000 bytes Max fields/rec: 128 Max field size: 254 bytes
Allowable field name characters: A-Z and the _ (upper-case), 0-filled
Allowable field types:

- **C** character, 1-254 bytes
- **D** date, 8 bytes, in the format YYYYMMDD (19920531)
- **L** logical, 1 byte, either space, "T" or "Y", "F" or "N"
- **M** memo, 10 bytes, used as pointer into .DBT file (currently not supported)
- **N** numeric, 1-19 bytes, ASCII format, uses explicit decimal if needed...
  ...decimal places may be 0, or 2 to (field size - 3) but no more than 15

Restrict all data in .DBF fields to ASCII. This means you should convert binary data to the equivalent ASCII representation, e.g., if you have the binary value 22154, it must first be converted to the string "22154" before you can store it to the .DBF data file. So, while your in-program code deals with binary data, your I/O code must convert it to/from ASCII. This is a dBASE-compatibility issue only. If you can forgo these requirements you can use binary fields, any-character field names, record sizes to 8192 bytes, and up to 255 fields.

A dBASE III .DBF is composed of 3 sections: the header, the field descriptors, and the data area.

The header structure (first 32 bytes of file):

Name	Type	Offset	Meaning
FileID	byte	0	data file type id, 03 standard (43,63,83,88h)
LastYR	byte	1	last update year, binary
LastMo	byte	2	last update month, binary
LastDA	byte	3	last update day, binary
NoRecs	long	4	number of records in file
HdrLen	word	8	length of header, including field descriptors, +1
RecLen	word	10	length of data record including delete tag
internal	byte	12-31	reserved

The last update values are updated to the current date whenever the .DBF file is flushed or closed. Likewise, the NoRecs value is updated whenever a record is added to the .DBF. The FileID is specified when you create the file, HdrLen and RecLen are computed and stored when the file is created, too.

The field descriptor format (follows header, one per field):

Name	Type	Offset	Meaning
FieldName	char	0	field name 10 ASCII characters, A-Z or _ (0-filled)
0T	byte	10	field name zero-termintor (must be 0)
FieldType	char	11	field type (C D L M N)
internal	long	12	reserved
FieldLen	byte	16	length of this field
FieldDC	byte	17	decimal count
internal	byte	18-31	reserved

The unused bytes in the FieldName must be set to zeroes (CHR$(0)).

Each field is described by a 32-byte descriptor. The first field's descriptor starts right after the header

proper, at offset +32. After the last field descriptor is data byte ASCII 13. (Note: the orginal dBASE III has a 0 byte following this ASCII 13.) Immediately following this is the actual record data.

The data record format:

The first record is located at offset HdrLen (from the header). The first byte of each record is a delete tag. This tag is maintained by the BULLET routines. A space, ASCII 32, means the record is not deleted; an asterisk, ASCII 42, means the record has been deleted (marked as deleted, often this is used as a method to temporarily tag records, for whatever purpose).

Following the tag is the data for each field, not delimited (i.e., the fields run together without anything separating them). The second record is at offset HdrLen+reclen. The start offset of any record in the file can be computed as (recordnumber - 1) * reclen + HdrLen. All data is in ASCII form.

An EOF marker (ASCII 26) is placed at the end of the last record.

See: Index specs

Index specs

BULLET uses a proprietary, modified b-tree index method to manage the index files. The supported key types are:

Type	Length	Meaning
Character	1-64	ASCII, NLS, or user-supplied sort table
Integer	2	signed or unsigned 16-bit value
Long Int	4	signed or unsigned 32-bit value

In addition to the above types, BULLET allows for unique or duplicate keys in the index file. If duplicates are allowed, BULLET enumerates each key with an enumerator word (see FirstKeyXB).

The key may be composed of up to 16 character fields or substrings within those fields. Numeric fields are considered character fields by BULLET unless the key is set to binary (see KeyFlags below). Integer or LongInt binary keys can be composed of a single field only. The key expression is specified in text (e.g., "LNAME+SUBSTR(FNAME,1,1)+MI") and is fully evaluated when the index file is created.

A BULLET index file is composed of 3 sections: the header, the collate-sequence table, and the node/key entry area.

The header structure:

Name	Type	Offset	Meaning
FileID	byte	0	index file type id, 20
RootNode	word	1	root node number
Keys	24bit	3	number of keys in index file
AvalNode	word	6	node number available for reuse
FreeNode	word	8	next free node number
KeyLen	byte	10	key length
NodeKeys	byte	11	number of keys that fit on a node
CodePage	word	12	code page ID
CtryCode	word	14	country code
internal	byte	16-21	reserved
KeyFlags	word	22	key flags
KeyExprn	byte	24-159	key expression
internal	byte	160	reserved
KeyXFlds	byte	161	number of fields used by key (1-16)
KeyXlate	byte	162-225	translated key expression
internal	byte	226-253	reserved
CTsize	word	254	collate-sequence table size

The collate-sequence table structure:

table	byte	256-511	sort weight table of ASCII character 0-255

Node/key entry structure (first entry is in node #1, file offset 512):

```
2A 0A 00 KEY123 7B 00 00  12 00 KEY178 B2 00 00 0C 00 ...
1. 2.    3.     4.        5.    6.     7.       8. 9.
```

1. Key count for that node (first byte of each node)
2. 16-bit node back pointer (for non-leaf nodes, 0 if leaf node)
3. First key value, "KEY123" in this case
4. 24-bit data record pointer (low word/hi byte) 7Bh = DBF record number 123
5. 16-bit node forward ptr/back ptr (for non-leaf nodes, 0 if leaf node)

--in this case, it indicates that the key following KEY123 is in node #12h
--and also that the key before KEY178 is in that node as well

6. Second key (here "KEY178")
7. 24-bit data pointer (record number in DBF)
8. 16-bit forward node pointer (for non-leaf nodes, 0 if leaf node)
9. Repeat 6 to 8 for each key on node. (node size is 512 bytes)

As in many b-tree implementations, BULLET's index files maintain an average load percentage of approximately 66%. This means that in any given node, 66% of the available space is in use. The free space in the node is attributable to the constant reshaping of the file as keys are inserted or deleted, causing the nodes to be split and merged. A split will occur when an insert needs to add a key to an already full node; a merge will occur when a neighboring node is small enough to be merged into a just split node. This constant prune-and-graft of the b-tree results in a node load of about 66% (50% in degenerate cases such as with already sorted data). It's this aspect of the b-tree that makes it a consistent performer and a widely-used method of managing index files.

The following formula can be used to determine the number of keys that an index file can hold:

```
MaxKeys = MaxNodes * MaxKeysPerNode * LoadFactor
MaxKeys = 65535 * 509/(keylen+5) * .66
```

The load factor can be increased to ~95% by using the ReindexXB routine. This load factor results in superior retrieval speeds since there are more keys on each node. Insertion speed will be decreased, however, since splitting will occur more frequently, though perhaps not noticeably.

Memory specs

BULLET allocates memory on an as-needed basis. When linked to an executable program, BULLET makes use of 17.5K of code space and about 1.5K of static DGROUP data space. To accomodate the wide variety of compilers, BULLET's API structure will have the linker included all of the library into your final EXE program.

All runtime memory allocations are obtained from the operating system (the far heap).

The amount of memory that BULLET requires is based on which routines are used. See the next screen for a list of the routines that make malloc calls to the operating system and how much memory they require.

Routines making dynamic memory allocations and amount (within ± 16 bytes):

Routine	Bytes	Basis
InitXB	272	permanent, released when program ends (JFTmode=1)
BackupFileXB	32K	temp, released when routine exits
CreateDXB	48+(NF*32)	temp, released when routine exits (NF=NoFields)
CreateKXB	544	temp, released when routine exits
OpenDXB	144+((1+NF)*32)	semi-permanent, released when file closed
OpenKXB	1264	semi-permanent, released when file closed
PackRecordsXB	RL to 64K	temp, released when routine exits (RL=RecLength)
ReindexXB	32K to 128K	temp, released when routine exits
UpdateXB	2K+RL	temp, released when routine exits (RL=RecLength)

For example, when BackupFileXB is called it attempts to allocate 32K from the OS. If 32K is not available, BackupFileXB returns with an error code of 8 (DOS error #8, not enough memory). If you won't be using Backup or Reindex, BULLET can make do with much less memory (use table above).

Needed stack space is 4K (max) for ReindexXB. Other routines can operate with less than 1K of stack space. In other words, stack use is minimal.

OS calls specs

BULLET makes use of the following operating system calls:

INT21/25	DOS_setvector	INT21/44/0B	DOS_setsharingretrycount
INT21/2A	DOS_getdate	INT21/48	DOS_malloc (*)
INT21/30	DOS_version	INT21/49	DOS_free (*)
INT21/35	DOS_getvector	INT21/51	DOS_getpsp
INT21/39	DOS_makedir	INT21/56	DOS_renamefile
INT21/3D	DOS_openfile	INT21/59	DOS_getextendederror
INT21/3E	DOS_closefile	INT21/5A	DOS_createtempfile
INT21/3F	DOS_readfile	INT21/5B	DOS_createnewfile
INT21/40	DOS_writefile	INT21/5C	DOS_lockunlockfile
INT21/41	DOS_deletefile	INT21/65/01	DOS_getextendedcountryinfo
INT21/42	DOS_movefileptr	INT21/65/06	DOS_getcollatesequencetable (+)
INT21/44/09	DOS_isdriveremote	INT21/67	DOS_sethandlecount
INT21/44/0A	DOS_isfileremote	INT2F/10/00	DOS_isshareinstalled

No other operating system calls are made. No BIOS calls are made.

(*) **Note:** In the Windows version, BULLET calls the GlobalAlloc/GlobalLock and GlobalUnlock/GlobalFree functions for memory allocation.
Furthermore, all calls to INT21 are done via the DOS3Call interface, which according to the SDK documentation is somewhat faster than calling INT21.

(+) **Note:** In the Windows version, BULLET makes a call to DPMI (INT31/2) to obtain a valid selector for the pointer to the collate table returned by function INT21/6506 (a DOS service emulated by Windows -- the real INT21/6506 is not issued by Windows). Since this emulation is flawed in standard mode, Bullet requires that Windows be in enhanced mode before attempting this DPMI call. Therefore, your program must be running under enhanced mode when calling CreateKeyXB or an error is returned. You are permitted, however, to call CreateKeyXB in standard mode provided that you supply a collate table pointer for Bullet to use (meaning you can run a Bullet program from standard mode provided that you supply a collate table for Bullet, since only here is enhanced mode required).

OS specs

BULLET is currently available only for MS-DOS and compatible operating systems, and Microsoft Windows. It requires DOS 3.3 or higher, or Windows 3.x or higher.

To provide efficient memory use, BULLET uses a single-buffer cache per index file. The single-buffer cache also provides for very quick network access since a minimum amount of memory needs to be flushed when releasing control of BULLET files. For maximum speed, however, an external high-performance disk cache can be used.

If you do not use a disk cache then it's recommended that you set your BUFFERS= statement in CONFIG.SYS to at least 20 or 30. Even without a disk cache, BULLET is still very fast. Also, be sure to set your FILES= to the number of files that you'll be opening at any one time. If you set FILES=20 you can have BULLET open 14 files (CZ.COM uses 1 and DOS reserves 5 more). You can set FILES=255 allowing BULLET to open up to 249 files at one time.

DO NOT set FILES= to a value greater than 255.

Network specs

BULLET currently operates on all DOS-compatible network platforms.

Be sure to install SHARE.EXE (or VSHARE, or compatible) on the server and, if you are mutlitasking or using the locking functions, on your local machine, also. If you'll be opening many files you should extended the default SHARE file-sharing information space and the number of locks that can performed at one time. The DOS 5.0 default is /F:2048 and /L:20. This allocates 2K for file-sharing info space and allows 20 consecutive locks to be active. If the F: value is too low, error 5 (extended error 32) is returned on an open attempt. If you extend the JFT in InitXB and plan to use many files, say more than 50, be sure to extend /F: by 2K for every 50 additional files and set the /L: to the number of files you plan on having open. If L: is too low, error 1 (ext err 36) is returned on a lock attempt.

As an example, if you'll be using 100 files, set FILES=106 in CONFIG.SYS, set SHARE /F:4096 /L:106, and IP.JFTmode=1 for InitXB. These values are a minimum. If you have more than one process active, you need to account for other apps.

Note that Windows always returns a "SHARE is installed" using the DOS detection routines used by BULLET. To determine if SHARE is actually installed, attempt to perform a lock using one of the LockXB routines. An error code indicates that SHARE (or compatible) is not installed.

InitXB

Pack: InitPack **Src:** InitXBsrc **Func:** 0/System

Before using any routine you must initialize the BULLET file system.

If you want more than the standard number of file handles, set InitPack.JFTmode to 1. This expands the current process's Job File Table to allow 255 open files maximum.

On return the DOS version (INT21/30h) is in InitPack.DOSver. Major version in the high byte. Minor in the low. The BULLET version (*100) is returned as is the address of the ExitXB routine. You can use this address to register ExitXB with your own _atexit function if your runtime library does not provide _atexit already.

The osversion field defines whether the library being used is the DOS version (in which case it will be 0) or the Windows version (the field will be set to 1).

Note: _atexit is a routine available in most DOS, OS/2, and ANSI runtime library code and is called just prior to the program ending. See AtExitXB for information on what to do if your library does not have _atexit.

ExitXB

Pack: ExitPack **Src:** ExitXBsrc **Func:** 1/System

Before ending your program you should call ExitXB to close any open BULLET files. This also will release any memory still allocated to those files. This restores the default keyboard break handlers if they were changed.

In normal operation you would see to closing all files yourself. However, if your program fails to reach the programmed end, it's very possible that files may still be left open. It is essential that you properly close all BULLET files before ending. There are two methods to achieve this:

1. Direct you startup code so that on fatal errors, your program executes ExitXB before returning to DOS.

2. Use AtExitXB to automatically register ExitXB to be executed in the normal shut-down code of the compiler. This method is preferred.

See: InitXB

AtExitXB

Pack: ExitPack **Src:** AtExitXBsrc **Func:** 2/System

Used to automatically close all BULLET files, release allocated memory, and restore the default Break handlers when your program ends. Your compiler generates specific code to be executed in the course of ending your program. AtExitXB registers the ExitXB routine to be performed in this compiler-generated code.

This routine is standard in most DOS, OS/2, and ANSI runtime libraries. If yours does not have _atexit, then you must link with the supplied NOATEXIT.OBJ file: C>link YOURPRG + NOATEXIT, ...

You can tell if your compiler doesn't supply _atexit at link time. LINK reports '_atexit' : unresolved external. Add NOATEXIT.OBJ as described above.

Be sure that your _atexit routine is for the medium, large, or huge memory models since BULLET uses multiple code segments and far calls.

See: BreakXB

MemoryXB

Pack: MemoryPack **Src:** MemoryXBsrc **Func:** 3/System

This is the only BULLET routine that can be used before InitXB. It reports the largest free block of memory available from Windows, but reports available memory greater than 960K as 960K (i.e., the memory available may be 10MB but this routine will never indicate more than 983,040 bytes). This routine is essentially of no use in Windows.

See: OpenDXB, OpenKXB

BreakXB

Pack: BreakPack **Src:** BreakXBsrc **Func:** 4/System

Disables system response to Control-C and Control-Break keys preventing users from inadvertently exiting the program without first doing a BULLET shutdown.

It's REQUIRED that you reinstate the default break handlers with this routine before ending your program. ExitXB automatically reinstates the default break handlers.

This routine will not disable Control-Alt-Delete (a warm-boot). If the user is at this point, he may prefer to exit via a warm-boot rather than reset the machine.

This routine will not surpress the ^C displayed by DOS. If you don't want the ^C to be displayed move the cursor to a location off-screen, say, row 26.

BackupFileXB

Pack: CopyPack **Src:** BackupFileXBsrc **Func:** 5/System

Copy an open BULLET key or data file. BULLET repacks and reindexes files in place, requiring less disk space to perform the function. BackupFileXB allows you to safely copy a file before doing this.

This function is recommended prior to packing a data file with PackRecordsXB since the data is very valuable. There is probably little need to do so when reindexing an index file since index files can be constructed very easily from the data file but a CopyKHXB to preserve the key expression is quick and recommended.

See: ReindexXB

StatHandleXB

Pack: StatHandlePack **Src:** StatHandleXBsrc **Func:** 6/System

Get information on a DOS file handle number to determine if it is a BULLET file and if so, if that file is a BULLET key or data file.

If the returned ID value is 0, the handle is to a BULLET index file. ID=1 then the handle is a BULLET .DBF file. ID= -1 then the handle is not a BULLET file.

See: CreateDXB, StatDXB, StatKXB

GetExtErrorXB

Pack: XErrorPack **Src:** GetExtErrorXBsrc **Func:** 7/System

Get the extended error information for the last operation. This information includes the extended error code, the error class, the recommended action, and the location of the error. See DOS errors for the extended error meaning and for the class, action, and locus code meanings.

Note that on fatal DOS errors, such as an open floppy drive door, the extended error code returned is 83 - fail on INT24. This indicates that the INT24 handler was invoked by DOS and that the INT24 handler told DOS to ignore the error. (BULLET invokes its own INT24 handler each time it accesses the DOS file system and restores it promptly after the access.) In such cases, this extended error code is less informative than the standard return code and, the other 'extended' information should be disregarded. (In fatal DOS errors the standard return code IS the extended error code.)

This routine returns the extended error information for the LAST DOS system error. This information remains the same until the next DOS system error.

DVmonCXB

Pack: DVmonPack **Src:** DVmonCXBsrc **Func:** 9/DEBUG

Control BULLET debug monitor.

This routine is available only in the debug engine.

The monitor displays in realtime the state of a data file handle, or an index and data file handle pair if an index handle is specified. DVmonCXB is best used on dual-display systems in which the video output is sent to the secondary video monitor. In any case, a 4000-byte screen image is updated in real-time.

To use the monitor, set mode=1, handle=file to monitor, and VideoSeg=segment address of 4000-byte area. The typical VideoSeg would be to video memory. If you have a color system as the main monitor and a mono as the secondary, set VideoSeg=0xB000. Detailed system stats are continually updated to the secondary monitor. If you have a single monitor with at least 2 video pages, set VideoSeg to your base address plus the page size\16, typically 0xB800+(4096\16). If you have only a single-page video system, you can allocate a 4000-byte memory area and update the video manually by moving it to your video display (80x25).

CreateDXB

Pack: CreateDataPack **Src:** CreateDXBsrc **Func:** 10/Mid-level

Create a new BULLET .DBF data file. Before using this routine allocate a field description array of TYPE FieldDescTYPE for at least as many fields as are in the record. It's recommended that this allocation be zeroed before use since the fieldnames must be 0-filled.

Conventional dBASE .DBF files have a FileID=3. Other possible FileIDs that you may come across are (in hex):

43h __ are special-use Xbase IV DBF files, BULLET can process these file IDs
63h / since they are similar to ID type 3
83h --- this DBF file has an Xbase III/III+ memo field/file
88h --- this DBF file has an Xbase IV memo field/file

In creating your .DBF files, specify FileID=3 to ensure compatibility across Xbase versions.

BULLET makes no special use of the FileID byte.

See: OpenDXB, CreateKXB

OpenDXB

Pack: OpenPack **Src:** OpenDXBsrc **Func:** 11/Mid-level

Open an existing .DBF data file for use. You need to specify two things, the filename and the DOS file access mode. If the open succeeds, the DOS file handle is returned. Use this handle for all further access to this file.

Each .DBF data file you open allocates 144+((1 + number of fields) * 32) bytes for internal use. This memory is not deallocated until you close the file with CloseDXB or execute ExitXB.

You must open the data file before you can open (or create) any of its index files.

See: OpenKXB

CloseDXB

Pack: HandlePack **Src:** CloseDXBsrc **Func:** 12/Mid-level

Close an existing .DBF data file for use. Closing the file updates the file header and deallocates the memory used by this file.

You MUST close all BULLET files before ending your program or file corruption may occur. To ensure that all files are closed in the event of an unscheduled program termination, use AtExitXB.

See: StatDXB, ExitXB, CloseKXB

StatDXB

Pack: StatDataPack **Src:** StatDXBsrc **Func:** 13/Mid-level

Get basic information on the BULLET .DBF data file handle specified. Information returned includes the number of records in the file, the record length, number of fields per record, and the date the file was last updated.

Typically, your program will keep track of whether a particular handle belongs to a data file or a key file. In cases where this is not possible, call the StatHandleXB routine to determine what file type a handle is.

Note that a just-created data file will have the LastUpdate date set to 0/0/0.

See: StatKXB

ReadDHXB

Pack: HandlePack **Src:** ReadDHXBsrc **Func:** 14/Mid-level

Reload the disk copy of the data header for the opened .DBF data file handle to the internal copy.

In single-user, single-tasking systems this routine is not needed. However, in a multi-user or multi-tasking system it's possible, and desirable, for two or more programs to use the same data file. Consider this scenario: A data file has 100 records. Two programs access this data file, both opening it. Program 1 locks the file, adds a new record, then flushes and unlocks the file. Program 1 knows that there are now 101 records in the file. However, Program 2 is not aware of the changes that Program 1 made--it thinks that there are still 100 records in the file. This out-of-sync situation is easily remedied by having Program 2 reload the data header from the file on disk.

How does Program 2 know that it needs to reload the header? It doesn't. Instead BULLET uses a simple yet effective approach when dealing with such problems. Whenever your program locks a file, BULLET automatically reloads the header. Whenever you unlock a file, BULLET automatically flushes the header.

See: ReadKHXB, LockXB

FlushDHXB

Pack: HandlePack **Src:** FlushDHXBsrc **Func:** 15/Mid-level

Write the internal copy of the data header for the opened .DBF data file handle to disk. The actual write occurs only if the header has been changed.

This is to ensure that the data header on disk matches exactly the data header that is being maintained by BULLET. Also, this routine updates the operating system's directory entry for this file.

Assume the following: A data file with 100 records. Your program opens the data file and adds 1 record. Physically, there are 101 records on disk. However, the header image of the data file on disk still reads 100 records. This isn't a problem, BULLET uses its internal copy of the data header and the internal copy does read 101 records. BUT, if there were a system failure now, the disk image would not get updated. After the system restart, BULLET opens the file, reads the header and thinks that there are 100 records. You lost a record. Now, if after that add above, your program issued a FlushDHXB, the header on disk is refreshed with the internal copy, keeping the two in-sync. Also, the routine updates the DOS directory entry, keeping things neat there as well. Still, it doesn't come without cost: flushing will take additional time, therefore, you may elect to flush periodically, or whenever the system is idle.

See: ReadDHXB, FlushKHXB, LockXB

CopyDHXB

Pack: CopyPack **Src:** CopyDHXBsrc **Func:** 16/Mid-level

Copy the .DBF file structure of an open data file to another DOS file.

This routine makes it easy for you to duplicate the structure of an existing .DBF file without having to specify all the information needed by CreateDXB.

The resultant .DBF will be exactly like the source, including number of fields and field descriptions. It will contain 0 records.

See: ZapDHXB

ZapDHXB

Pack: HandlePack **Src:** ZapDHXBsrc **Func:** 17/Mid-level

Delete all records for a .DBF data file.

This routine is similar to CopyDHXB except for one major difference: ALL DATA RECORDS IN THE *SOURCE* FILE ARE PHYSICALLY DELETED, so be *careful*.

If you have a .DBF file with 100 records and use ZapDHXB on it, all 100 records will be physically deleted and the file truncated to 0 records. There is no return from this routine. All data is gone.

C A U T I O N

See: ZapKHXB

CreateKXB

Pack: CreateKeyPack **Src:** CreateKXBsrc **Func:** 20/Mid-level

Create a new BULLET key file. Before you can create a key file, you must first have opened (and have created if necessary) the BULLET .DBF data file that it is to index. (BULLET couples index and data files tightly.)

To create the key file, you need to provide the key expression, key flags, .DBF file link handle, and optionally, the code page ID, country code, and collate table.

Key Expression

The key expression is an ASCII character string composed of the elements that are to make up this index file's key. The key can be composed of any or all of the fields in the .DBF data record or sub-strings within any of those fields. Two functions are supported in evaluating a key expression. These are **SUBSTR()** and **UPPER()**. SUBSTR() extracts part of a string starting at a particular position for x number of characters. UPPER() converts all lower-case letters to their upper-case equivalent. Since BULLET supports NLS, UPPER() conversion is not required for proper sorting of mixed-case text strings.

All names used in the key expression must be a valid field name in the DBF data file. Some sample key expressions given that the .DBF has the following fields:

Fields...	**Valid key expressions**
FNAME C 25 0	"LNAME"
LNAME C 25 0	"LNAME+FNAME"
SSN C 9 0	"SUBSTR(LNAME,1,4)+SUBSTR(FNAME,1,1)+SUBSTR(SSN,6,4)
DEPT N 5 0	"UPPER(LNAME+FNAME)" (for non-NLS index files)
: :	"DEPT+SSN" (N- + C-type is valid for non-binary keys)

The key expression is given to Bullet in the form of a text string, pointed to by AP.keyexpptr, during CreateKXB.

Key Flags

The key expression is used in conjunction with the key flags to determine the type of key generated.

First, if your index file is to disallow duplicate keys, add 1 to KeyFlag.

If you have a key composed of a character field(s) or portions thereof, you specify a KeyFlag = 2. This instructs BULLET that the sort order is left-to-right (proper mixed-case sorting is available, see code page ID).

If you have a key composed of a numeric field(s) or portions thereof, you can also specify a KeyFlag = 2. This instructs BULLET to treat the numeric field as a regular character field for sorting. To ensure proper sorting, you must decimal-align the +numeric strings in the .DBF data field, i.e., right-justify the numeric strings (dBASE .DBF numeric strings are stored as ASCII strings). These non-binary numeric fields are just like character fields to BULLET.

In addition, if you have a key composed of a SINGLE numeric field (fld type N) and the field is an integer (NO DECIMAL POINT), you can specify a KeyFlag of 16 or 32. KeyFlag=16 is for a field known to be in word/integer range; KeyFlag=32 if the field is known to be in LongInt range. These KeyFlag values instruct BULLET to sort the key as a 16/32-bit BINARY value. It also stores the key as a 16- or 32-bit value (only 2 or 4 bytes) in the index, eventhough the data field is in ASCII (keyflag=16 or 32).

Although not dBASE compatible, you may use BINARY FIELDS in your data records. dBASE always has ASCII data in the data fields, even if the field is numeric. For example, an N type field of 8.2 is stored as an ASCII text string in the data record, say, a string like " 1100.55". If you want dBASE compatibility your field data must also be ASCII. However, if you can forgo this requirement, you can use binary values in the fields.

To do this you must specify a field type of "B" (actually, anything but a "N") and, IF IT IS TO BE USED AS A KEY FIELD, also set the 16- or 32-bit KeyFlag. Unique and signed may also be flagged. The field length for a "B" field type is 2 or 4. Make sure the key flags match (2 if cINTEGER, 4 if cLONG).

If you specify a binary key flag (for either N or B field types), you must also specify whether the field is to be treated as a signed or unsigned value. If values less than 0 are possible, add to KeyFlag the hex number 0x8000.

```
KeyFlag = cUNIQUE|cCHAR;              /* unique character key (NLS sort) */
KeyFlag = cINTEGER|cUNIQUE;           /* unique unsigned integer (binary sort) */
KeyFlag = cUNIQUE|cSIGNED|cLONG;      /* unique signed long */
KeyFlag = cCHAR;                      /* character key with duplicates allowed */
KeyFlag = cCHAR|cINTEGER;             /* THIS IS AN INVALID KEY FLAG! */
```

Click here to see the values defined in BULLET.H.

StatKXB is used to query KeyFlags.

The key expression you specify may be up to 136 characters, and evaluate out to 64 bytes (62 bytes if unique key is not specified). I.e, "SUBSTR(..." can be up to 136 characters, and that the actual key built from this expression can be no longer that 64 bytes, or 62 if you did not specify UNIQUE. In general, shorter keys (the key itself, not the expression) offer better performance.

DBF File Link Handle (XBlink)

Since BULLET evaluates the key expression at CreateKXB, it must have access to the DBF file to verify that the key expression is valid. You must therefore supply CreateKXB with the OS file handle of the opened DBF data file.

National Language Support (NLS)

With DOS 3.3 and later, NLS is available. BULLET uses NLS to build the collate sequence table that it uses to ensure proper sorting of mixed-case keys as well as the sorting of foreign language alphabets. In order for BULLET to use the proper collate table, it must know what code page ID and coutry code to use. This table is made part of the index file so that all subsequent access to the index file maintains the original sort order, even if the MIS shop is moved to another location/computer system using another country code/code page.

Code Page ID

To use the default code page ID of the computer in use, specify a code page ID of -1. This instructs BULLET to use the collate-sequence table as provided by MS-DOS running on the machine. You may also specify the code page ID for BULLET to use, but only if support for the code page ID is available on your machine. Look in your DOS manual under CUSTOMIZING FOR INTERNATIONAL USE for specific code page IDs and country codes. See also the COUNTRY and NLSFUNC commands. You may also specify a code page ID = 0 in which case no collate table is used.

Country Code

To use the default country code of the computer in use, specify a country code of -1. This instructs BULLET to use the collate-sequence table as provided by MS-DOS running on the machine. You may also specify the country code for BULLET to use, but only if support for the country code is available on your machine. Look in your DOS manual under CUSTOMIZING FOR INTERNATIONAL USE for specific code page IDs and country codes. See also the COUNTRY and NLSFUNC commands. You may also specify a country code = 0 in which case no collate table is used.

Typically, you set CodePageID = -1, CoutryCode = -1 and CollatePtr = 0.

User-specified Collate Table

If you are to use a MS-DOS supplied collate table (BOTH codepage ID and country codes are non-zero) then you do not need to specify a collate table--DOS will. The option to allow a user-specified collate table is to work around some DOS versions supplying incorrect collate tables. If you find that the DOS-supplied collate table is not valid (it's stored in the second sector of the file) for your country, you can supply the table to be used by pointing the CollatePtr variables to your in-memory version of a valid collate table. If you want to use the DOS-supplied collate table, you MUST set the CollatePtr variables = 0.

Note: The collate table is a 256-byte table that contains the sort value of each character (0-255). For example, the first byte would be 0, second would be 1, and so on. Values for characters up to the lower-case letters (ASCII 97) are usually as you would expect: "A" has a value of 65. However, the lower-case letters have the same value as their upper-case counterparts: "a" also has a value of 65. BULLET uses this collate table to ensure proper sorting.

If you specify EITHER code page ID OR country code = 0 then no collate table is used or built. Instead, sorting is done by standard ASCII sort. This is somewhat faster but less versatile. Use UPPER() for mixed-case sort if needed.

OpenKXB

Pack: OpenPack **Src:** OpenKXBsrc **Func:** 21/Mid-level

Open an existing key file for use.

Each key file that you open allocates 1264 bytes for internal use. This memory is not deallocated until you close the file with CloseKXB or execute ExitXB.

You must open the data file before you can open its related index file because you must supply the handle of the data file that this index files indexes.

See: OpenDXB

CloseKXB

Pack: HandlePack **Src:** CloseKXBsrc **Func:** 22/Mid-level

Close an open key file. Closing the file updates the file header and deallocates the memory used by this file.

You MUST close all BULLET files before ending your program or file corruption may occur. To ensure that all files are closed on the event of an unscheduled program termination, use AtExitXB.

See: CloseDXB

StatKXB

Pack: StatKeyPack **Src:** StatKXBsrc **Func:** 23/Mid-level

Get basic information on a BULLET key file handle specified. Information returned includes the number of keys in the file, the key length, the data file handle for this key, the last accessed record number of that data file, NLS information, and the key flags.

Typically, your program will keep track of whether a particular handle belongs to a key file or a data file. In cases where this is not possible, call the StatHandleXB routine to determine what file type a handle is.

See: StatDXB

ReadKHXB

Pack: HandlePack **Src:** ReadKHXBsrc **Func:** 24/Mid-level

Reload the disk copy of the key header for the opened key file handle to the internal copy.

In single-user, single-tasking systems this routine is not needed. However, in a multi-user or multi-tasking system it's possible, and desirable, for two or more programs to use the same data file. Consider this scenario: A key file has 100 keys. Two programs access this key file, both opening it. Program 1 locks the file, adds a new key, then flushes and unlocks the file. Program 1 knows that there are now 101 keys in the file. However, Program 2 is not aware of the changes that Program 1 made--it thinks that there are still 100 keys in the file. This out-of-sync situation is easily remedied by having Program 2 reload the key header from the file on disk.

How does Program 2 know that it needs to reload the header? It doesn't. Instead BULLET uses a simple yet effective approach when dealing with such problems. Whenever your program locks a file, BULLET automatically reloads the header. Whenever you unlock a file, BULLET automatically flushes the header.

See: FlushKHXB, ReadDHXB, LockXB

FlushKHXB

Pack: HandlePack **Src:** FlushKHXBsrc **Func:** 25/Mid-level

Write the internal copy of the key header for the opened key file handle to disk. The actual write occurs only if the header has been changed.

This is to ensure that the key header on disk matches exactly the key header that is being maintained by BULLET. Also, this routine updates the operating system's directory entry for this file.

Assume the following: A data file with 100 keys. Your program opens the key file and adds 1 key. Physically, there are 101 keys on disk. However, the header image of the data file on disk still reads 100 keys. This isn't a problem, BULLET uses its internal copy of the key header and the internal copy does read 101 keys. BUT, if there were a system failure now, the disk image would not get updated. After the system restart, BULLET opens the file, reads the header and thinks that there are 100 keys. You lost a key. Now, if after that add above, your program issued a FlushKHXB, the header on disk is refreshed with the internal copy, keeping the two in-sync. Also, the routine updates the DOS directory entry, keeping things neat there as well. Still, it doesn't come without cost: flushing will take additional time, therefore, you may elect to flush periodically, or whenever the system is idle.

See: ReadKHXB, LockXB

CopyKHXB

Pack: CopyPack **Src:** CopyKHXBsrc **Func:** 26/Mid-level

Copy the key file structure of an open key file to another DOS file.

This routine makes it easy for you to duplicate the structure of an existing key file without having to specify all the information needed by CreateKXB.

The resultant key file will be exactly like the source, including key flags and key expression. It will contain 0 keys.

ZapKHXB

Pack: HandlePack **Src:** ZapKHXBsrc **Func:** 27/Mid-level

Delete all keys for a key file.

This routine is similar to CopyKHXB except for one major difference: ALL KEYS IN THE *SOURCE* FILE ARE PHYSICALLY DELETED, so be *careful*.

If you have a key file with 100 keys and use ZapKHXB on it, all 100 keys will be physically deleted and the file truncated to 0 keys. There is no return from this routine. All data is gone.

C A U T I O N

See: CopyKHXB, ZapDHXB

GetDescriptorXB

Pack: DescriptorPack **Src:** GetDescriptorXBsrc **Func:** 30/Mid-level

Get the field descriptor information for a field.

You can specifiy either the fieldname or the field number (position of the field within the record where the first field is #1) to get info on.

The field descriptor contains the following information:

FIELDNAME 10 upper-case characters, A-Z and _ allowed, unused space is 0-filled and is 0-terminated (11 bytes, ASCII, byte 11 always=0)

FIELDTYPE single ASCII character where C=character, N=numeric, D=date, L=logical, and M=memo field (1 byte, ASCII)

FIELDLEN length of field: C=1-254, N=1-19, D=8 (yyyymmdd), L=1 (T/F/space), M=10, this is total field length (1 byte, binary)

FIELDDC places right of decimal point if N field type, minimum if not 0 is 2, can be up to 6 or 8, non-N fields always 0 (1 byte, binary)

GetRecordXB

Pack: AccessPack **Src:** GetRecordXBsrc **Func:** 31/Mid-level

Get the physical record from the data file into a data buffer by record number.

The data buffer is typically a struct variable defined as the DBF record itself is defined. For example, if the DBF record has 2 fields, LNAME and FNAME, then variable would be struct'ed as:

```
struct rectype {
        char   tag;          /* The Xbase DBF delete flag (must be included)   */
        char   lastname[25]; /* same field length as the .DBF LNAME field      */
        char   firstname[25]; /* same field length as the .DBF FNAME field      */
}; /* 51 */
struct rectype recbuff;
```

This method of accessing the data file does not use any indexing. Therefore, it typically is not used except for special purposes. The preferred method to access the data is by one of the keyed Get() routines.

See: GetEqualXB

AddRecordXB

Pack: AccessPack **Src:** AddRecordXBsrc **Func:** 32/Mid-level

Append the record in the data buffer to the end of the DBF file.

This method of adding a record does not involve any indexing. It is typically used to build a data file en masse and do the indexing after the entire .DBF file(s) has been built.

If you have several thousand data records to be added at once, this method of building the DBF first and then using the ReindexXB routine is often faster than using the InsertXB routine for each record to add.

The AddRecordXB is very fast. 400 recs/sec on an AT machine is typical. Over 2000 recs/sec can be added on a fast 486 machine--that's 120,000 records added per minute.

The record number used is determined by BULLET and it is returned in AP.RecNo.

See: UpdateRecordXB

UpdateRecordXB

Pack: AccessPack **Src:** UpdateRecordXBsrc **Func:** 33/Mid-level

Write the updated data record to the the physical record number.

This method of writing the updated record must not be used if any field(s) in the record is used as a key field(s) and has been changed.

This method of updating a record is very fast if you know that that update is not going to alter any field used as a key in any index file that uses it. You must, of course, first get the data record into the record buffer. Then you can change it and write the update out to disk with this routine.

If you need to change a field(s) that is used as a key field or part of one, use the UpdateXB routine. UpdateXB not only dynamically updates all related index files if you change a key field, it also will undo any and all changes if an error occurs in the transaction.

See: GetRecordXB, UpdateXB

DeleteRecordXB

Pack: AccessPack **Src:** DeleteRecordXBsrc **Func:** 34/Mid-level

Tag the record at the physical record number as being deleted.

This does not tag any in-memory copies of the record so be sure to mark any such copies as being deleted yourself.

The first byte of every .DBF record is reserved for the delete tag. This tag is a space (ASCII 32) if the record is normal, or a * (ASCII 42) if it's marked as being deleted. This delete tag is a reserved field in the DBF record and as such is not defined as a formal field with a descriptor, etc. Make sure that you define your in-memory buffers to reserve the first byte for the delete tag.

The Xbase DBF standard doesn't physically remove records marked as deleted from the data file. It doesn't mark them as available/reusable either. To physically remove records marked as deleted use PackRecordsXB.

Records can be temporarily marked as deleted then recalled to normal status. The Key/Get routines (GetFirstXB, etc.) return the record number needed for this routine after each access in AP.RecNo.

See: UndeleteRecordXB

UndeleteRecordXB

Pack: AccessPack **Src:** UndeleteRecordsrc **Func:** 35/Mid-level

Tag the record at the physical record number as being normal (not deleted).

This does not tag any in-memory copies of the record so be sure to mark any such copies as being normal yourself.

The first byte of every .DBF record is reserved for the delete tag. This tag is a space (ASCII 32) if the record is normal, or a * (ASCII 42) if it's marked as being deleted. This delete tag is a reserved field in the DBF record and as such is not defined as a formal field with a descriptor, etc. Make sure that you define your in-memory buffers to reserve the first byte for the delete tag.

The Xbase DBF standard does not physically remove records marked as deleted from the data file so you can "recall" then back to normal status as easily as you marked them deleted.

See: DeleteRecordXB

PackRecordsXB

Pack: AccessPack **Src:** PackRecordsXBsrc **Func:** 36/Mid-level

Rebuild the open DBF file by physically removing all records marked as deleted.

Packing occurs in place using the existing file. It's recommended that you use BackupFileXB to copy the current DBF file before using this routine in case of a failure during the pack process.

The newly packed file is truncated to reflect the current, actual size.

If there are index files for this .DBF file, they MUST all be reindexed after the pack process by using ReindexXB.

This routine dynamically allocates at least as many bytes as the length of the record. More if available.

See: DeleteRecordXB

FirstKeyXB

Pack: AccessPack **Src:** FirstKeyXBsrc **Func:** 40/Mid-level

Retrieve the first key in index order from the index file.

This routine does not access the .DBF file and so does not retrieve the data record. What it does do is locate the first key of the index, returning it, and also returning the record number within the .DBF that the key indexes.

To retrieve the data record you can use the GetRecordXB routine. The preferred method, however, is to use the GetFirstXB.

The key returned includes an enumerator if a non-unique index file is involved.

The enumerator is a little-endian 16-bit value that serves to differentiate up to 65535 "identical", non-unique keys. It is attached to all keys of non-unique index files and occupies the last two bytes of the key.

This routine is typically used to position the index file to the first key so as to allow forward in-order access to the keys by using NextKeyXB.

See: LastKeyXB

EqualKeyXB

Pack: AccessPack **Src:** EqualKeyXBsrc **Func:** 41/Mid-level

Search for the exact key in the index file.

This routine does not access the .DBF file and so does not retrieve the data record. What it does do is search for the key in the index, and if found, returns the record number within the .DBF that the key indexes. The key must be an exact match, including enumerator word if a non-unqiue index file.

To retrieve the data record you can use the GetRecordXB routine. The preferred method, however, is to use the GetEqualXB.

This routine will only find EXACT matches to the specified key (including the enumerator if applicable). However, even if the exact key is not found in the index file, the index file is positioned so that the next NextKeyXB retrieves the key that would have followed the unmatched specified key. For example, if the key to match was "KINGS" (a partial key in this case), EqualKeyXB would return a key not found error. If you were to now do a NextKeyXB, the next key would be returned, let's say it is "KINGSTON".

NextKeyXB

Pack: AccessPack **Src:** NextKeyXBsrc **Func:** 42/Mid-level

Retrieve the next key in index order from the index file.

This routine does not access the .DBF file and so does not retrieve the data record. What it does do is retreive the next key of the index, returning it, and also returning the record number within the .DBF that the key indexes.

To retrieve the data record you can use the GetRecordXB routine. The preferred method, however, is to use the GetNextXB.

The key returned includes an enumerator if a non-unique index file is involved.

This routine is typically called after the index file has first been positioned to a known key using either FirstKeyXB or EqualKeyXB, or after a previous NextKeyXB or even PrevKeyXB. What it basically does is get the key following the current key, and then make that key the new current key.

PrevKeyXB

Pack: AccessPack **Src:** PrevKeyXBsrc **Func:** 43/Mid-level

Retrieve the previous key in index order from the index file.

This routine does not access the .DBF file and so does not retrieve the data record. What it does do is retreive the previous key of the index, returning it and also returning the record number within the .DBF that the key indexes.

To retrieve the data record you can use the GetRecordXB routine. The preferred method, however, is to use the GetPrevXB.

The key returned includes an enumerator if a non-unique index file is involved.

This routine is typically called after the index file has first been positioned to a known key using either LastKeyXB or EqualKeyXB, or after a previous PrevKeyXB or even NextKeyXB. What it basically does is to get the key previous the current key, and then make that key the new current key.

LastKeyXB

Pack: AccessPack **Src:** LastKeyXBsrc **Func:** 44/Mid-level

Retrieve the last key in index order from the index file.

This routine does not access the .DBF file and so does not retrieve the data record. What it does do is locate the last key of the index, returning it, and also returning the record number within the .DBF that the key indexes.

To retrieve the data record you can use the GetRecordXB routine. The preferred method, however, is to use the GetLastXB.

This routine is typically used to position the index file to the last key so as to allow reverse in-order access to the keys by using PrevKeyXB.

StoreKeyXB

Pack: AccessPack **Src:** StoreKeyXBsrc **Func:** 45/Mid-level

Insert the key into the index file in proper key order.

This routine does not add the data record to the .DBF file. It only inserts the key and record number into the index file. Use InsertXB instead.

To do a complete data record and key insert, you could use AddRecordXB to add the data record to the .DBF, BuildKeyXB to construct the key, then StoreKeyXB to insert the key and record number information into the index file. If that key already exists and the file allows duplicate keys, you need to attach the proper enumerator word and retry StoreKeyXB.

This is much too much to do. Instead, just use InsertXB. All these details including adding the data record and multi-key inserts are performed automatically with just the single call.

DeleteKeyXB

Pack: AccessPack **Src:** DeleteKeyXBsrc **Func:** 46/Mid-level

Physically remove the specified key from the index file.

This routine requires an EXACT key match for all bytes of the key, including the enumerator word if a non-unique index file is involved.

This routine would seldom be used, typically, since deleted dBASE data records are only physically deleted during a PackRecordsXB and the index file is rebuilt afterward using ReindexXB.

See: CurrentKeyXB

BuildKeyXB

Pack: AccessPack **Src:** BuildKeyXBsrc **Func:** 47/Mid-level

Build the key for the specifed data record based on the key expression for the index file. If the index file is non-unique, a 0-value enumerator is attached.

The enumerator is a little-endian 16-bit value that serves to differentiate up to 65535 "identical", non-unique keys. It is attached to all keys of non-unique index files and occupies the last two bytes of the key.

This routine, like most of the mid-level routines, typically would not be used since the high-level access routines take care of this detail automatically.

See: StoreKeyXB

CurrentKeyXB

Pack: AccessPack **Src:** CurrentKeyXBsrc **Func:** 48/Mid-level

Retrieve the current key value for the specified key file handle and also the data record number that it indexes.

This routine is useful in that it retrieves on demand the actual key value of the last accessed key in the index file (and the data record number). Most often you don't need this information so it would be a waste of time and space for your program to explicitly track each current key for each index file that you have open.

See: ReindexXB, DeleteKeyXB

GetFirstXB

Pack: AccessPack **Src:** GetFirstXBsrc **Func:** 60/High-level

Retrieve the first indexed key and data record.

The key returned includes an enumerator if a non-unique index file is involved.

This routine is typically used to process a database in index order starting at the first ordered key (and its data record). After processing this first entry, subsequent in-order access of the database is achieved by using GetNextXB until the end of the database is reached.

This routine, like all the high-level Get routines, fills in the AP.RecNo of the record accessed. In GetFirstXB's case, it fills AP.RecNo with the record number pointed to by the first key. Since this is so, the AP pack is primed for an UpdateXB after each high-level Get. Other methods to get the record number are to use CurrentKeyXB or any of the Key routines (FirstKeyXB, etc.).

See: GetEqualXB, GetLastXB

GetEqualXB

Pack: AccessPack **Src:** GetEqualXBsrc **Func:** 61/High-level

Search for the exact key in the index file and return its data record.

This routine will only find EXACT matches to the specified key (including the enumerator if applicable). However, even if the exact key is not found in the index file, the index file is positioned so that the next GetNextXB retrieves the key that would have followed the unmatched specified key. For example, if the key to match was "KINGS" (a partial key in this case), GetEqualXB would return a key not found error. If you were to now do a GetNextXB, the next key and data record would be returned, let's say the key is "KINGSTON" and its data record is the data record for that key. Another GetNextXB would retrieve the key and record after that. (GetPrevXB can be used in this fashion too.)

This routine, like all the high-level Get routines, fills in the AP.RecNo of the record accessed. In GetEqualXB's case, it fills AP.RecNo with the record number pointed to by the matched key. Since this is so, the AP pack is primed for an UpdateXB after each high-level Get. Other methods to get the record number are to use CurrentKeyXB or any of the Key routines (EqualKeyXB, etc.).

GetNextXB

Pack: AccessPack **Src:** GetNextXBsrc **Func:** 62/High-level

Retrieve the next indexed key and its data record.

The key returned includes an enumerator if a non-unique index file is involved.

This routine is typically calld after the index file has first been positioned to a known key using either GetFirstXB or GetEqualXB, or after a previous GetNextXB or even GetPrevXB. What it basically does is get the key and data record following the current key, and then make that key the new current key.

This routine, like all the high-level Get routines, fills in the AP.RecNo of the record accessed. In GetNextXB's case, it fills AP.RecNo with the record number pointed to by the next key. Since this is so, the AP pack is primed for an UpdateXB after each high-level Get. Other methods to get the record number are to use CurrentKeyXB or any of the Key routines (NextKeyXB, etc.).

GetPrevXB

Pack: AccessPack **Src:** GetPrevXBsrc **Func:** 63/High-level

Retrieve the previous indexed key and record.

The key returned includes an enumerator if a non-unique index file is involved.

This routine is typically called after the index file has first been positioned to a known key using either GetLastXB or GetEqualXB, or after a previous GetPrevXB or even GetNextXB. What it basically does is to get the key and data record previous the current key, and then make that key the new current key.

This routine, like all the high-level Get routines, fills in the AP.RecNo of the record accessed. In GetPrevXB's case, it fills AP.RecNo with the record number pointed to by the previous key. Since this is so, the AP pack is primed for an UpdateXB after each high-level Get. Other methods to get the record number are to use CurrentKeyXB or any of the Key routines (PrevKeyXB, etc.).

GetLastXB

Pack: AccessPack **Src:** GetLastXBsrc **Func:** 64/High-level

Retrieve the last indexed key and record.

This routine is typically used to process a database in reverse index order starting at the last ordered key (and its data record). After processing this last entry, subsequent reverse-order access of the database is achieved by using GetPrevXB until the top of the database is reached.

This routine, like all the high-level Get routines, fills in the AP.RecNo of the record accessed. In GetLastXB's case, it fills AP.RecNo with the record number pointed to by the last key. Since this is so, the AP pack is primed for an UpdateXB after each high-level Get. Other methods to get the record number are to use CurrentKeyXB or any of the Key routines (LastKeyXB, etc.).

See: GetEqualXB, GetFirstXB

InsertXB

Pack: AccessPack **Src:** InsertXBsrc **Func:** 65/High-level

Add the data record to data file and insert the related key(s) into the linked index file(s).

This routine is used to add new entries into a database, one at a time. The data record is first added to the data file, then for each related index file, a key is inserted into the appropriate index file. Up to 32 index files can be automatically maintained for each data file.

This and several other routines are transaction-based. If a failure occurs prior to the routine's completion, all changes made to the database by the routine will be backed-out and the database (data and related index file(s)) effectively restored to its original state.

If the routine failed to complete, the function return value is the number of the pack that caused the failure. The pack's Stat is checked to determine the error code. If the function return value is 0, YOU MUST STILL check the first pack's Stat. If it's non-zero, then the failure occured with the data record.

See: UpdateXB, StoreKeyXB

UpdateXB

Pack: AccessPack **Src:** UpdateXBsrc **Func:** 66/High-level

Modify an existing data record (identified by record number) and automatically perform any index file updates needed to keep the index file(s) in sync.

If any key fields changed between the original record and the new one, this routine updates the appropriate index file(s) by replacing the original key(s) with new the key(s) based on the updated data record. Up to 32 index files can be automatically maintained for each data file. Get routines (GetFirstXB, etc.) set the AP.RecNo of the record that UpdateXB uses.

This and several other routines are transaction-based. If a failure occurs prior to the routine's completion, all changes made to the database by the routine will be backed-out and the database (data and related index file(s)) effectively restored to its original state.

If the routine failed to complete, the function return value is the number of the pack that caused the failure. The pack's Stat is checked to determine the error code. If the function return value is 0, YOU MUST STILL check the first pack's Stat. If it's non-zero, then the failure occured with the data record.

See: UpdateRecordXB

ReindexXB

Pack: AccessPack **Src:** ReindexXBsrc **Func:** 67/High-level

Reindex all related index files for a data file.

The index file(s) must already exist and be open. Any existing key data is overwritten by the new key data. In other words, if you have a 10MB index file, ReindexXB uses the same file space building the new keys over the old. This results in a less fragmented disk and also minimizes disk space needed. You can also create a new, empty index file and reindex to that. This would be useful, for instance, if you needed to create a temporary index file--something that you'd use for a report, say, then delete after the report.

This routine creates a TEMPORARY work file in either the current directory or, if the DOS environment variable TMP is defined, in the TMP= directory. The size of this file is approx. bytes = (RECORDS * (KEYLEN+6)). ReindexXB can operate in as little as 32K of available memory and can use up to 128K. The resultant index file(s) are optimized for minimum size AND maximum retrieval speed.

If the routine failed to complete, the function return value is the number of the pack that caused the failure. The pack's Stat is checked to determine the error code. A return value of zero indicates no error occured.

See: PackRecordsXB

LockXB

Pack: AccessPack **Src:** LockXBsrc **Func:** 80/Network

Lock all bytes in the index file handle(s) for exclusive use by the current process and reload the index file header(s) from disk. Also lock all bytes in the related data file and reload the data file header from disk.

The files must have been opened with the appropriate share attribute and not in compatibility mode. SHARE.EXE MUST be installed or DOS error 1 is issued.

This routine is transaction-based and will lock all index files specified in AccessPack and the data file. If any lock fails, all previous locks by this routine are released. The return value indicates which access pack failed, if any. This value is used as the index into the AccessPack group for you to identify the error code. See LockXBsrc for determining this exactly.

Use the DriveRemoteXB and/or FileRemoteXB to determine if locking is necessary. If the files are on a remote drive then it is best to use locking. Locking may also be necessary on multitasking local machines accessing shared files.

This routine is a combination of LockKeyXB and LockDataXB.

UnlockXB

Pack: AccessPack **Src:** UnlockXBsrc **Func:** 81/Network

Unlock all bytes in the specified file handle(s) (previously locked) and flush the file header(s) to disk (flush done before lock(s) released). Also unlock all bytes in the related data file and flush the data file header to disk.

The files must have been opened with the appropriate share attribute and not in compatibility mode. SHARE.EXE MUST be installed or DOS error 1 is issued.

This routine is transaction-based and will unlock all index files specified in AccessPack and the data file. If an unlock fails the routine exits with a return value indicating which access pack failed. This value is used as the index into the AccessPack group for you to identify the error code. Note that this routine does not attempt to re-lock those files unlocked successfully if an error occurs in the transaction. If an error does occur (unlikely) you will need to provide for unlocking the remaining files manually with the UnlockKeyXB and UnlockDataXB routines. You should not rely on the operating system to automatically unlock files when they're closed.

This routine is a combination of UnlockKeyXB and UnlockDataXB.

LockKeyXB

Pack: AccessPack **Src:** LockKeyXBsrc **Func:** 82/Network

Lock all bytes in the index file handle(s) for exclusive use by the current process and reload the index file header(s) from disk.

The files must have been opened with the appropriate share attribute and not in compatibility mode. SHARE.EXE MUST be installed or DOS error 1 is issued.

This routine is transaction-based and will lock all index files specified in AccessPack. If any lock fails, all previous locks by this routine are released. The return value indicates which access pack failed, if any. This value is used as the index into the AccessPack group for you to identify the error code.

The advantage of using region locks (LockKeyXB locks the entire file region) to control file access is that the file does not need to be opened/closed using the Deny Read/Write sharing attribute. Opening the file for Deny None, and controlling subsequent access with region locks, allows for faster processing since files do not need to be constantly opened and closed, as they would if access were controlled by opening with Deny Read/Write.

See: UnlockKeyXB, LockXB

UnlockKeyXB

Pack: AccessPack **Src:** UnlockKeyXBsrc **Func:** 83/Network

Unlock all bytes in the specified file handle(s) (previously locked) and flush the file header(s) to disk (flush done before lock(s) released).

The files must have been opened with the appropriate share attribute and not in compatibility mode. SHARE.EXE MUST be installed or DOS error 1 is issued.

This routine is transaction-based and will unlock all index files specified in AccessPack. If an unlock fails the routine exits with a return value indicating which access pack failed. This value is used as the index into the AccessPack group for you to identify the error code.

All file locks should be released when exclusive access in no longer needed.

It is not recommended that you end your program without having released active file locks. This is especially a valid concern for DOS versions prior to 5.0. DOS 5 releases locks on files that are closed.

See: LockKeyXB, LockDataXB, UnlockXB

LockDataXB

Pack: AccessPack **Src:** LockDataXBsrc **Func:** 84/Network

Lock all bytes in the file handle's data file for exclusive use by the current process and reload the data file header from disk. You must set AP.RecNo=0 to do this. To lock a single record, set AP.RecNo=record# to lock.

The files must have been opened with the appropriate share attribute and not in compatibility mode. SHARE.EXE MUST be installed or DOS error 1 is issued.

This routine locks the specified data file. If the handle specified is that of an index file, that index file's related data file handle is used. For single-record locks, AP.Handle must have a data file handle specified. Header loading is not performed if locking a single record.

The advantage of using region locks (LockDataXB locks the entire file region) to control file access is that the file does not need to be opened/closed using the Deny Read/Write sharing attribute. Opening the file for Deny None, and controlling subsequent access with region locks, allows for faster processing since files do not need to be constantly opened and closed, as they would if access were controlled by opening with Deny Read/Write.

See: UnlockDataXB, LockKeyXB, LockXB

UnlockDataXB

Pack: AccessPack **Src:** UnlockDataXBsrc **Func:** 85/Network

Unlock all bytes in the specified file handle (previously locked) and flush the data file header to disk (flush done before lock released). To do this you must set AP.RecNo=0. To unlock a single record, set AP.RecNo=record# to unlock.

The files must have been opened with the appropriate share attribute and not in compatibility mode. SHARE.EXE MUST be installed or DOS error 1 is issued.

This routine unlocks the specified data file. If the handle specified is that of an index file that index file's related datafile handle is used. For single-record unlocks, AP.Handle must have a data file handle specified. Flushing is not performed if unlocking a single record.

All file locks should be released when exclusive access in no longer needed.

It is not recommended that you end your program without having released active file locks. This is especially a valid concern for DOS versions prior to 5.0. DOS 5 releases locks on files that are closed.

See: LockDataXB, UnlockXB

DriveRemoteXB

Pack: RemotePack **Src:** DriveRemoteXBsrc **Func:** 86/Network

Determine if specified drive is remote (default drive=0, A:=1, B=2, C=3...).

This routine uses INT21/44/sub function 09.

In addition to returning the IsRemote state, this routine sends back the result of the DX register and also the install state of SHARE.EXE.

The meaning of the bitflags in Flags are (where IsRemote=0):

Bit	Meaning drive...
1	1=uses 32-bit sectoring
6	1=accepts Generic IOCTL (for INT21/44/0D,0E,0Fh)
7	1=accepts Query IOCTL Device (INT21/44/11h)
9	1=is local but shared by other computers in the network
11	1=accepts Does-Device-Use-Removable-Media (INT21/44/08)
13	1=requires media descriptor in FAT
14	1=accepts Receive/Send Control Data from Block Device (INT21/44/04,05)
15	1=is Substitution drive (set by the DOS SUBST command)
	(all other bits=0)

See: FileRemoteXB, LockXB

FileRemoteXB

Pack: RemotePack **Src:** FileRemoteXBsrc **Func:** 87/Network

Determine if specified handle of file or device is remote.

This routine uses INT21/44/sub function 0Ah.

In addition to returning the IsRemote state, this routine sends back the result of the DX register and also the install state of SHARE.EXE.

Flags bit 7=1 then handle is device, =0 then handle is file.

Bit	Meaning DEVICE...
0	1=is console input device
1	1=is console output device
2	1=is null device
3	1=is clock device
4	1=is special device
5	1=is in binary mode, 0=in ASCII
6	0=returns EOF if device is read
11	1=is network spooler
12	1=is NoInherit

Bit	Meaning DEVICE...(cont)
13	1=is named pipe
15	1=is remote, 0=is local
	(all other bits=0)
Bit	**Meaning FILE...**
0-5	xxxxxx=drive number (0=A...)
6	1=has not been written to
12	1=is NoInherit
14	1=date/time not set at close
15	1=is remote, 0=is local
	(all other bits=0)

See: DriveRemoteXB, LockXB

SetRetriesXB

Pack: SetRetriesPack **Src:** SetRetriesXBsrc **Func:** 88/Network

Set the number of times DOS retries disk operations after a failure due to file-sharing operations (locked file regions from LockXB routines).

This routine uses INT21/44/sub function 0Bh.

By default DOS retries an operation 3 times (without pausing between attempts) before returning an error to the application.

If you change the default values it's recommended that the default state be restored before your application ends (Retries=3, Pause=1).

These values are pretty much determined by trial-and-error. You may find that adding a delay between retries returns fewer access-denied errors.

See: LockXB

DeleteFileDOS

Pack: DOSFilePack **Src:** DeleteFileDOSsrc **Func:** 00/DOS

Delete the specified file.

This routine uses DOS INT21/41 (interrupt 21h function 41h).

RenameFileDOS

Pack: DOSFilePack **Src:** RenameFileDOSsrc **Func:** 101/DOS

Rename a file. May also be used to move the file to a new directory within the partition.

This routine uses DOS INT21/56.

If the specified directory differs from the file's directory, the file's directory entry is moved to the new directory.

For example, if the FilenamePtr filename is C:\LP100\PROJ94A.INF and the NewFilenamePtr filename is C:\ARCH\PROJ93A.INA, the file is essentially renamed and also moved to the \ARCH directory.

CreateFileDOS

Pack: DOSFilePack **Src:** CreateFileDOSsrc **Func:** 102/DOS

Create a new file.

This routine uses INT21/3C.

The specified filename/pathname must NOT already exist.

The file created is not left open. You must OpenFileDOS to use it.

The attribute used during the create can be:

Attribute	Value	Meaning
Normal	0	normal access permitted to file
Read-Only	1	read-only access permitted to file
Hidden	2	file does not appear in directory listing
System	4	file is a system file
Volume	8	FILENAME used as volume label if no current label
Archive	20h	file is marked for archiving

See: OpenFileDOS

AccessFileDOS

Pack: DOSFilePack **Src:** AccessFileDOSsrc **Func:** 103/DOS

Determine if the specified file can be accessed with the specified access/sharing mode.

This routine uses INT21/3D and INT21/3E.

Basically, a Does-File-Exist routine. It uses the specified access/sharing attributes when trying to open the file. For example, if you specify DFP.Attr = &H42 (R/W access + Deny None sharing) and use AccessFileDOS on a Read-Only DOS file, the return value would be DOS error 5, Access Denied.

OpenFileDOS

Pack: DOSFilePack **Src:** OpenFileDOSsrc **Func:** 104/DOS

Open the specified file with the specified access/sharing mode.

This routine uses INT21/3D.

Access	**Value**	**Meaning**
Read-only	0	open for read-only access
Write-only	1	open for write-only access
Read/Write	2	open for read/write access
Share		
Compatibility	0	any process may share file (not recommended)
Deny Read/Write	10h	no other process may share file
Deny Write	20h	no other process may share file for write
Deny Read	30h	no other process may share file for read
Deny None	40h	any process may share file except in Compatibilty mode
Inherit		
NoInheritFlag	80h	if set child processes do not inherit file handles (child process cannot inherit handle > 20)

The file access mode is a combination of ACCESS + SHARE + INHERIT.

See: OpenPack, CloseFileDOS

SeekFileDOS

Pack: DOSFilePack **Src:** SeekFileDOSsrc **Func:** 105/DOS

Position the DOS file pointer of the specified file to the specified position.

This routine uses INT21/42.

The position is a 32-bit value and is relative to either the start of the file, the current file pointer position, or the end of the file.

Method	Meaning
0	start move from the start of file (offset is a 32-bit unsigned value)
1	start move at the current position (offset a signed value)
2	start move at the end of file (offset a signed value)

For example, to move to the 511th byte of a file (byte 0 being the first), set the offset value to 511 and use Method 0. On return, the absolute offset value of the new position is returned. This is useful with Method 2 since you can specify an offset of 0 and have the file length returned.

Never position the file pointer to before the start of file.

ReadFileDOS

Pack: DOSFilePack **Src:** ReadFileDOSsrc **Func:** 106/DOS

Read from the file or device the specified number of bytes into a buffer.

This routine uses INT21/3F.

On block devices (such as disks) input starts at the current file position and the file pointer is repositioned to the last byte read +1.

It is possible to read less than the bytes specified without an error being generated. Compare the bytes to read with the returned bytes read value. If less then end of file was reached during the read, if 0 then file was at EOF.

By using DOS's predefined handles you can read from the keyboard (STDIN) by using the STDIN handle, 0. The input will terminate after all specified bytes have been read or after a CR (ASCII 0Dh). If more bytes are entered than were requested, the next read will retrieve those excess bytes. Therefore, it's suggested that you specify 129 bytes to input (DOS will process 127+CR/LF bytes maximum when reading the STDIN device). Post-process the entered data by scanning for the CR/LF.

See: WriteFileDOS

ExpandFileDOS

Pack: DOSFilePack **Src:** ExpandFileDOSsrc **Func:** 107/DOS

Expands the specified file by the specified number of bytes.

This routine uses INT21/42 and INT21/40.

This routine is useful in pre-allocating disk space. By reserving disk space in advance you can guarantee that enough disk space will be available for a future operation (especially if more than 1 process is running). You'll also be able ensure that the disk space that a file does use is as contiguous as possible.

Database systems are dynamic and their files typically allocate new space on an as-needed basis. This dynamic allocation can cause parts of a file to be located throughout the disk system, possibly affecting performance drastically. By pre-allocating the disk space you can be assured of consistent throughput performance since the file is contiguous.

WriteFileDOS

Pack: DOSFilePack **Src:** WriteFileDOSsrc **Func:** 108/DOS

Write to the file or device the specified number of bytes from a buffer.

This routine uses INT21/40.

If the number of bytes written is less than the specified bytes, this routine returns a -2 error code (or 65554 unsigned).

On block devices (such as disk) output starts at the current file position, and the file pointer is repositioned to the last byte written +1.

If the specified bytes to write is 0, the file is truncated at the current file pointer position.

By using DOS's predefined handles you can write to the screen (STDOUT) by'using the STDOUT handle, 1.

See: ReadFileDOS

CloseFileDOS

Pack: DOSFilePack **Src:** CloseFileDOSsrc **Func:** 109/DOS

Close the file flushing any internal buffers, releasing any locked regions, and update the directory entry to the correct size, date, and time.

This routine uses INT21/3E.

If you have opened a file using the DOS open routine you should close it when you no longer need it.

This routine can be used to close the predefined DOS handles (0-4) and make those handles available for reuse. Typically handles 0 and 1 should not be closed by an application since they are the STDIN and STDOUT that DOS uses for the current application (keyboard and screen).

Since BULLET provides for up to 250 user file handles for your applications it isn't necessary for you to eek 3 more file handles by closing handles 2-4.

See: OpenFileDOS

MakeDirDOS

Pack: DOSFilePack **Src:** MakeDirDOSsrc **Func:** 110/DOS

Create a new subdirectory.

This routine uses INT21/39.

AccessPack

Src: InsertXBsrc **Func:** InsertXB and many more

```
struct accesspack {
   unsigned        func;           /* varies                                  */
   unsigned        stat;           /* ret:completion status                   */
   unsigned        handle;         /* OS handle                               */
   long            recno;          /* in:rec# to get/delete/update if applicable
   */
                                   /* in:single rec# to lock or 0=lock all    */
                                   /* ret:record number of data record        */
   void far        *recptr;        /* far pointer to record storage buffer    */
   void far        *keyptr;        /* far pointer to search key buffer        */
   void far        *nextptr;       /* far pointer to next key access pack     */
                                   /* or 0:0 if end of link or if N/A         */
};
```

The NextPtr variables are only used by InsertXB, UpdateXB, ReindexXB, and the LockXB routines. NextPtr is used as a link to the next related access pack, if any. Not all entries are used by all routines. Generally, any routine that gets/puts user data to the database uses this pack.

BreakPack

Src: BreakXBsrc **Func:** BreakXB

```
struct breakpack {
   unsigned        func;           /* 4                                          */
   unsigned        stat;           /* ret:completion status                      */
   unsigned        mode;           /* =0 disable Ctrl-C/Ctrl-Brk, 1=restore      */
};
```

A simple pack.

CopyPack

Src: BackupFileXBsrc **Func:** BackupFileXB, CopyDHXB, CopyKHXB

```
struct copypack {
   unsigned        func;           /*5=BackupFileXB,16=CopyDHXB,26=CopyKHXB     */
   unsigned        stat;           /* ret:completion status                      */
   unsigned        handle;         /* handle of BULLET file                      */
   char far        *filenameptr;   /* far pointer to filenameZ                   */
};
```

CreateDataPack

Src: CreateDXBsrc **Func:** CreateDXB

```
struct createdatapack {
   unsigned       func;          /* 10                                    */
   unsigned       stat;          /* ret:completion status                 */
   char far       *filenameptr;  /* far pointer to filenameZ to create    */
   unsigned       nofields;      /* number of fields per record           */
   void far       *fieldlistptr; /* far pointer to field list             */
   unsigned       fileid;        /* file signature byte, usually=3        */
};
```

The FieldListPtr variables point to an array of struct fielddesctype. This array is dimensioned for as many fields as there are in the record and contains the field descriptors, one for each field.

See: FieldDescType

CreateKeyPack

Src: CreateKXBsrc **Func:** CreateKXB

```
struct createkeypack {
   unsigned        func;           /* 20                                        */
   unsigned        stat;           /* ret:completion status                     */
   char far        *filenameptr;   /* far pointer to filenameZ                  */
   char far        *keyexpptr;     /* far pointer to key expressionZ            */
   unsigned        xblink;         /* BULLET XB data handle this file indexes   */
   unsigned        keyflags;       /* bit 0=unique,1=char,4=int,5=lng,F=signed  */
   int             codepageid;     /* codepage for NLS, -1 use system default   */
   int             countrycode;    /* country code for NLS, -1 to use default   */
   char far        *collateptr;    /* far ptr to prg-supplied collate table     */
                                   /* or 0:0 if using sys-determined NLS table  */
};
```

Bit 14 in KeyFlags (0Eh) is set by BULLET during CreateKXB if a collate table is present.

Note: In the Windows version, if requested to use the system default collate table, BULLET makes a call to DPMI (INT31) function2 in order to obtain a valid selector for the pointer returned by DOS.

See: What is NLS

DescriptorPack

Src: GetDescriptorXBsrc **Func:** GetDescriptorXB

```
struct descriptorpack {
   unsigned        func;           /* 30                                    */
   unsigned        stat;           /* ret:completion status                 */
   unsigned        handle;         /* data file handle to get info on       */
   unsigned        fieldnumber;    /* field number to get info on; if 0...  */
   struct fielddesctype    fd;     /* ...search for DP.FD.FieldName         */
};
```

GetDescriptorXB allows you to get the field descriptor info for a particular field number (as in the first field, or the 10th field, etc.) or, if you don't know the physical field number, the routine can also get the info for a field by field name.

To get the info for field number, say 5, set DP.FieldNumber = 5. The DP.FD structure element is filled in with field 5's information.

To get the info for a field by fieldname, say LASTNAME, set dp.fieldnumber=0 & strcpy(dp.fd.fieldname, "LASTNAME") - the fieldname must be zero-filled and zero-terminated.

See: FieldDescType

DOSFilePack

Src: AccessFileDOSsrc **Func:** AccessFileDOS
(all routines ending with DOS)

```
struct dosfilepack {
   unsigned        func;           /* varies, see DeleteFileDOS for first      */
   unsigned        stat;           /* ret:completion status                    */
   char far        *filenameptr;   /* far pointer to filenameZ                 */
   unsigned        handle;         /* in: handle to access                     */
                                   /* ret: handle opened                       */
   unsigned        asmode;         /* open access/sharing mode                 */
   unsigned        bytes;          /* in: bytes to read  ret: bytes read       */
   long            seekoffset;     /* seek to file position                    */
   unsigned        method;         /* seek method                              */
   void far        *bufferptr;     /* far pointer to read/write buffer         */
   unsigned        attr;           /* file create directory entry attribute    */
   char far        *newnameptr;    /* pointer to new filenameZ for rename      */
};
```

All of the xDOS routines use this pack. Often only a few of the structure member elements are used by any one of the routines. Set only those needed.

DVmonPack

Src: DVmonCXBsrc **Func:** DVmonCXB

```
struct dvmonpack { /* AVAILABLE ONLY IN THE DEBUG ENGINE                    */
   unsigned        func;           /* 9                                    */
   unsigned        stat;           /* ret:completion status                */
   unsigned        mode;           /* =0 disable montitoring, =1 enable    */
   unsigned        handle;         /* file handle to monitor               */
   unsigned        vs;             /* segment for screen image (eg, 0xB800)*/
};
```

This routine is supplied only in the BULLET debug engine. It displays real-time monitoring information of a .DBF file or index and .DBF file pair including searches, seeks, hits, current record number, current key, key node contents, key node pointers, stack state, key and record counts, and other info.

ExitPack

Src: InitXBsrc **Func:** ExitXB, AtExitXB

```
struct exitpack {
   unsigned        func;           /* 1=ExitXB, 2=AtExitXB                    */
   unsigned        stat;           /* ret:completion status                   */
};
```

FieldDescType

Src: CreateDXBsrc **Func:** CreateDXB

```
struct fielddesctype {            /* used by CreateDataPack ONLY              */
   char            fieldname[11]; /* 0-filled (use only ASCII 65-90,95)       */
   char            fieldtype;     /* Char,Numeric,Date,Logical,Memo           */
   unsigned long   fieldda;       /* =0,reserved                              */
   unsigned char   fieldlen;      /* C=1-254,N=1-19(varies),D=8,L=1,M=10      */
   unsigned char   fielddc;       /* dec places for FieldType=N (0,2-15)      */
   long            fieldrez;      /* =0,reserved                              */
   char            filler[10];    /* =0,reserved                              */
};
```

If you can can forgo dBASE compatility you can use the B field type. This type is for fields that contain binary data (all dBASE fields contain ASCII text or numeric strings). If you specify a FieldType = "B" for, say an integer field, use a FieldLen = 2. If the field is a long integer, use FieldLen = 4. You can also use this non-standard field type for indexing. See CreateKXB for more.

See: CreateDataPack

HandlePack

Src: CloseDXBsrc

Func: CloseDXB, ReadDHXB, FlushDHXB, ZapDHXB
CloseKXB, ReadKHXB, FlushKHXB, ZapKHXB

```
struct handlepack {
   unsigned        func;          /* 12=CloseDXB,14=ReadDHXB,15=FlushDHXB      */
                                  /* 17=ZapDHXB, 22=CloseDXB,24=ReadKHXB       */
                                  /* 25=FlushKHXB,27=ZapKHXB                   */
   unsigned        stat;          /* ret:completion status                     */
   unsigned        handle;        /* handle of BULLET file                     */
};
```

InitPack

Src: InitXBsrc **Func:** InitXB

```
struct initpack {
   unsigned        func;           /* 0                                          */
   unsigned        stat;           /* ret:completion status                      */
   unsigned        jftmode;        /* expand JFT if non-zero                     */
   unsigned        dosver;         /* ret:DOS version                            */
   unsigned        version;        /* ret:BULLET version * 100                   */
   unsigned        osversion;      /* ret:0=real mode DOS; 1=prot mode Windows   */
   unsigned long   exitptr;        /* ret:far pointer to ExitXB routine          */
};
```

MemoryPack

Src: MemoryXBsrc **Func:** MemoryXB

```
struct memorypack {
   unsigned        func;           /* 3                                    */
   unsigned        stat;           /* ret:completion status                */
   unsigned long   memory;         /* ret:largest free OS memory block     */
};
```

OpenPack

Src: OpenDXBsrcCTX_OPENDXBSRC **Func:** OpenDXBCTX_FN_OPENDXB, OpenKXBCTX_FN_OPENKXB

```
struct openpack {
   unsigned        func;           /* 11=OpenDXB,21=OpenKXB                  */
   unsigned        stat;           /* ret:completion status                  */
   unsigned        handle;         /* ret:OS handle of file opened           */
   char far        *filenameptr;   /* far pointer to filenameZ to open       */
   unsigned        asmode;         /* sharing mode(see OpenFileDOS)          */
   unsigned        xblink;         /* if opening index,related data file     */
                                   /* (if opening data file, not used)       */
};
```

Note: you must supply xbHandle on index file opens

RemotePack

Src: DriveRemoteXBsrcCTX_DRIVEREMOTESRC **Func:** DriveRemoteXBCTX_FN_DRIVEREMOTEXB, FileRemoteXBCTX_FN_FILEREMOTEXB

```
struct remotepack {
   unsigned        func;           /* 86=DriveRemoteXB,87=FileRemoteXB          */
   unsigned        stat;           /* ret:completion status                     */
   unsigned        handle;         /* handle/drive depending on routine         */
   unsigned        isremote;       /* ret:0=local,1=remote                      */
   unsigned        flags;          /* ret:dx register returned by DOS           */
   unsigned        isshare;        /* ret:0=SHARE.EXE not loaded                */
};
```

SetRetriesPack

Src: SetRetriesXBsrc **Func:** SetRetriesXB

```
struct setretriespack {
   unsigned        func;           /* 88                                   */
   unsigned        stat;           /* ret:completion status                */
   unsigned        mode;           /* 0=set DOS default else Pauses        */
                                   /* Retries below                        */
   unsigned        pause;          /* 0-65535 loops between retries        */
   unsigned        retries;        /* 0-65535 retries to access file       */
};
```

The default values for Retries is 3 and Pause is 1.

The Pause value is used as a simple loop counter used to waste time. This loop IS dependent on CPU power so values are not portable across different machines.

Do not use unrealistic values. For example, don't set Retries to 30,000 unless you really want to wait for DOS to try 30,000 times before returning an error!

StatDataPack

Src: StatDXBsrc **Func:** StatDXB

```
struct statdatapack {
   unsigned        func;          /* 13                                    */
   unsigned        stat;          /* ret:completion status                 */
   unsigned        handle;        /* BULLET data file to get status on     */
   unsigned char   filetype;      /* ret:1=BULLET XB data file             */
   unsigned char   dirty;         /* ret:0=not changed                     */
   unsigned long   recs;          /* ret:records in file                   */
   unsigned        reclen;        /* ret:record length                     */
   unsigned        fields;        /* ret:fields per record ()              */
   char            f1;            /* reserved (1=update DVmon)             */
   unsigned char   luyear;        /* ret:binary, year file last updated    */
   unsigned char   lumonth;       /* ret:month                             */
                                  /* LUs are 0 if DBF newly created        */
   unsigned char   luday;         /* ret:day                               */
   unsigned        hereseg;       /* ret:this file's control segment       */
   char            filler[10];    /* reserved                              */
};
```

See: StatKeyPack

StatKeyPack

Src: StatKXBsrc **Func:** StatKXB

```
struct statkeypack {
   unsigned        func;           /* 23                                    */
   unsigned        stat;           /* ret:completion status                 */
   unsigned        handle;         /* BULLET key file to get status on      */
   unsigned char   filetype;       /* ret:0=BULLET XB key file              */
   unsigned char   dirty;          /* ret:0=not changed                     */
   unsigned long   keys;           /* ret:keys in file                      */
   unsigned        keylen;         /* ret:key length                        */
   unsigned        xblink;         /* ret:related BULLET XB data handle     */
   unsigned long   xbrecno;        /* ret:recno attached to current key     */
   unsigned        hereseg;        /* ret:this file's control segment       */
   unsigned        codepageid;     /* ret:codepage of key file sort         */
   unsigned        countrycode;    /* ret:countrycode of key file sort      */
   unsigned        collatetablesize; /* ret:size of collate table, 0 or 256 */
   unsigned        keyflags;       /* ret:bit 0=unique,1=char,4=int,        */
   char            filler[2];      /*     5=long,E=NLS,F=signed             */
};
```

See: StatDataPack

StatHandlePack

Src: StatHandleXBsrc **Func:** StatHandleXB

```
struct stathandlepack {
   unsigned        func;           /* 6                                          */
   unsigned        stat;           /* ret:completion status                      */
   unsigned        handle;         /* file handle to get information on          */
   unsigned        id;             /* ret:0=index,1=data,-1=not BULLET handle    */
   char far        *filenameptr;   /* pointer to filename of handle              */
};
```

XErrorPack

Src: GetExtErrorXBsrc **Func:** GetExtErrorXB

```
struct xerrorpack {
   unsigned        func;           /* 7                                  */
   unsigned        stat;           /* ret:extended error                 */
   unsigned        class;          /* ret:error class                    */
   unsigned        action;         /* ret:suggested action               */
   unsigned        location;       /* ret:error location                 */
};
```

See: DOS errors

BULLET errors

200 key not found	The search key for Equal was not matched exactly (including enumerator word attached to non-unique keys -- see docs). Next/Prev routines can be used to continue search from point of mismatch.
201 key already exists	Attempted to add a key that already exists in the index file created to allow only unique keys.
202 end of file	A Next routine is past the last key of the index file.
203 top of file	A Prev routine is before the first key of the index file.
204 key file empty	A key access was attempted with no keys in the index file.
205 key type unknown	Generally indicates a corrupt index header (key flags unknown at key insert).
	reserved,206-207
208 no more nodes	The index file has reached full capacity (32MB). ReindexXB can often shrink an index file by 30 to 50%.
209 key file corrupt	The index file is corrupt (write attempt to node 0).
210 key file corrupt	The index file is corrupt (internal overflow).
	reserved,211-219
220 incorrect DOS version	BULLET requires DOS 3.3 or later.
221 invalid key length	The key is > 62 bytes (or 64 if unique specified).
222 file not open	The specified handle is not an open BULLET file.
	reserved,223
224 invalid record number	The specified record number is < 0, past the last record number in the .DBF, or is > 16,777,215.
	reserved,225-227
228 invalid file type	The specified handle is not the correct type for the operation (i.e., specifying a data file handle for a key file operation).
	reserved,229-232
233 init not active	InitXB must be called before all others except MemoryXB.
234 init already active	InitXB has already been called. Use ExitXB first to call InitXB more than once per process. (Make sure the xxP.Func <> 0.)
235 too many indexes	BULLET can handle up to 32 index files per transaction record with the InsertXB and UpdateXB routines. Contact the author if you need to allow for more than 32 index files/transaction record.

236 null record pointer	The supplied pointer to the record buffer is null (e.g., AP.recptr = NULL;) and must not be for the operation to continue.
237 null key pointer	The supplied pointer to the key buffer is null (e.g., AP.keyptr = NULL;) and must not be for the operation to continue.
	reserved,238-239
240 invalid key expression	The CreateKXB key expression could not be evaluated.
	reserved,241
242 field not found	The fieldname was not found in the descriptor area.
243 invalid field count	Too many fields were specified or the specified field number is past the last field.
	reserved,244-249
250 invalid country info	The specifed country code or code page ID is not valid or not installed (according to DOS).
251 invalid collate table size	The specified country code/code page ID uses a collate-sequence table > 256 bytes (2-byte characters as with Kanji).
252 invalid key flags	The specified keyflags are invalid.
253 enhanced mode required for operation	CreateKeyXB requires Windows enhanced mode unless a programmer-supplied collate table pointer is provided (and appropriate code page/country code values -- i.e., non-zero).
	reserved,254
255 evaluation mode shutdown	BULLET evaluation period has completed. You can reinstall to continue evaluation, though you may want to consider your motives for reinstalling since the original evaluation period has expired. This error occurs only after the evaluation period has expired. It is not recommended that you continue to use BULLET after the evaluation period. It is possible for no 255 error to be generated for quite some time since it occurs only under certain load conditions and then only when certain routine sequences are performed. The specified evaluation period of 21 days should be adhered to.

See: DOS errors

DOS errors

-3 unexpected end of file
-2 disk full
-1 bad filename
0 no error
1 function not supported
2 file not found
3 path not found
4 too many open files
5 access denied (see Network specs)
6 handle invalid
7 MCBs destroyed
8 not enough memory
9 memory block address invalid
10 environment invalid
11 format invalid
12 access code invalid
13 data invalid
reserved-0Eh
15 disk drive invalid
16 cannot remove current directory
17 not same device
18 no more files
19 disk write protected
20 unknown unit
21 drive not ready
22 unknown command
23 data error (CRC)
24 bad request structure length
25 seek error
26 unknown medium type
27 sector not found
28 printer out of paper
29 write fault
30 read fault
31 general failure
32 sharing violation
33 lock violation
34 disk change invalid/wrong disk
35 FCB unavailable
36 sharing buffer overflow
37 code page mismatched
38 handle EOF
39 handle disk full
reserved-28h
reserved-29h
reserved-2Ah
reserved-2Bh
reserved-2Ch
reserved-2Dh
reserved-2Eh
reserved-2Fh
reserved-30h
reserved-31h
50 network request not supported

51 remote computer not listening
52 duplicate name on network
53 network pathname not found
54 network busy
55 network device no longer exists
56 NETBIOS command limit exceeded
57 network adapter hardware error
58 incorrect response from network
59 unexpected network error
60 incompatible remote adapter
61 print queue full
62 no spool space
63 not enough space to print file
64 network name deleted
65 network access denied
66 network device type incorrect
67 network name not found
68 network name limit exceeded
69 NETBIOS session limit exceeded
70 sharing temporarily paused
71 network request not accepted
72 print/disk redirection paused
reserved-49h
reserved-4Ah
reserved-4Bh
reserved-4Ch
reserved-4Dh
reserved-4Eh
reserved-4Fh
80 file exists
81 duplicate FCB
82 cannot make
83 fail on INT24
84 out of structures
85 already assigned
86 invalid password
87 invalid parameter
88 network write fault
reserved-59h
90 sys comp not loaded

DOS Class Codes

1 out of resources
2 temporary situation
3 authorization
4 internal error
5 hardware failure
6 system failure
7 application error
8 not found
9 bad format
10 locked
11 media failure
12 already exists
13 unknown

DOS Action Codes

1 retry immediately
2 delay and retry
3 reenter input
4 abort ASAP

DOS Locus Codes

1 unknown
2 block device
3 network
4 serial device

5	abort immediately	5	memory
6	ignore error		
7	user intervention		

See: BULLET errors

InitXBsrc

Func: InitXB **Pack:** InitPack **Func:** 0/System

```
struct initpack IP;
struct exitpack EP;

IP.func = INITXB;                /* InitXB defined in BULLET.H*/
IP.jftmode = 1;                  /* expand JFT to 255 handles */
stat = BULLET(&IP);
if (stat == 0) {
  EP.func = ATEXITXB;            /* register ExitXB with _atexit routine */
  stat = BULLET(&EP);
}
if (stat != 0) ...               /* error */
```

Note: When compiling for Windows, it is advisable to verify that the Windows version of BULLET is linked by testing IP.osversion (Fab thinks so).

ExitXBsrc

Func: ExitXB **Pack:** ExitPack **Func:** 1/System

struct exitpack EP;

```
EP.func = EXITXB                    /* ExitXB defined in BULLET.H*/
stat = BULLET(&EP)
```

The return value from ExitXB is currently always 0.

AtExitXBsrc

Func: AtExitXB **Pack:** ExitPack **Func:** 2/System

```
struct initpack IP;
struct exitpack EP;

IP.func = INITXB;                  /* InitXB defined in BULLET.H*/
IP.jftmode = 1;                    /* expand JFT to 255 handles */
stat = BULLET(&IP);
if (stat == 0) {
  EP.func = ATEXITXB;              /* register ExitXB with _atexit routine */
  stat = BULLET(&EP);
}
if (stat != 0) ...                 /* error */
```

MemoryXBsrc

Func: MemoryXB **Pack:** MemoryPack **Func:** 3/System

```
struct memoryPack MP;

MP.func = MEMORYXB;
stat = BULLET(&MP);

/* MP.memory = amount of far heap available */
```

MP.memory does not reflect memory available through DOS in the UMB area. It's possible that all memory requests can be satisfied by UMB RAM. Consult a DOS 5+ programmer reference for more information on this (see DOS INT21/58 for more).

BreakXBsrc

Func: BreakXB **Pack:** BreakPack **Func:** 4/System

```
struct breakpack BP;

BP.func = BREAKXB;                /* BreakXB defined in BULLET.H*/
BP.mode = 0;                      /* disable Ctrl-C/Ctrl-BREAK
                                     (do nothing on those keys) */
stat = BULLET(&BP);               /* stat=0 always */
```

If BreakXB is called multiple times with the same BP.mode each time, only the first is acted on. You can set BP.mode = 1 to restore the default handlers (those installed originally) and then again set BP.Mode = 0 to disable them.

ExitXB calls this routine automatically as part of the BULLET shutdown to restore the original default break handlers.

BackupFileXBsrc

Func: BackupFileXB **Pack:** CopyPack **Func:** 5/System

```
struct accesspack AP;
struct copypack CP;

CP.func = BACKUPFILEXB;           /* defined in BULLET.H*/
CP.handle = datahandle;           /* handle of data file to backup */
CP.filenameptr = newfilename;     /* filename to save to */
stat = BULLET(&CP);
if (stat == 0) {
  AP.func = PACKRECORDSXB;
  AP.handle = datahandle;
  stat = BULLET(&AP);
}
if (stat != 0) ...                /* error */
```

StatHandleXBsrc

Func: StatHandleXB **Pack:** StatHandlePack **Func:** 6/System

```
struct stathandlepack SHP;
struct statkeypack SKP;
struct statdatapack SDP;

SHP.func = STATHANDLEXB;          /* defined in BULLET.H*/
SHP.handle = thehandlenumber;
stat = BULLET(&SHP);
if (SHP.ID == 0) {                /* handle belongs to index file (index/key)*/
  SKP.func = STATKXB;             /* get key stats - see StatKXB/StatDXB for more*/
  SKP.handle = passedhandlenumber;     /* on the SKP structure*/
  stat = BULLET(&SKP);
}
elseif (SHP.ID == 1) {            /*.DBF data file */
                                  /* get DBF stats */
}
else
                                  /* error not a BULLET file type */
}
```

GetExtErrorXBsrc

Func: GetExtErrorXB **Pack:** XErrorPack **Func:** 7/System

```
/* an error just occured in the range 1 to 199 as returned in one of the
   pack.stat variables (current max DOS error is 90 (5Ah)) */

/* remember, transaction-based routines return a bad pack index in the
   return stat value, which you use to check the appropriate pack.Stat
   variable */

struct xerrorpack XEP;

XEP.func = GETEXTERRORXB;          /* defined in BULLET.H */
stat = BULLET(&XEP);
if (rstat != 0) {
                                   /* error=XEP.rstat
                                      error class=XEP.class
                                      recommened action=XEP.action
                                      location=XEP.location*/
}
```

DVmonCXBsrc

Func: DVmonCXB **Pack:** DVmonPack **Func:** 9/DEBUG

```
/* at this point a data file and a key file have been opened.
   kf is that key file's DOS handle
   */

struct dvmonpack DVP;

DV.func = DVMONCXB;               /* defined in BULLET.H*/
DV.mode = 1;                      /* enable monitoring */
DV.handle = kf;                   /* monitor key handle (and its XBlink file) */
DV.videoseg = 0xB800+(4096/16);   /* output to color screen, pg 1 (pg 0 to ?) */
stat = BULLET(&DV);               /* stat=0 always even if not DEBUG ENGINE */
```

For two-monitor systems (with a color monitor as the main system) output should be directed to 0xB000, the mono monitor's video memory.

DVmonCXB stands for Dual Video Monitor Control XB. This module is available on the BBS in the BULLET conference files. It is not included in the distribution, nor is a Windows version currently available, so this routine should not be used in Windows.

CreateDXBsrc

Func: CreateDXB **Pack:** CreateDataPack **Func:** 10/Mid-level

```
struct createdatapack CDP;
struct fielddesctype fieldlist[2];     /* fld descriptions for each field
                                          in the record (record has 2 fields) */

/* build FD first for each of the fields in the record */

memset(fieldlist,0,sizeof(fieldlist);  /* 0-fill is important */
strcpy(fieldlist[0].fieldname, "STUDENT");
fieldlist[0].fieldtype = 'C';
fieldlist[0].fieldlen = 20;
fieldlist[0].fielddc = 0;
strcpy(fieldlist[1].fieldname, "SCORE");
fieldlist[1].fieldtype = 'N';
fieldlist[1].fieldlen = 3;
fieldlist[1].fielddc = 0;
/* build the CDP */
CDP.func = CREATEDXB;              /* defined in BULLET.H */
CDP.filenameptr = filename;        /* filenameZ (Z=0-terminated str) */
CDP.nofields = 2;                  /* this example has 2 fields */
CDP.fieldlistptr = fieldlist       /* point to the first field decription...*/
CDP.fileid = 3;                    /* standard dBASE file ID */
stat = BULLET(&CDP);               /* create the DBF data file */
if (stat !=0) ...                  /* error */
```

Normally this code would be written as a generalized FUNCTION. The CDP could be a global allocation and the fieldlist would also.

BULLET can be customized very easily and with very little overhead. If you don't like the default API, just develop your own using your own parameter order, etc.

(for BINARY FieldType="B" see FieldDescType)

OpenDXBsrc

Func: OpenDXB **Pack:** OpenPack **Func:** 11/Mid-level

```
struct openpack OP;

OP.func = OPENDXB;                /* defined in BULLET.H */
OP.filenameptr = filename;        /* file to open (must already exist) */
OP.asmode = ReadWrite | DenyNone;     /* defined in BULLET.H*/
stat = BULLET(&OP);
if (stat !=0) ...                 /* error */
```

The ASmode (access/sharing mode) determines how the operating system controls access to the file. See OpenFileDOS for the meanings of the various ASmodes.

CloseDXBsrc

Func: CloseDXB **Pack:** HandlePack **Func:** 12/Mid-level

```
struct handlepack HP;

HP.func = CLOSEDXB;                /* defined in BULLET.H*/
HP.handle = datahandle;            /* handle of the file to close */
stat = BULLET(&HP);
if (stat !=0) ...                  /* error */
```

StatDXBsrc

Func: StatDXB **Pack:** StatDataPack **Func:** 13/Mid-level

```
struct statdatapack SDP;

SDP.func = STATDXB;                 /* defined in BULLET.H*/
SDP.handle = datahandle;            /* data handle to get stats on */
stat = BULLET(&SDP);                /* must be data handle, use StatHandleXB */
                                    /* if you don't know the type of file */
if (stat == 0) {                    /* SDP.filetype is set to 1
                                       SDP.dirty is set to 1 if the file has
                                       ...changed (0=not changed)
                                       SDP.recs = number records in DBF file
                                       SDP.recLen = record length
                                       SDP.fields = number fields in record
                                       SDP.f1 is reserved
                                       SDP.luyear = year file last updated,
                                       ...binary (year = ASC(SDP.LUyear))
                                       SDP.lumonth = month, binary
                                       SDP.luday = day, binary
                                       SDP.hereseg is set to this handle's
                                       control segment (location in memory)
   */
}
```

ReadDHXBsrc

Func: ReadDHXB **Pack:** HandlePack **Func:** 14/Mid-level

```
struct handlepack HP;

HP.func = READDHXB;                /* defined in BULLET.H*/
HP.handle = datahandle;            /* handle of file whose header to reload */
stat = BULLET(&HP);
if (stat != 0) ...                 /* error */
```

This routine is automatically called by the network lock routines.

FlushDHXBsrc

Func: FlushDHXB **Pack:** HandlePack **Func:** 15/Mid-level

```
struct handlepack HP;

HP.func = FLUSHDHXB;              /* defined in BULLET.H */
HP.handle = datahandle;           /* handle of file you want to flush */
stat = BULLET(&HP);
if (stat != 0) ...                /* error */
```

Note that the physical write to disk is performed only if the file has changed since the open or last flush.

This routine is automatically called by the network unlock routines.

See: UnlockDataXB

CopyDHXBsrc

Func: CopyDHXBsrc **Pack:** CopyPack **Func:** 16/Mid-level

```
struct copypack CP;

CP.func = COPYDHXB;                /* defined in BULLET.H */
CP.handle = datahandle;            /* handle of file to copy from (source) */
CP.filenameptr = filename;         /* pointer to filenameZ for copy (dest) */
stat = BULLET(&CP);
if (stat !=0) ...                  /* error */
```

ZapDHXBsrc

Func: ZapDHXB **Pack:** HandlePack **Func:** 17/Mid-level

```
struct handlepack HP;

HP.func = ZAPDHXB;                /* defined in BULLET.H */
HP.handle = datahandle;           /* handle of file you want to !ZAP! */
stat = BULLET(&HP);
if (stat !=0)...                  /* error */
```

Note that this removes ALL data records from the data file.

CreateKXBsrc

Func: CreateKXB **Pack:** CreateKeyPack **Func:** 20/Mid-level

```
/* This code assumes that the datafile was created as in CreateDXBsrc,
   and that the datafile was opened as in OpenDXBsrc.
   */

strcpy(keyexpression,"SUBSTR(STUDENT,1,5)");       /* a non-unique, char key*/
struct createkeypack CKP;

CKP.func = CREATEKXB;              /* defined in BULLET.H*/
CKP.filenameptr = filename;        /* far pointer to filenameZ */
CKP.keyexpptr = keyexpression;     /* far pointer to key expressionZ */
CKP.XBlink = datahandle;           /* the datafile handle returned from OpenDXB */
CKP.KeyFlags = cCHAR;              /* -KEYFLAGS- are defined in BULLET.H*/
CKP.CodePageID = -1;               /* use DOS default code page ID */
CKP.CountryCode = -1;              /* use DOS default country code */
CKP.CollatePtrOff = 0;             /* no user-supplied collate table...*/
CKP.CollatePtrSeg = 0;             /* ...*/
stat = BULLET(&CKP);
IF (stat !=0) ...                  /* error */
```

Normally this code would be written as a generalized FUNCTION.

OpenKXBsrc

Func: OpenKXB **Pack:** OpenPack **Func:** 21/Mid-level

```
struct openpack OP;

OP.func = OPENKXB;                /* defined in BULLET.H */
OP.filenamePtr = filename;        /* point to filenameZ (Z=0-terminated str) */
OP.asmode = ReadWrite | DenyNone;   /* defined in BULLET.H*/
OP.xblink = datafilehandle;       /* Needs to know the data file handle */
stat = BULLET(&OP);
if (stat !=0) ...                 /* error */
```

The ASmode (access/sharing mode) determines how the operating system controls access to the file. See OpenFileDOS for the meanings of the various ASmodes.

Before you can open an index file you must first open its associated data file.

See: OpenFileDOS

CloseKXBsrc

Func: CloseKXB **Pack:** HandlePack **Func:** 22/Mid-level

```
struct handlepack HP;

HP.func = CLOSEDXB;                /* defined in BULLET.H*/
HP.handle = indexhandle;           /* handle of the file to close */
stat = BULLET(&HP);
if (stat !=0) ...                  /* error */
```

StatKXBsrc

Func: StatKXB **Pack:** StatKeyPack **Func:** 23/Mid-level

```
struct statkeypack SKP;

SKP.func = STATKXB;                /* defined in BULLET.H*/
SKP.handle = indexhandle;          /* handle to get stats on */
stat = BULLET(&SKP);               /* must be index handle, use StatHandleXB
                                      if you don't know the type of file a
                                      handle's for */
if (stat == 0) {
   /* SKP.filetype is set to 0
      SKP.dirty is set to 1 if the file has changed (0=not changed)
      SKP.keys = number of key in the index file (index file=key file)
      SKP.keyLen = physical key length (1-64 bytes)
      SKP.xblink = datafile handle that this index file is associated with
      SKP.xbrecno is set to record number associated with last accessed key
      SKP.hereseg is set to this handle's control segment (location in memory)
      SKP.codepageid returns this index file's permanent code page ID
      SKP.countrycode returns this index file's permanent country code
      SKP.collatetablesize = 0 (no collate table present) or 256 (table present)
      SKP.keyflags= key flags specifed at CreateKXB (except NLS flag may be set)
      (NLS flag is bit 14, 0x4000) */
}
else
   /* error                        */
```

ReadKHXBsrc

Func: ReadKHXB **Pack:** HandlePack **Func:** 24/Mid-level

```
struct handlepack HP;

HP.func = READKHXB;                /* defined in BULLET.H*/
HP.handle = indexhandle;           /* handle of file whose header to reload */
stat = BULLET(&HP);
if (stat !=0) ...                  /* error */
```

This routine is automatically called by the network lock routines.

See: LockKeyXB

FlushKHXBsrc

Func: FlushKHXB **Pack:** HandlePack **Func:** 25/Mid-level

```
struct handlepack HP;

HP.func = FLUSHKHXB;              /* defined in BULLET.H*/
HP.handle = indexhandle;          /* handle of file you want to flush */
stat = BULLET(&HP);
if (stat !=0) ...                 /* error */
```

Note that the physical write to disk is performed only if the file has changed since the open or last flush.

This routine is automatically called by the network unlock routines.

See: UnlockKeyXB

CopyKHXBsrc

Func: CopyKHXB **Pack:** CopyPack **Func:** 26/Mid-level

```
struct copypack CP;

CP.func = COPYKHXB;                /* defined in BULLET.H*/
CP.handle = indexhandle;           /* handle of file to copy from (source) */
CP.filenameptr = filename;         /* far pointer to filenameZ for copy */
stat = BULLET(&CP);
if (stat !=0) ...                  /* error */
```

ZapKHXBsrc

Func: ZapKHXB **Pack:** HandlePack **Func:** 27/Mid-level

```
struct handlepack HP;

HP.func = ZAPKHXB;                /* defined in BULLET.H*/
HP.handle = indexhandle;          /* handle of file you want to !ZAP! */
stat = BULLET(&HP);
if (stat !=0) ...                 /* error */
```

Note that this removes ALL keys from the index file.

GetDescriptorXBsrc

Func: GetDescriptorXB **Pack:** DescriptorPack **Func:** 30/Mid-level

```
struct desciptorpack DP;

DP.func = GETDESCRIPTORXB;         /* defined in BULLET.H*/
DP.handle = datahandle;            /* handle of file
if (fieldnumber > 0) {
  DP.fieldnumber = fieldnumber;    /* field number to get info on */
}
else {                             /* or, if 0, then */
  DP.fieldnumber = 0;
  strcpy(DP.FD.fieldname,fieldname);      /* fieldname to get info on */
}
stat = BULLET(&DP);
if stat == 0 {
                                   /* DP.FD.fieldname is set to field name
                                      DP.FD.fieldtype is set to field type
                                      DP.FD.fieldlen is set to field length
                                      DP.FD.fielddc is set to field DC */
}
else
                                   /* error */
```

GetRecordXBsrc

Func: GetRecordXB **Pack:** AccessPack **Func:** 31/Mid-level

```
struct rectype {                    /* simple DBF record layout
   */
  char  tag;                        /* THE FIRST BYTE OF YOUR RECORD TYPES
                                       MUST BE TAG!  (a single byte) */
  char  code[5];
  char  bday[8];                    /* no \0 for DBF date field */
};
struct rectype recbuff;             /* 2 fields: code/C/5.0, bday/D/8.0 */
struct accesspack AP;

AP.func = GETRECORDXB;              /* defined in BULLET.H*/
AP.handle = datahandle;             /* handle to get record from */
AP.recno = recnotoget;              /* record number to get */
AP.recptr = recbuff;                /* read record from disk into recbuff */
stat = BULLET(&AP);
if (stat == 0) {
   /* recbuff.code and .bday set to contents of record (recnotoget) */
else
   /* error */
```

AddRecordXBsrc

Func: AddRecordXB **Pack:** AccessPack **Func:** 32/Mid-level

```
struct rectype {                    /* simple DBF record layout
   */
  char  tag;                        /* THE FIRST BYTE OF YOUR RECORD TYPES
                                       MUST BE TAG! (a single byte) */
  char  code[5];
  char  bday[8];
};
struct rectype recbuff;             /* 2 fields: code/C/5.0, bday/D/8.0 */
struct accesspack AP;

/* be sure to init the tag field to a space (ASCII 32) */
recbuff.tag = ' ';
strcpy(recbuff.code,"1234");
strcpy(recbuff.bday,"19331122");  /* won't be 0-term since must be 8 */

AP.func = ADDRECORDXB;              /* defined in BULLET.H*/
AP.handle = datahandle;             /* handle to put record to */
AP.recptr = recbuff;                /* write recbuff to disk */
stat = BULLET(&AP);
if (stat == 0) {
   /* AP.recno set to record number to used by just written record */
}
```

UpdateRecordXBsrc

Func: UpdateRecordXB **Pack:** AccessPack **Func:** 33/Mid-level

```
/* see GetRecordXBsrc for this source example's preliminary code
   */

AP.func = GETRECORDXB;            /* first get the record to update */
AP.handle = datahandle;
AP.recno = recnotoget;            /* Do NOT use UpdateRecordXB to change */
AP.recptr = recbuff;              /* any field(s) used in a key expression. */
stat = BULLET(&AP);               /* Instead use UpdateXB. */
if (stat==0) {
  strcpy(recbuff.dbay,"19591122");    /* change only non-key portions of record */
  AP.func = UPDATERECORDXB;       /* defined in BULLET.H*/
  stat = BULLET(&AP)              /* other AP. values are already set by Get */
}
if (stat !=0) ...                 /* error */
```

See: UpdateXB

DeleteRecordXBsrc

Func: DeleteRecordXB **Pack:** AccessPack **Func:** 34/Mid-level

```
struct accesspack AP;

AP.func = DELETERECORDXB;          /* defined in BULLET.H*/
AP.handle = datahandle;            /* handle of record to delete */
AP.recno = recnotodelete;          /* to determine which record number any record */
stat = BULLET(&AP);                /* is, use one of the keyed access routines */
if (stat !=0) ...                  /* error */
```

UndeleteRecordsrc

Func: UndeleteRecordXB **Pack:** AccessPack **Func:** 35/Mid-level

```
struct accesspack AP;

AP.func = UNDELETERECORDXB;        /* defined in BULLET.H */
AP.handle = datahandle;            /* handle of record to undelete */
AP.recno = recnotoundelete;        /* to determine which record number any record */
stat = BULLET(&AP);                /* is, use one of the keyed access routines */
if (stat !=0) ...                  /* error */
```

PackRecordsXBsrc

Func: PackRecordsXB **Pack:** AccessPack **Func:** 36/Mid-level

```
struct accesspack AP;

AP.func = PACKRECORDSXB;          /* defined in BULLET.H*/
AP.handle = datahandle;           /* handle of data file to pack */
stat = BULLET(&AP);
if (stat !=0) ...                 /* error */
```

FirstKeyXBsrc

Func: FirstKeyXB **Pack:** AccessPack **Func:** 40/Mid-level

```
struct accesspack AP;

AP.func = FIRSTKEYXB;              /* defined in BULLET.H*/
AP.handle = indexhandle;           /* handle to index file to access key from */
AP.keyptr = keybuffer;             /* far pointer to key buffer */
stat = BULLET(&AP);
if (stat==0) {
   /* keybuff is filled in with the key (for as many bytes as the key length)
      When using the key returned, be aware that if UNIQUE was NOT specified
      then the enumerator word is attached to the end of the key (the right
      two bytes).
      Also, AP.recno is set to record number of the first key in index. */
else
   /* error */
```

EqualKeyXBsrc

Func: EqualKeyXB **Pack:** AccessPack **Func:** 41/Mid-level

```
struct accesspack AP;

AP.func = EQUALKEYXB;              /* defined in BULLET.H */
AP.handle = indexhandle;           /* handle to index file to find key from */
AP.keyptr = keybuffer;             /* far pointer to key buffer (include enumerator
*/
stat = BULLET(&AP);                /* if non-UNQIUE key) (double NULL good start) */
if (stat==0) {
   /* the key matched exactly (including enumerator, if present)
      keybuff is NOT ALTERED
      AP.recno is set to the record number of that key */
else
  if (stat == 200) {
    AP.func = NEXTKEYXB            /* if not found, get following key and check the
*/
    stat = BULLET(&AP)             /* key proper (key less the enumerator bytes) */
    if (stat==0) {                 /* (i.e., it may be proper key but enumerator=1)
*/

/* See NextKeyXBsrc for continuation, since this routine find only the exact match,
including enumerator if non-unique, and so if not found, NextKeyXB will locate the
key item following the 'not found' search. */
```

NextKeyXBsrc

Func: NextKeyXB **Pack:** AccessPack **Func:** 42/Mid-level

```
/* see EqualKeyXBsrc for preliminary code */

AP.func = NEXTKEYXB;              /* defined in BULLET.H*/
rstat = BULLET(&AP);              /* KEYLEN assumed to equal actual key length
   */
if (stat==0) {                    /* ...as returned by StatKXB */
  if (IndexFileIsNotUnique) {
    if (KeyProperMatchesKeyToFind) {
      /* found a match to the key proper
```

This code example follows up on the EqualKeyXBsrc example. See EqualKeyXB for more information on finding partial keys.

PrevKeyXBsrc

Func: PrevKeyXB **Pack:** AccessPack **Func:** 43/Mid-level

```
struct accesspack AP;

/* assume code has already executed to locate a key--this code then gets
   the key before that one */

AP.func = PREVKEYXB;              /* defined in BULLET.H*/
AP.handle = indexhandle;          /* handle to index file to access key from */
AP.keyptr = keybuff;              /* far pointer to key buffer */
stat = BULLET(&AP);
if (stat==0) {
   /* keybuff is filled in with the key (for as many bytes as the key length).
      Also, AP.recno is set to the record number of the key. */
}
else
   /* error */
```

LastKeyXBsrc

Func: LastKeyXB **Pack:** AccessPack **Func:** 44/Mid-level

```
struct accesspack AP;

AP.func = LASTKEYXB;              /* defined in BULLET.H*/
AP.handle = indexhandle;          /* handle to index file to access key from */
AP.keyptr = keybuff;              /* far pointer to key buffer */
stat = BULLET(&AP);
if (stat==0) {
   /* keybuff is filled in with the key (for as many bytes as the key length).
      Also, AP.recno is set to the record number of the last key. */
else
   /* error */
```

StoreKeyXBsrc

Func: StoreKeyXB **Pack:** AccessPack **Func:** 45/Mid-level

```
struct accesspack AP;

/* Assume record has been added to data file (AddRecordXB, returning
   recno2use) and key has been built (BuildKeyXB returning keytoadd[]). */

AP.func = STOREKEYXB;              /* defined in BULLET.H*/
AP.handle = indexhandle;           /* handle to index file to insert key into */
AP.recno = recno2use;              /* associate this record number with key */
AP.keyptr = keytoadd;              /* far pointer to key buffer */
stat = BULLET(&AP);
if (stat==0)
  /* key added */
else
  if (rstat==201) {
    /* key already exists, you need to construct a unique enumerator -
       provided file wasn't created for UNIQUE keys...INSTEAD USE InsertXB! */
  }
  else
    /* other error */
```

DeleteKeyXBsrc

Func: DeleteKeyXB **Pack:** AccessPack **Func:** 46/Mid-level

```
struct accesspack AP;

AP.func = DELETEKEYXB;            /* defined in BULLET.H */
AP.handle = indexhandle;          /* handle to index file of key to delete */
AP.keyptr = keybuffer;            /* far pointer to key buffer (incl enum) */
                                  /* (this key is searched for exactly) */
stat = BULLET(&AP);               /* if exact match found the key is deleted! */
if (stat==0)
   /* key deleted permanently */
else
  if (rstat==200) {
    /* key as stated was not in the index file--if the index is not UNQIUE
       then you must supply the exact enumerator along with the key proper
       to delete - you can use the CurrentKeyXB routine to obtain the exact
       current key */
  }
else
  /* other error */
```

BuildKeyXBsrc

Func: BuildKeyXB **Pack:** AccessPack **Func:** 47/Mid-level

```
struct accesspack AP;

/* Assume record has been built and is ready to be added to the data file.
   The record is in the variable recbuff. */

AP.func = BUILDKEYXB;              /* defined in BULLET.H*/
AP.handle = indexhandle;           /* handle to index file key is to be built for
   */
AP.recptr = recbuff;               /* far pointer to data record buffer */
AP.keyptr = keybuff;               /* far pointer to key buffer */
stat = BULLET(&AP)
if (stat==0) {
  /* key built okay so can do a AddRecordXB followed by a StoreKeyXB
     but, again, InsertXB takes care of all this detail and then some-use it */
}
else
  /* error */
```

CurrentKeyXBsrc

Func: CurrentKeyXB **Pack:** AccessPack **Func:** 48/Mid-level

```
struct accesspack AP;

AP.func = CURRENTKEYXB;           /* defined in BULLET.H*/
AP.handle = indexhandle;          /* handle to index file */
AP.keyptr = keybuff;              /* far pointer to key buffer */
stat = BULLET(&AP);
if (stat==0) {
   /* keybuff set to current key (valid only for KeyLen bytes)
      Also, AP.recno is set to the record number of the key. */
}
else
   /* error */
```

GetFirstXBsrc

Func: GetFirstXB **Pack:** AccessPack **Func:** 60/High-level

```
struct accesspack AP;

AP.func = GETFIRSTXB;             /* defined in BULLET.H*/
AP.handle = indexhandle;          /* handle to index file to access key from */
AP.recptr = recbuff;              /* far pointer to record buffer */
AP.keyptr = keybuff;              /* far pointer to key buffer */
stat = BULLET(&AP);
if (stat==0) {
  /* keybuff is filled in with the key (for as many bytes as the key length)
     recbuff is filled in with the data record
     AP.recno is set to the record number of the first key in the index. */
}
else
  /* error */
```

GetEqualXBsrc

Func: GetEqualXB **Pack:** AccessPack **Func:** 61/High-level

```
struct accesspack AP;

AP.func = GETEQUALXB;              /* defined in BULLET.H*/
AP.handle = indexhandle;           /* handle to index file to access key from */
AP.recptr = recbuff;               /* far pointer to record buffer */
AP.keyptr = keybuff;               /* far pointer to key buffer */
stat = BULLET(&AP);
if (stat==0)
  /* recbuff and AP.recno filled as expected (keybuff remains the same) */
else
  if (rstat==200) {
    AP.func = GETNEXTXB;  /* if not found, can get following key--the next */
    stat = BULLET(&AP);  /* key would logically follow the key not found */
  }
  else
    /* error */
```

GetNextXBsrc

Func: GetNextXB **Pack:** AccessPack **Func:** 62/High-level

```
struct accesspack AP;

AP.func = GETNEXTXB;              /* defined in BULLET.H */
AP.handle = indexhandle;          /* handle to index file to access key from */
AP.recptr = recbuff;              /* far pointer to record buffer */
AP.keyptr = keybuff;              /* far pointer to key buffer */
stat = BULLET(&AP);
if (stat==0) {
  /* keybuff is filled in with the next key (as many bytes as the key length)
     recbuff is filled in with the data record
     AP.recno is set to the record number of the next key in the index
     This key is made the current key so GETNEXTXB again gets next, and so on */
}
else
  /* error */
```

GetPrevXBsrc

Func: GetPrevXB **Pack:** AccessPack **Func:** 3/High-level

```
struct accesspack AP;

AP.func = GETPREVXB;              /* defined in BULLET.H */
AP.handle = indexhandle;          /* handle to index file to access key from */
AP.recptr = recbuff;              /* far pointer to record buffer */
AP.keyptr = keybuff;              /* far pointer to key buffer */
stat = BULLET(&AP);
if (stat==0) {
  /* keybuff is filled in with the prev key (as many bytes as the key length)
     recbuff is filled in with the data record
     AP.recno is set to the record number of the prev key in the index
     This key is made the current key so GETPREVXB again gets prev, and so on */
}
else
  /* error */
```

GetLastXBsrc

Func: GetLastXB **Pack:** AccessPack **Func:** 64/High-level

```
struct accesspack AP;

AP.func = GETLASTXB;              /* defined in BULLET.H*/
AP.handle = indexhandle;          /* handle to index file to access key from */
AP.recptr = recbuff;              /* far pointer to record buffer */
AP.keyptr = keybuff;              /* far pointer to key buffer */
stat = BULLET(&AP);
if (stat==0) {
  /* keybuff is filled in with the key (for as many bytes as the key length)
     recbuff is filled in with the data record
     AP.recno is set to the record number of the last key in the index. */
else
  /* error */
```

InsertXBsrc

Func: InsertXB **Pack:** AccessPack **Func:** 65/High-level

```
struct accesspack AP[3];          /* array of 3 access packs, 1 for each index
                                     file; keybuff / recbuff previously defined */
for (i=0,i<3,i++) {               /* 3=number of related indexes to maintain */
  AP[i].func = INSERTXB;
  AP[i].handle = indexhandle[i];  /* each index file's handle */
  AP[i].recptr = recbuff;
  AP[i].keyptr = keybuff;
  AP[i].nextptr = &AP[i+1];       /* point to NEXT access pack */
}
AP[2].nextptr = NULL;             /* reset last access pack to end-link value */
stat = BULLET(&AP);
if (stat==0) {                    /* if stat=0 must still check AP[0].stat */
  if (AP[0].stat != 0) {          /* error when adding data record */
    /* error */
  }
}
else
  trueerror = AP[stat-1].stat;    /* returned rstat is array index of bad pack
                                     -1 since C packs are 0 based. */
```

UpdateXBsrc

Func: UpdateXB **Pack:** AccessPack **Func:** 66/High-level

```
struct accesspack AP[3];          /* array of 3 access packs, 1 for each index
                                     file; keybuff / recbuff previously defined */
for (i=0,i<3,i++) {               /* 3=number of related indexes to maintain */
  AP[i].func = UPDATEXB;
  AP[i].handle = indexhandle[i];  /* each index file's handle */
  AP[i].recno = recordnumbertoupdate;    /* the record number to update */
  AP[i].recptr = recbuff;
  AP[i].keyptr = keybuff;
  AP[i].nextptr = &AP[i+1];       /* point to NEXT access pack */
}
AP[2].nextptr = NULL;             /* reset last access pack to end-link value */
stat = BULLET(&AP);
if (stat==0) {                    /* if stat=0 must still check AP[0].stat */
  if (AP[0].stat != 0) {          /* error when adding data record */
    /* error */
  }
}
else
  trueerror = AP[stat-1].stat;    /* returned rstat is array index of bad pack
                                     -1 since C packs are 0 based. */
```

ReindexXBsrc

Func: ReindexXB **Pack:** AccessPack **Func:** 67/High-level

```
struct accesspack AP[3];          /* array of 3 access packs, 1 for each index
                                     file; keybuff / recbuff previously defined */
for (i=0,i<3,i++) {               /* 3=number of related indexes to maintain */
  AP[i].func = REINDEXB;
  AP[i].handle = indexhandle[i];    /* each index file's handle */
  AP[i].nextptr = &AP[i+1];       /* point to NEXT access pack */
}
AP[2].nextptr = NULL;             /* reset last access pack to end-link value
   */
stat = BULLET(&AP);
if (stat !=0) {
  trueerror = AP[stat-1].stat;    /* returned rstat is array index of bad pack
                                  /* -1 since C packs are 0 based. */
}
```

LockXBsrc

Func: LockXB **Pack:** AccessPack **Func:** 80/Network

```
struct accesspack AP[3];          /* array of 3 access packs, 1 per index file */

for (i=0,i<3,i++) {               /* 3=number of related indexes to maintain */
  AP[i].func = LOCKXB;
  AP[i].handle = indexhandle[i];    /* each index file's handle */
  AP[i].nextptr = &AP[i+1];       /* point to NEXT access pack */
}
AP[2].nextptr = NULL;             /* reset last access pack to end-link value */
stat = BULLET(&AP);
if (stat > 3) {                   /* if stat > 3 (> number of packs) then the... */
  trueerror = AP[2].stat;         /* ...lock failed on the data file */
}
else                              /* and the error code is in the last pack */
  if (rstat !=0) {
    trueerror = AP[rstat-1].stat    /* ...lock failed on index file # stat
                                       -1 since C packs are 0 based. */
  }
```

The Lock routines use a different method to identify the bad pack when the failure was caused by the data file. See above.

See: UnlockXBsrc

UnlockXBsrc

Func: UnlockXB **Pack:** AccessPack **Func:** 81/Network

```
struct accesspack AP[3];          /* array of 3 access packs, 1 per index file */

for (i=0,i<3,i++) {               /* 3=number of related indexes to maintain */
  AP[i].func = UNLOCKXB;
  AP[i].handle = indexhandle[i];    /* each index file's handle */
  AP[i].nextptr = &AP[i+1];       /* point to NEXT access pack */
}
AP[2].nextptr = NULL;             /* reset last access pack to end-link value */
stat = BULLET(&AP);
if (stat > 3) {                   /* if stat > 3 (> number of packs) then the...*/
   trueerror = AP[2].stat;        /* ...unlock failed on the data file */
}
else                              /* and the error code is in the last pack */
  if (rstat !=0) {
    trueerror = AP[stat-1].stat     /* ...unlock failed on index file # stat
                                       -1 since C packs are 0 based. */
  }
```

The Lock routines use a different method to identify the bad pack when the failure was caused by the data file. See above.

See: LockXBsrc

LockKeyXBsrc

Func: LockKeyXB **Pack:** AccessPack **Func:** 82/Network

```
struct accesspack AP[3];          /* array of 3 access packs, 1 per index file */
for (i=0,i<3,i++) {               /* 3=number of related indexes to maintain */
  AP[i].func = LOCKKEYXB;
  AP[i].handle = indexhandle[i];  /* each index file's handle */
  AP[i].nextptr = &AP[i+1];       /* point to NEXT access pack */
}
AP[2].nextptr = NULL;             /* reset last access pack to end-link value */
stat = BULLET(&AP);
if (stat !=0) {
  trueerror = AP[stat-1].stat     /* ...lock failed on index file # stat
                                     -1 since C packs are 0 based. */
                                  /* if stat > packs (3) then failed on last
                                     internal ReadKHXB. This is EXTREMELY unlikely.
*/
}
```

See: UnlockKeyXBsrc

UnlockKeyXBsrc

Func: UnlockKeyXB **Pack:** AccessPack **Func:** 83/Network

```
struct accesspack AP[3];          /* array of 3 access packs, 1 per index file
                                     keybuff and recbuff previously defined */
for (i=0,i<3,i++) {               /* 3=number of related indexes to maintain */
  AP[i].func = UNLOCKKEYXB;
  AP[i].handle = indexhandle[i];    /* each index file's handle */
  AP[i].nextptr = &AP[i+1];       /* point to NEXT access pack */
}
AP[2].nextptr = NULL:             /* reset last access pack to end-link value */
stat = BULLET(&AP);
if (stat !=0) {
  trueerror = AP[stat-1].stat;    /* ...lock failed on index file # stat
                                     -1 since C packs are 0 based.
}
```

See: LockKeyXBsrc

LockDataXBsrc

Func: LockDataXB **Pack:** AccessPack **Func:** 84/Network

```
struct accesspack AP;

AP.func = LOCKDATAXB;              /* defined in BULLET.H */
AP.handle = datahandle;            /* handle of data file to lock */
AP.recno = 0L;                     /* =0 to lock all or, set to actual record */
stat = BULLET(&AP);                /* num to lock, as in AP.recno = lockthisrec */
if (stat !=0) ...                  /* error */
```

See: UnlockDataXBsrc

UnlockDataXBsrc

Func: UnlockDataXB **Pack:** AccessPack **Func:** 85/Network

```
struct accesspack AP;

AP.func = UNLOCKDATAXB;          /* defined in BULLET.H*/
AP.handle = datahandle;          /* handle of data file to unlock */
AP.recno = 0L;                   /* =0 to unlock all or, set to actual record */
stat = BULLET(&AP);              /* num to unlock as in AP.recno = lockthisrec */
if (stat !=0) ...                /* error */
```

Note: you cannot unlock parts of a file with 1 single unlock (where AP.recno=0). Instead, you must unlock each record individually - that is, if you made any single-record locks

See: LockDataXBsrc

DriveRemoteXBsrc

Func: DriveRemoteXB **Pack:** RemotePack **Func:** 86/Network

```
struct remotepack RP;

RP.func = DRIVEREMOTEXB;          /* defined in BULLET.H */
RP.handle = drive2check;          /* drive (0=default, 1=A:,2=B:,3=C:...) */
stat = BULLET(&RP);
if (stat==0) {
  /* RP.isremote set to 0 if drive local, 1 if remote
     RP.flags set to DX register as returned by DOS
     RP.isshare set to 0 if SHARE.EXE is not loaded, non-zero SHARE installed */
}
else
  /* error (like invalid drive) */
```

See: FileRemoteXBsrc

FileRemoteXBsrc

Func: FileRemoteXB **Pack:** RemotePack **Func:** 87/Network

```
struct remotepack RP;

RP.func = FILEREMOTEXB;           /* defined in BULLET.H */
RP.handle = filehandle;           /* file handle to check */
stat = BULLET(&RP);
if (stat==0) {
  /* RP.isremote set to 0 if file local, 1 if remote
     RP.flags set to DX register as returned by DOS
     RP.isshare set to 0 if SHARE.EXE is not loaded, non-zero SHARE installed */
}
else
  /* error  (like invalid handle) */
```

See: DriveRemoteXBsrc

SetRetriesXBsrc

Func: SetRetriesXB **Pack:** SetRetriesPack **Func:** 88/Network

```
struct setretriespack SRP;

SRP.func = SETRETRIESXB;
SRP.mode = 1;                       /* 1=set to user values, 0=set DOS default */
SRP.pause = 5000;                   /* do 5,000 loops between retries */
SRP.retries = 5;                    /* try 5 times before giving up with error */
stat = BULLET(&SRP);
if (stat !=0) ...                   /* error */
```

Note that this routine has not been useful since the advent of fast computers: the delayed used in current MS-DOS versions simply loops in place and is completely dependent on CPU speed. For anything beyond an XT, the delay generated is too fast to be useful. Use your own delay if needed. Use time-out semaphores if you have them.

DeleteFileDOSsrc

Func: DeleteFileDOS **Pack:** DOSFilePack **Func:** 100/DOS

```
struct dosfilepack DFP;

DFP.func = DELETEFILEDOS;          /* defined in BULLET.H */
DFP.filenameptr = filename;
stat = BULLET(&DFP);
if (stat !=0) ...                  /* error */
```

RenameFileDOSsrc

Func: RenameFileDOS **Pack:** DOSFilePack **Func:** 101/DOS

```
struct dosfilepack DFP;

DFP.func = RENAMEFILEDOS;          /* defined in BULLET.H */
DFP.filenameptr = orgfilename;
DFP.newnameptr = newfilename;
stat = BULLET(&DFP);
if (stat !=0) ...                  /* error */
```

CreateFileDOSsrc

Func: CreateFileDOS **Pack:** DOSFilePack **Func:** 102/DOS

```
struct dosfilepack DFP;

DFP.func = CREATEFILEDOS;          /* defined in BULLET.H*/
DFP.filenameptr = filename;
DFP.attr = 0;                      /* normal file directory attribute */
stat = BULLET(&DFP);
if (stat !=0) ...                  /* error */
```

AccessFileDOSsrc

Func: AccessFileDOS **Pack:** DOSFilePack **Func:** 103/DOS

```
struct dosfilepack DFP;

DFP.func = ACCESSFILEDOS;          /* defined in BULLET.H*/
DFP.filenameptr = filename;
DFP.asmode = 0x42;                 /* attempt R/W DENY NONE access */
stat = BULLET(&DFP);
if (stat !=0) ...                  /* error */
```

OpenFileDOSsrc

Func: OpenFileDOS **Pack:** DOSFilePack **Func:** 104/DOS

```
struct dosfilepack DFP;

DFP.func = OPENFILEDOS;           /* defined in BULLET.H */
DFP.filenameptr = filename;
DFP.ASmode = 0x42;                /* open in R/W DENY NONE access */
stat = BULLET(&DFP);
if (stat !=0) ...                 /* error
                                     else DFP.handle set to handle of open file */
```

See: CloseFileDOSsrc

SeekFileDOSsrc

Func: SeekFileDOS **Pack:** DOSFilePack **Func:** 105/DOS

```
struct dosfilepack DFP;

DFP.func = SEEKFILEDOS;           /* defined in BULLET.H*/
DFP.handle = handle;
DFP.seekoffset = 0L;              /* position 0 relative EOF (get length of file)
*/
DFP.method = 2;                   /* seek from END of file */
stat = BULLET(&DFP);
if (stat==0) {
  /* DFP.SeekOffset set to absolute current offset
     in this case, the DFP.seekoffset equals then length of the file */
}
else
  /* error */
```

ReadFileDOSsrc

Func: ReadFileDOS **Pack:** DOSFilePack **Func:** 106/DOS

```
struct dosfilepack DFP;

DFP.func = READFILEDOS;           /* defined in BULLET.H */
DFP.handle = handle;
DFP.bytes = bytes2read;           /* 16-bit value, in this case 512 since that's */
DFP.bufferptr = dosbuff;          /* the size of dosbuff */
stat = BULLET(&DFP);
if (stat==0) {
  if (DFP.bytes != bytes2read) {     /* check if EOF processed */
    /* hit EOF before reading all 512 bytes */
  }
  else {
    /* ReadBuff filled with 512 bytes of data read from the current disk pos
       disk position moved to the last byte read + 1 */
  }
}
else
  /* error */
```

See: WriteFileDOSsrc

ExpandFileDOSsrc

Func: ExpandFileDOS **Pack:** DOSFilePack **Func:** 107/DOS

```
struct dosfilepack DFP;

DFP.func = EXPANDFILEDOS;          /* defined in BULLET.H */
DFP.handle = handle;
DFP.seekoffset = bytes2expandby;
stat = BULLET(&DFP);
if (stat==0) {
  /* file expanded by number of bytes specified */
else
  /* error */
```

WriteFileDOSsrc

Func: WriteFileDOS **Pack:** DOSFilePack **Func:** 108/DOS

```
struct dosfilepack DFP;

DFP.func = WRITEFILEDOS;          /* defined in BULLET.H */
DFP.handle = handle;
DFP.bytes = bytes2write;          /* 16-bit value, in this case 512 since that's */
DFP.buffptr = dosbuff;            /* the size of dosbuff */
stat = BULLET(&DFP);
if (stat=0xFFFE)                  /* -2 that is */
  /* disk full */
else
  if (rstat !=0) ...              /* error */
```

Unlike ReadFileDOS, if the number of bytes actually written does not equal bytes2write, the WriteFileDOS routine returns a DISK FULL error code (-2).

See: ReadFileDOSsrc

CloseFileDOSsrc

Func: CloseFileDOS **Pack:** DOSFilePack **Func:** 109/DOS

```
struct dosfilepack DFP;

DFP.func = CLOSEFILEDOS;          /* defined in BULLET.H */
DFP.handle =handle2close;
stat = BULLET(&DFP);
if (stat !=0) ... /* error
```

See: OpenFileDOSsrc

MakeDirDOSsrc

Func: MakeDirDOS **Pack:** DOSFilePack **Func:** 110/DOS

```
struct dosfilepack DFP;

DFP.func = MAKEDIRDOS;            /* defined in BULLET.H*/
DFP.filenameptr = newdirectoryname;
stat = BULLET(&DFP);
if (stat !=0) ...                 /* error */
```

The predefined key flags are:

cUNIQUE	1	
cCHAR	2	
cINTEGER	16	
cLONG	32	
cNLS	0x4000	note: cNLS is set by BULLET
cSIGNED	0x8000	

Credits:

Original Bullet for DOS documentation conversion to WinWord2, and WinHelp preparation, and impetus for this Windows version of Bullet, by Fabiano Fabris (FidoNet 2:285/304.100 or fab@fido.lu) during December 1993. Thanks, Fabiano. Post processing (the easy part) done by the author in WinWord6, August 18, 1994.